FCEFyN

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS y NATURALES

PROGRAMACIÓN CONCURRENTE

TRABAJO PRACTICO FINAL

Tomás Alejo Cisneros, 43.825.451, Ing. Computación

Profesores: Luis Orlando Ventre

Mauricio Ludemann

Año 2025

UNC

Índice

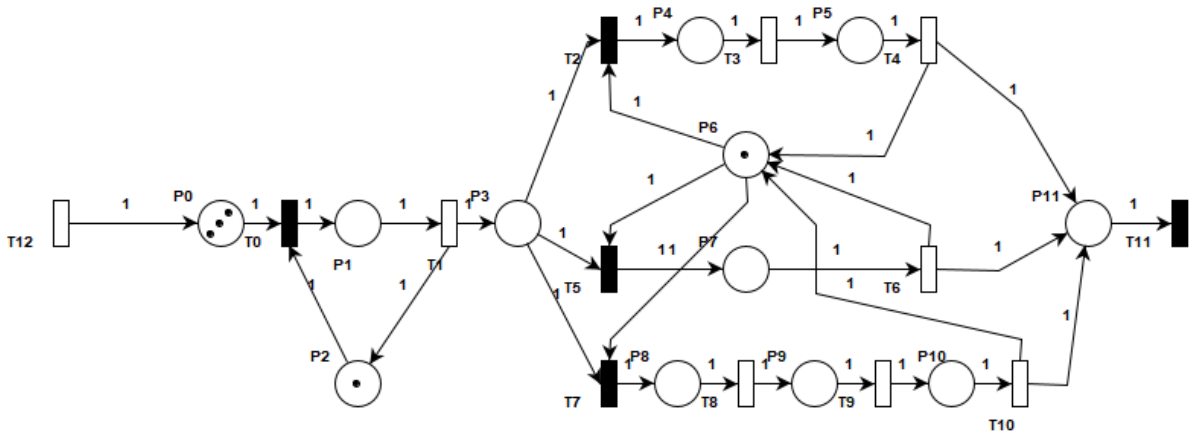
Introducción

En este trabajo, se llevará a cabo la implementación, ejecución y posterior análisis de un monitor de concurrencia para simular y ejecutar el modelo de un sistema diseñado en una Red de Petri. Para ello, se utilizará diversas herramientas, como PIPE (para el modelado de la Red de Petri) y distintos lenguajes de programación como JAVA y Python.

A través de este trabajo recorreremos todos los conceptos asociados a la Red de Petri, diagramas UML, **JAVA**, Python, expresiones regulares, entre otros. Todo esto, con el objetivo de realizar un proyecto integro que nos permita usar y aprovechar todo el abanico de herramientas a nuestro alcance para instruirnos en el mundo de la concurrencia y en el modelado de sistemas.

Desarrollo

Primeramente, se presentará nuestro sistema modelado en una Red de Petri para verificar y analizar las propiedades que nuestra red posee.



En este caso, nuestra Red de Petri representa un sistema encargado de procesar datos, las cuales tienen que pasar por 3 etapas: Cargado, Procesado y Exporte. En donde en la etapa de procesado, existen 3 segmentos que difieren en el procesado a aplicar.

- **Cargado/Loader:** Esta etapa de la red, se tiene una plaza P2 con 1 token, que representa la capacidad de este Loader. Se toman los tokens desde la plaza P0 y su respectivo buffer (entryBuffer) y luego del disparo de T0 y T1 se llega a nuestro segundo buffer: loadBuffer.
- **Procesado/Processor:** Esta etapa posee 3 segmentos, los cuales difieren debido a su complejidad. Desde P3 se presenta un conflicto que a través de nuestra politica, se resolverá.

    Si se elige T2, luego se dispara T3 y T4 (modo procesamiento medio)

    Si se elige T5, será solo T6 (modo procesamiento simple)

Si se elige T7, será T8, T9, T10 (modo procesamiento alto)

Luego de esto, el dato ya procesado, se entregará en un buffer listo para exportar, es decir el “exportBuffer”

- **Exporte/Exporter:** Ésta etapa solo consta de un solo segmento formado por T11 da fin al ciclo de procesado de un dato.

Los segmentos capados por una plaza, hace que dichos segmentos funcionen como una unidad atómica, es decir que su comportamiento es el de un bloque indivisible controlado por la disponibilidad de la plaza que lo limita.

También, podemos apreciar que las transiciones que terminan una etapa poseen un tiempo de disparo, no son transiciones inmediatas. Este tiempo, más adelante le asignaremos el nombre de alfa.

Sin embargo, hay que separar aquellas transiciones intermedias que no inician ni terminan una etapa pero si poseen tiempo, le asignaremos el mismo tiempo alfa.

Propiedades

Si tomamos en cuenta que la transición T12 (que alimenta a P0, la plaza que da inicio a todo) es una transición fuente, ésta posee un límite externo impuesto por la cantidad máxima de imágenes a procesar (en nuestro caso, será de 200). Por este motivo, la red **es acotada, ya que, en el contexto de nuestro sistema**, la cantidad de tokens nunca crecerá más allá del número de imágenes previstas para ser procesadas.

Respecto de la seguridad, una red se considera segura si en todo marcado alcanzable ninguna plaza supera 1 token. Como se observa en la ejecución del programa, plazas como P3 o P0 pueden contener más de un token simultáneamente, lo cual es coherente con el comportamiento esperado de nuestro programa. Por lo tanto, **la red no es segura**.

```
[Processor-1] Desde P3 (tokens = 4) se disparará T5
```

Ilustración 1: Hilo Processor, mostrando a P3 con más de 1 token.

En cuanto a la propiedad de **deadlock**, en **nuestra red no alcanza dicho estado** porque, para todo marcado alcanzable desde el marcado inicial, siempre existe al menos una transición sensibilizable. Esto se evidencia al analizar las etapas activadas por plazas como **P3, P11 y P0**, donde el consumo de tokens en una transición alimenta inmediatamente otra etapa del procesamiento de datos. De esta manera, mientras al menos una de estas etapas mantenga tokens, siempre habrá un disparo posible. En consecuencia, el sistema solo podría detenerse por una condición externa (como dijimos anteriormente, llegar al número máximo de imágenes a procesar) pero no por un deadlock inherente a la red.

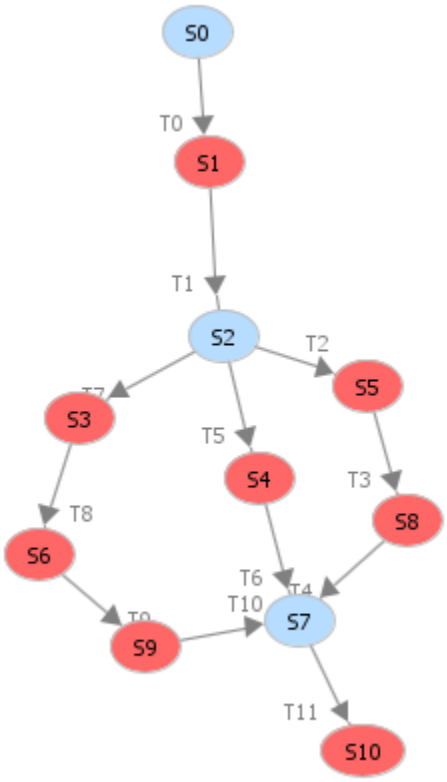


Ilustración 2: Grafico de alcanzabilidad, se aprecia que siempre se sensibilizará una transición desde el inicio del proceso.

En cuanto a la propiedad de **vivacidad**, bajo las políticas de disparo diseñadas para resolver los distintos conflictos del sistema, la red puede caracterizarse como **seudo-viva** y sus transiciones como **cuasi-vivas**.

Es **seudo-viva** porque, mientras que queden datos por procesar, siempre existe al menos una transición sensibilizable, de modo que la red nunca se bloquea completamente por un deadlock estructural, sino que sólo se detiene cuando se alcanza la condición externa de fin de procesamiento de todos los datos.

Al mismo tiempo, las transiciones son **cuasi-vivas**, porque la política de elección en los conflictos asegura que, en el conjunto de las 200 invariantes, cada transición del modelo se sensibilice y dispare al menos una vez (ninguna queda sin usar durante el funcionamiento).

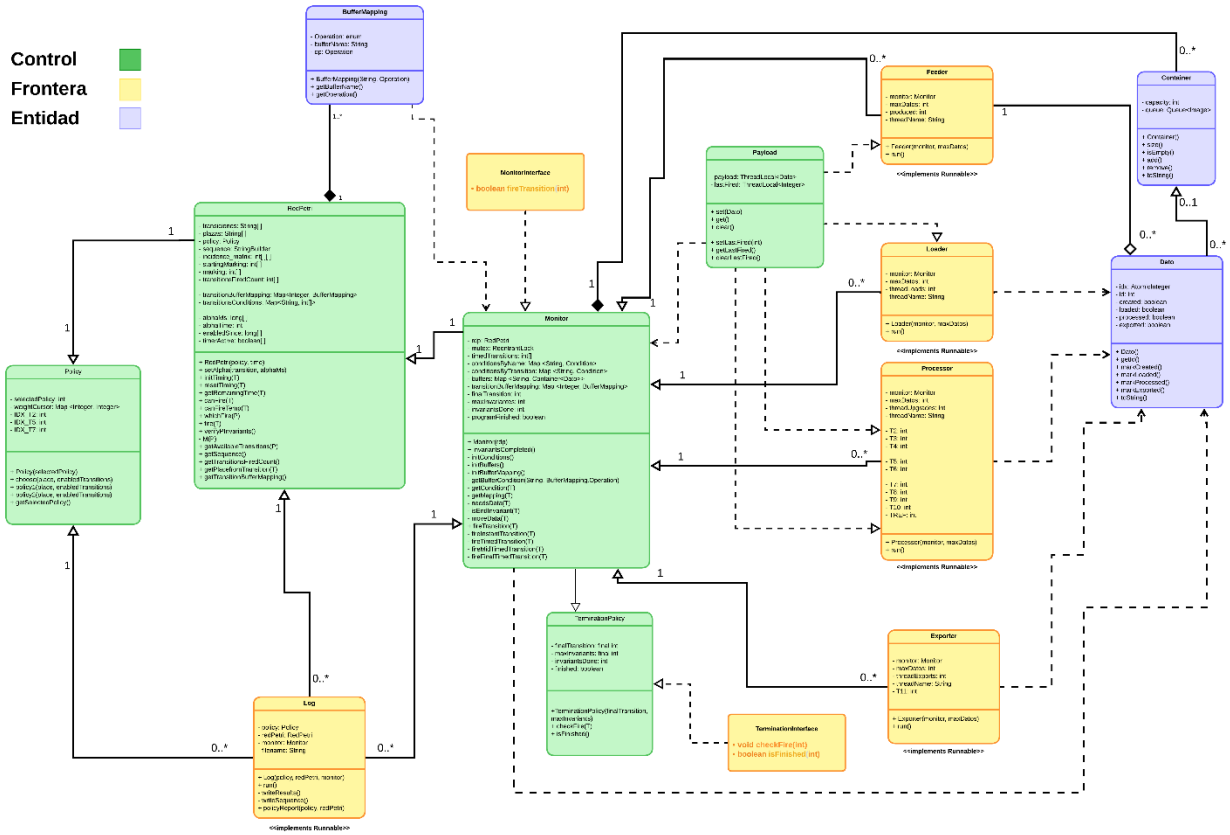
Modelado de la Red de Petri en JAVA

Teniendo en cuenta el enunciado sobre nuestra Red de Petri y sabiendo que debemos hacer uso de un monitor de concurrencia para guiar la correcta ejecución de nuestra red, procederemos a programar tanto el monitor como las clases del sistema:

“En la Figura 1 se observa una red de Petri que modela un sistema de procesamiento de datos, donde las plazas P2 y P6 representan recursos compartidos del sistema: la plaza P2 modela el bus de acceso al buffer, mientras que la plaza P6 representa la unidad de procesamiento. Por otro lado, las plazas P3 y P11 corresponden a buffers, siendo P3 el buffer de datos a procesar y P11 el buffer de salida. La plaza P0 se define como una plaza idle asociada a la cola de arribo de datos al sistema, y la plaza P1 representa un dato accediendo al buffer a través del bus. Cada dato puede ser procesado mediante uno de tres modos de procesamiento posibles: un modo de complejidad simple, compuesto por una única etapa en la plaza P7; un modo de complejidad media, conformado por dos etapas en las plazas P4 y P5; y un modo de complejidad alta, que consta de tres etapas realizadas en las plazas P8, P9 y P10.”

Primeramente, realizaremos los diagramas tanto de clases como de secuencias en lenguaje UML.

Diagrama de Clases



Podemos apreciar que nuestro diagrama tiene 3 colores que representan las características que definen a cada clase.

- **Control:** Coordinan el flujo del sistema, aplica reglas de negocio y orquesta la interacción entre Frontera y Entidad. Contiene la lógica del proceso.

- **Frontera**: Son actores que interactúan con el núcleo del sistema (Monitor / RDP / Policy) desde afuera, pidiendo y entregando imágenes.
- **Entidad**: No coordinan procesos, no manejan hilos, no deciden qué transición disparar, solo representan *cosas* dentro del modelo.

Relaciones del Diagrama de Clases

Las relaciones entre las clases reflejan el flujo de procesamiento, la coordinación entre hilos y la administración del estado de las imágenes a lo largo del sistema. Se describen a continuación, las relaciones representadas en el modelo:

Relaciones entre clases de Control

Las clases **Monitor**, **RedPetri** y **Policy** se encuentran inherentemente asociadas por su rol de coordinación del sistema.

- **Monitor – RedPetri (1 a 1):**
  Cada monitor administra exactamente una instancia de la Red de Petri, responsable de evaluar transiciones, gestionar sensibilización y registrar la secuencia de disparos. El monitor depende directamente de la lógica que provee la RdP.
- **RedPetri – Policy (1 a 1):**
  La Red de Petri utiliza una instancia de Policy para determinar de qué manera se resolverán los conflictos presentes en nuestra RdP. La relación demuestra que la definición de políticas es imprescindible para la ejecución de la RdP.
- **Monitor – TerminationPolicy (1 a 1):**
  El Monitor mantiene una relación de composición con TerminationPolicy, ya que la crea, la posee y controla su ciclo de vida para aplicar la política de finalización del sistema.

Relación con clases Frontera (Workers)

Las clases **Feeder, Loader, Processor, Exporter, Log y la interfaz del monitor** son actores externos que interactúan con las clases de control del sistema. Cada una se asocia con el Monitor mediante una relación **1 a 0..*** (menos MonitorInterface que tiene una relacion de realizacion/implementación), lo que expresa que:

- Un **Monitor** coordina múltiples workers del mismo tipo, bajo el estandar Controller - Workers.
- Cada worker depende de un único Monitor durante toda su ejecución, ya que este provee los mecanismos de sincronización y acceso a recursos compartidos.

Estas relaciones modelan el flujo de trabajo concurrente, donde los threads operan sobre imágenes en distintas etapas, bajo la tutela del Monitor.

Relaciones entre Control y Entidad

Las entidades principales del sistema son **Image** y **Container**. Contienen el estado del procesamiento y representan los recursos gestionados por el Monitor.

- **Monitor – Container (1 a muchos):**
  El monitor administra tantos buffers o contenedores distintos como la red de petri imponga, son necesarios para organizar el procesamiento en etapas. Poseen una relación de composición, ya que el monitor crea a los contenedores y si el monitor no existiera, los contenedores tampoco.

- **Monitor – Dato:**
  El monitor no almacena datos directamente como atributos, sino que interactúa con ellas temporalmente durante la ejecución de métodos. Poseen una relación de dependencia.

Relaciones entre Entidades

- **Container – Dato (1 a muchos):**
  Cada contenedor puede almacenar cero o muchos Datos simultáneamente.
  A su vez, un Dato puede encontrarse en **ninguno o solo un contenedor** a la vez, representado mediante la multiplicidad **0..1**.

Dependencias de los Workers hacia Image

Las clases Frontera conformado por los workers y una clase Payload, presentan relaciones de **dependencia** hacia Dato *(excepto **con el metodo Feeder, que presenta una relación de agregación)***, representadas mediante líneas punteadas en el diagrama. Esto indica que los workers utilizan las imágenes de manera temporal, ya que las obtienen desde el *Monitor*, las procesan y luego las entregan al siguiente buffer o al Payload sin conservarlas como parte de su estado interno.

Diagrama de Secuencia

Debido al tamaño de nuestro diagrama de secuencia, se brinda la posibilidad de visualizarla a través de un hipervínculo. El diagrama puede consultarse aquí: [Enlace]

El diagrama de secuencia tiene como propósito representar de manera detallada el disparo de una transición controlado por el monitor, haciendo referencias a los distintos usos de las clases Policy, RedPetri, Payload, y algunos elementos importantes a considerar como el mutex del monitor.

Para ello, nuestro diagrama expone la interacción entre los Workes en general y el *Monitor*, quien actúa como la unidad principal de coordinación y el manejo concurrente con la incorporación del *Mutex*. De este modo, se observa claramente como maneja el monitor las distintas solicitudes de disparo de los Workers mientras que cumple con la logica otorgada por la Red de Petri.

A su vez, el diagrama también marca claramente dónde aparecen los **mecanismos de concurrencia** del sistema. Se puede ver cuándo el Monitor aplica exclusión mutua para proteger el marcado de la RdP y cómo usa variables de condición para despertar o bloquear hilos según la disponibilidad de los Datos en las distintas etapas.

Además, el diagrama muestra cómo funciona la **semántica temporal** de las transiciones: en qué momento se inicia o continua el timer, cuándo se comprueba, si ya se alcanzó el tiempo mínimo (alfa) y cómo se reinicia el contador después de disparar la transición.

Básicamente, el diagrama resulta muy útil para explicar el comportamiento general del sistema y para ver cómo interactúan los hilos y la Red de Petri durante el procesamiento completo de una imagen.

Secuencia completa de una invariante

Para poder obtener un seguimiento más profundo del diagrama que modela la secuencia entera de una invariante se recomienda descargar el proyecto entero. Lo puedes descargar aquí:

Primero, empezaremos explicando el flujo general de nuestro monitor a la hora de disparar una transicion:

- Flujo general: fireTransition(T)
  - **Un hilo que representa alguna etapa (Feeder/Loader/Processor/Exporter) llama a:**
    - **monitor.fireTransition(T)**
  - **El Monitor decide si T es:**
    - **temporal: llama a fireTimedTransition(T)**
    - **instantánea: llama a fireInstantTransition(T)**

- Disparo transicion con semantica temporal: fireTimedTransition(T):
  - **La lógica temporal se divide en 2 tipos, según su clasificación:**
    - Si la transicion tiene alguna condicion o mapeo designado, se llama a:
      - Transición temporal final: fireFinalTimedTransition(T)
    - Si no, se llama a:
      - Transición temporal intermedia: fireMidTimedTransition(T)

2.1. Temporal Intermedia

- Mientras el sistema no haya finalizado:
  - el hilo intentará disparar la transición.
- **Entrada a la sección crítica**
  - Se adquiere el lock (mutex.lock()).
  - Se verifica:
    - Rdp.canFire(T).
- **Si la transición no está sensibilizada:**
  - El hilo se bloquea con condition.await().
- **Inicialización y evaluación del timer:**
  - Se llama a:
    - Rdp.initTiming(T).
  - Se inicia el conteo asociado a la transición.
  - Se evalua el cumplimiento del tiempo minimo necesario:
    - Rdp.canFireTemp(T)
    - **Si el tiempo aún no se cumplió:**
      - Se libera el lock.
      - El hilo duerme Thread.sleep(remainingTime)

- **Disparo temporal**
  - Cuando el tiempo se cumple:
    - Rdp.fire(T),
    - Payload.setLastFired(T),
    - Rdp.resetTiming(T).
  - El método retorna **true**.

2.2. Temporal Final

- **Operaciones adicionales, se repiten los pasos anteriores y:**
  - se carga nuestra ultima transicion en Payload.setLastFired(T),
  - se realiza moveData(T) tras el disparo, que setea el dato en Payload
  - Tras el disparo exitoso:
    - signalAllOnCondition(T) despierta a los hilos bloqueados,
    - El lock se libera,
    - el método retorna true o false si el programa finaliza o el hilo es interrumpido.

- Disparo de transición instantánea (fireInstantTransition(Tref))
  - **Entrada a la sección crítica (exclusión mutua)**
    - El Monitor adquiere el lock:
      - mutex.lock().
    - A partir de este punto:
      - solo un hilo puede interactuar con la Red de Petri,
      - consultar buffers,
      - y modificar el estado del sistema.
  - **Bucle principal de disparo**
    - Dentro de un while (!programFinished):
      - el hilo intentará disparar la transición hasta lograrlo,
      - o hasta que el sistema finalice.
  - **Resolución del conflicto de disparo**
    - El Monitor identifica la plaza asociada a la transición de referencia:
      - place = Rdp.getPlacefromTransition(Tref).
    - Se consulta a la Red de Petri:
      - chosen = Rdp.whichFire(place).

    Interpretación:
    - whichFire(place):
      - Asocia la plaza recibida como argumento, con las transiciones habilitadas en esa plaza.

- Ya con esos datos recabados, los pasa a la clase Policy, donde se resolverá el conflicto, y que según la politica elegida se elegirá una transición.

Si no hay transición válida

- Si chosen < 0:
  - no existe ninguna transición disparable desde esa plaza.
- El hilo:
  - se bloquea con condition.await(Tref),
  - y vuelve a intentar cuando sea despertado.

- **Verificación de disponibilidad de datos (buffers)**
  - Si la transición elegida **consume datos**:
    - needsData(chosen) == true.
  - El Monitor obtiene el buffer correspondiente a la transición.

  Si el buffer está vacío:
    - No se puede disparar la transición.
    - El hilo:
      - se bloquea con awaitBufferHasData(bufferName),
      - espera hasta que otro hilo agregue datos,
      - y luego reintenta.

- **Verificación de habilitación por sensibilizado:**
  - Se consulta:
    - Rdp.canFire(chosen).

  Si la transición no está habilitada
    - El marcado actual no permite el disparo.
    - El hilo:
      - se bloquea con condition.await(chosen),
      - y vuelve a intentar cuando el estado de la red cambie.

- **Disparo efectivo de la transición**

Cuando todas las condiciones se cumplen:

- El Monitor ejecuta:
  - Rdp.fire(chosen).
  - Payload.setLastFired(chosen):
    - se guarda qué transición fue disparada
  - moveData(chosen):
    - se mueve o consume el dato asociado al disparo.

  Finalmente:
  - el método retorna true.

- **Salida de la sección crítica**
  - En el bloque finally:
    - se libera el lock con mutex.unlock(), permitiendo que otros hilos ingresen al Monitor.

Política

Tenemos una clase Policy encargada de **resolver conflictos** cuando en una misma plaza de la Red de Petri, hay **más de una transición habilitada**, en este caso, nuestra red solo posee un **unico conflicto** que se ubica en la plaza P3.
El método central es:

```
public int choose(int place, int[] enabledTransitions)
```

Donde:

- **Place:** identificador de la plaza (P0, P6, P14, etc.).
- **enabledTransitions:** lista de transiciones habilitadas en esa plaza.
- Devuelve **transición elegida** o un valor negativo si no se puede elegir.

Además, se maneja el concepto de elegir entre **politica 1/politica 2**, según una variable llamada **"selectedPolicy"**

- Si **selectedPolicy** es igual a 1, se ejecutará el programa en base a la politica 1.
- Si **selectedPolicy** es igual a 2, se ejecutará la segunda politica del programa.

Política 1 – Elección al azar entre los 3 modos de procesamiento

Nuestra primera politica es la encargada de seleccionar aleatoriamente entre las 3 transiciones disponibles para disparar (T2, T5 y T7), dichas transiciones se corresponden a un modo de procesado de datos especifico, por ejemplo, para T2, es el modo de procesado MEDIO.

- Si solo hay una transicion habilitada desde una plaza, se elige esa unica transicion.
- Si hay conflicto (cuando el dato se encuentra depositado en P3), la transición a disparar se selecciona mediante un valor aleatorio generado por *Math.random()*, que se mapea sobre el conjunto de transiciones habilitadas.

Pseudo codigo:

```
function policy1(place, enabledTransitions[]):

    si enabledTransitions[] está vacía:
        retornar -1

    si hay una sola transición habilitada:
        retornar esa transición

    si hay conflicto varias transiciones habilitadas en P3:
        Genera un r aleatorio en el rango [0,1)

        Variable paso //cantidad de transiciones habilitadas

        Variable índice //la parte entera (r / paso)

        si índice ≥ cantidad de transiciones habilitadas:
           índice = cantidad de transiciones habilitadas – 1

       retornar enabledTransitions[índice]
```

Política 2 – Distribución 60/20/20 entre T2, T5 y T7

Nuestra segunda política está pensada de igual manera que la politica 1, para **arreglar específicamente el conflicto presente en la plaza P3,** donde se tiene que elegir entre las transiciones **T2, T5 y T7.**

En esta politica sin embargo, la estrategia es diferente. Ya no se tomará en cuenta una selección aleatoria, sino un **comportamiento bien determinista frente a la elección de las transiciones**: en nuestra red de 200 disparos, 120 van a ser pertenecientes a T5 (perteneciente al modo SIMPLE de procesado) y los 80 disparos restantes se repartiran equitativamente entre T2 y T7 (correspondientes a los modos MEDIO y ALTO respectivamente), estableciendo un reparto de disparos 60/20/20.

- Si T2, T5 y T7 están habilitadas (conflicto en P3):
    - A través de un **round robin con peso/ponderado** resolveremos el conflicto.
        - Con un cursor de **5 estados** (estado inicial = 0):
            - 0,1,2 corresponden al disparo de T5
            - El indice 3 le pertenerá a T2
            - Finalmente, el indice 4, se le asigna a T7
        - Una vez seleccionada la transicion, el cursor se incrementa y se reinicia al alcanzar el ultimo estado, garantizando el siguiente comportamiento de elección de disparos:

          T5 → T5 → T5 → T2 → T7 → T5 → T5 → T5 → T2 → T7 → ...

- Si no existe conflicto, se elije la primera habilitada.

Pseudo codigo:

```
function policy2(place, enabledTransitions):

    si enabledTransitions[] está vacía:
        retornar -1

    comprobamos si estamos en P3 (T2, T5 y T7 estarán habilitadas):

         hasT2,hasT5, hasT7 = false;

    para cada transicion en enabledTransitions:
        si transition == T2:
            hasT2 = verdadero
        si transition == T5:
            hasT5 = verdadero
        si transition == T7:
            hasT7 = verdadero


    si evidentemente estamos en P3:
         si hasT2 y hasT5 y hasT7:

        counter = currentCursorValue(place)   // valor por defecto: 0
        selectedTransition = indefinido // transición a elegir

        si counter < 3:
            selectedTransition = T5
        sino si counter == 3:
            selectedTransition = T2
        sino:
            selectedTransition = T7

        updateCursor(place, (counter + 1) mod 5) //Avanzamos el cursor

        retornar selectedTransition

    si no hay conflicto:
         se elige la primera transición habilitada
        retornar enabledTransitions[0]
```

Analisis de politica

Ya con nuestro proyecto diagramado, implementado y funcionando. Procederemos a realizar el analisis del mismo, en este caso, empezaremos con el analisis de la politica:

Para ello, guardamos en un archivo log, los resultados del uso de nuestra política en la red para resolver los conflictos. Se hicieron abundantes ejecuciones, sin embargo, solo se mostrarán las 4 más relevantes y distintas para poder llegar a una conclusión certera.

Política 1

Empezaremos observando el rendimiento de nuestra política 1 (los datos se procesan utilizando cualquier modo al azar)

- **resultados_2025-12-15_10-34-53**

Transición	Modo	Datos Procesados	TOTAL DISPAROS
T2	Proc. Bajo	60	200
T5	Proc. Medio	69	
T7	Proc. Alto	71	

- **resultados_2026-01-31_21-46-56**

Transición	Modo	Datos Procesados	TOTAL DISPAROS
T2	Proc. Bajo	43	200
T5	Proc. Medio	76	
T7	Proc. Alto	81	

- **resultados_2025-12-15_11-21-31**

Transición	Modo	Datos Procesados	TOTAL DISPAROS
T2	Proc. Bajo	84	200
T5	Proc. Medio	60	
T7	Proc. Alto	56	

- **resultados_2025-12-01_23-00-23**

Transición	Modo	Datos Procesados	TOTAL DISPAROS
T2	Proc. Bajo	57	200
T5	Proc. Medio	82	
T7	Proc. Alto	61	

Estos datos nos presentan un comportamiento no determinístico, evidenciado por la variabilidad de la cantidad de datos procesados en los tres modos aun manteniendo constante el número total de disparos. Esta variación es consecuencia directa del uso de una política de selección aleatoria, la cual no aplica ningún criterio de prioridad ni de complejidad al momento de resolver el conflicto en la plaza P3. Como resultado, todos los modos poseen la misma probabilidad de ser ejecutados, evitando situaciones de inanición, pero sin garantizar una distribución uniforme .

Política 2

Ahora realizaremos el análisis de la segunda política, recordando que se trata de una prioridad de un 60% para el modo SIMPLE de procesamiento, frente al 20% de prioridad para los modos restantes, esto teóricamente nos deja con un numero fijo de ejecuciones, siendo 120 datos procesados en el modo SIMPLE y 40 para el modo MEDIO y ALTO de procesamiento. Como en las ejecuciones realizadas se vió un comportamiento muy marcado, solo se mostrará un archivo log, que refleja los resultados de todos los restantes logs donde se aplicó la segunda política.

- **resultados_2025-12-01_23-12-29**

Transición	Modo	Datos Procesados	Porcentaje	TOTAL DISPAROS
T2	Proc. Bajo	120	60%	200 (100%)
T5	Proc. Medio	40	20%	
T7	Proc. Alto	40	20%	

Los resultados observados muestran que el sistema **siempre termina acomodándose a la proporción del 60/20/20**. Esto ocurre por la lógica de **la política que decide tres veces por T2 para luego elegir una vez por T5 y una vez T7**. De esta manera, no queda lugar al azar: se repite exactamente el mismo comportamiento en todas las ejecuciones, en contraposición de la ejecución no determinística de la política 1. Esta estabilidad confirma que la **política logra su objetivo**: priorizar el procesado hacia el modo SIMPLE.

Análisis de tiempos

Realizando un primer acercamiento hacia la implementación de la semántica temporal, es necesario establecer un tiempo ***alfa*** arbitrario para luego, a partir de este alfa obtener un primer tiempo tope de ejecución de nuestro sistema. En nuestro caso, se partió del análisis del comportamiento del sistema en un **modo secuencial**, ya que este enfoque permite estimar el **tiempo máximo de ejecución posible** para el sistema: al no existir concurrencia entre etapas, cada operación debe completarse íntegramente antes de habilitar la siguiente, generando así el peor caso temporal.

Bajo esta perspectiva, tomamos la **invariante de transición** que presenta el **menor tiempo total** de procesamiento, ya que representa nuestro mejor caso de ejecución sin concurrencia. De este modo, ya cualquier implementación que añada concurrencia tiene como objetivo lograr un tiempo de ejecución igual o mejor a esta referencia. **Esta invariante** se concibe como una secuencia lineal donde el dato siempre es procesado en el modo SIMPLE, ya que esto nos ahorra hasta 2 etapas adicionales de procesamiento, las cuales **también demandan tiempo de ejecución.**

La duración total surge de la suma de los tiempos ***alfa*** asociados a las transiciones temporales.

Etapa	Transición temporal	alfa	Acumulado
Creación	T12	60 ms	**60 ms**
Carga	T1	60 ms	**120 ms**
Procesado	T6	60 ms	**180 ms**

Considerando a nuestro alfa como 60ms, tenemos que una invariante completa todas las etapas de nuestro programa en aproximadamente 180ms. Tomando en cuenta que ejecutaremos no una sola invariante sino 200, el tiempo total que tomaría nuestro programa en finalizar, seria:

$$180ms * 200 = 36000ms$$

Básicamente, una ejecución completa procesando las 200 invariantes nos da alrededor de 36 segundos.

Ya aplicando la concurrencia usando la cantidad necesaria de hilos para obtener el mayor paralelismo posible y partiendo desde la política 1, tomamos 20 muestras del tiempo total de ejecución de nuestro programa y llegamos a un promedio:

Muestras de tiempo utilizando POLITICA 1 (en ms):

25124	**24523**	**25072**	**25451**	**24401**
25109	**25280**	**24172**	**26524**	**23791**
24868	**23904**	**25173**	**24470**	**24844**
25737	**25294**	**25530**	**25259**	**24379**

Tomando un promedio de esas muestran, nos dá un aproximador de **24945ms**.

Ya con el tiempo promedio de la ejecución del programa aplicando la política 1, hagamos lo mismo con la política 2 para apreciar si varía el tiempo o no y si lo hace, de qué forma.

Muestras de tiempo utilizando POLITICA 2 (en ms):

20996	**20322**	**20887**	**21088**	**20086**
20111	**20088**	**20219**	**20338**	**20245**
20313	**20428**	**20279**	**20220**	**20452**
20158	**20186**	**20264**	**20393**	**20151**

Podemos apreciar que el promedio de nuestro sistema ejecutándose en base a la política 2 es de **20361ms**.

Ya sea que nuestra red se ejecute con la política 1 o la política 2, las dos vencen el tiempo mínimo que una ejecución secuencial podría ofrecer, si somos más precisos: la **política 1** implica una mejora del **30.7%** mientras que la **política 2** ofrece una mejora del **43.4%** frente a los 36s del secuencial.

Para redondear los resultados, se podría decir que a partir de los tiempos obtenidos puede observarse que la concurrencia implementada en la red de Petri permite una reducción significativa del tiempo total de ejecución respecto del caso secuencial mínimo. La mejora obtenida demuestra que la red es capaz de ejecutar múltiples invariantes de manera concurrente y que la política de disparo influye directamente en el rendimiento del sistema.

Sin embargo, hay que analizar también los casi 5 segundos menos de ejecución de la **política 2 frente a la política 1**.

El factor principal de esa diferencia es la forma en la que la **política 2** realiza la ejecución de los disparos. Como ya especificamos en el apartado de política, lo hace de manera controlada, donde de los 200 disparos realizados, el 60% corresponde a un procesado simple, compuesto por una única etapa de 60 ms, mientras que el 20% corresponde a un procesado medio de dos etapas y el 20% restante a un procesado alto de tres etapas, todas con una duración de 60 ms por etapa.

Esta estrategia permite que la mayoría de los invariantes atraviesen un camino de procesamiento corto, reduciendo el tiempo promedio de permanencia en la red y a su vez, el uso de los recursos compartidos.

En contraste, la **política 1** selecciona el modo de procesamiento de manera aleatoria entre los tres posibles, lo que genera una mayor concentración de invariantes con procesados medio y alto. Esta **aleatoriedad introduce mayor variabilidad en los tiempos de ejecución** (por eso podemos ver ejemplos que van desde 23791ms hasta los 26524) **y aumenta la probabilidad de usar por más tiempo, el único recurso compartido** de la etapa de procesamiento, incrementando el tiempo total necesario para completar los disparos.

Análisis de Invariantes

Invariantes de Transiciones

Un invariante de transición representa un ciclo completo de transiciones que devuelve a la red a un marcado equivalente al inicial y describe en nuestro caso, el procesamiento completo de un dato a través de todas las etapas del sistema.

Para poder aplicar este análisis con herramientas (PIPE y luego una posterior verificación de las invariantes en Python) e incluso poder llevar a cabo el proyecto principal, fue necesario modificar la red, agregando una nueva transición **T12**, ya que la versión original no tenía ninguna transición que "alimente" con tokens a P0. Dicho esto, **T12** modela el arribo de nuevos datos al sistema y su tiempo representa el intervalo de llegada de los datos desde el exterior.

Con este ajuste, **PIPE** pudo detectar **tres invariantes**, cada uno correspondiente a una variante posible del recorrido de un dato según las decisiones tomadas en el único conflicto de la red:

- **T2 vs T5 vs T7,**

Las tres invariantes de transición identificadas son:

N.º	Invariante de Transición
IT1	T12 - T0 – T1 – T2 – T3 – T4 – T11
IT2	T12 - T0 – T1 – T5 – T6 – T11
IT3	T12- T0 – T1 – T7 – T8 – T9 – T10 – T11

Cada uno de estos recorridos refleja una ejecución válida del sistema. Para poder comprobar si nuestro sistema dispara la RdP de una forma correcta, analizaremos las ejecuciones e intentaremos detectar estos recorridos observando un archivo de secuencia con las transiciones disparadas.

La complicación viene al ver el archivo de secuencia, el cual no tiene las invariantes explicitas en él, sino que todas las transiciones se encuentran solapadas entre si debido al uso de la concurrencia. Para poder filtrar las distintas invariantes en un archivo con los disparos solapados, se hará uso de las expresiones regulares.

Las expresiones regulares o REGEX son **un patrón de búsqueda** que sirve para identificar, validar, extraer o reemplazar texto dentro de una cadena.

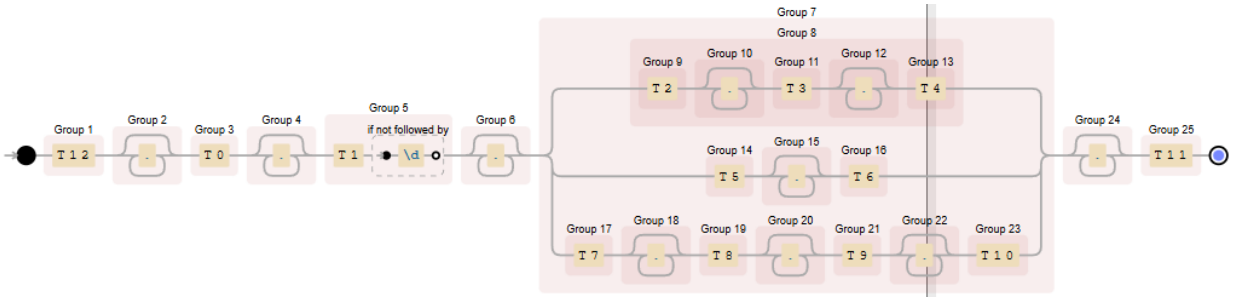
Para establecer qué patrón REGEX vamos a utilizar, observemos el formato de nuestro archivo de secuencia:

```
"T12T0T12T1T7T0T1T8T12T0T9T12T1T0T10T12T1T11T7T0T8T1T12T0T12…"
```

Vemos que, además del solapamiento mencionado, todas las transiciones se encuentran pegados una a la otra y no existe ningún tipo de separación entre ellas. Ya observado el formato de la secuencia, buscaremos un REGEX adecuado para filtrar los distintos patrones de ejecución.

```
(T12)(.*?)(T0)(.*?)(T1(?!\d))(.*?)(((T2)(.*?)(T3)(.*?)(T4))|(T5)(.*?)(T6)|(T7)(.*?)(T8)(.*?)(T9)(.*?)(T10))(.*?)(T11)
```

Gráficamente, esta REGEX procesa los datos de la siguiente manera:



Esta expresión detecta **un ciclo completo de transiciones**, permitiendo variaciones en los puntos de conflicto y aceptando cualquier cantidad de contenido intermedio.

El operador ".*?" se emplea como cuantificador no codicioso/lazy/vago, permitiendo consumir cualquier secuencia de caracteres, pero en su mínima extensión posible, de manera que la expresión regular respete el orden de las transiciones sin comerse ocurrencias intermedias.

- Si no tuviera el "?", el operador estaría en modo codicioso y podría procesar todas las transiciones que se encuentren entre, por ejemplo, T0 y el **ultimo** T1
- Con el carácter "?" cambiamos de modo al operador y procesa cualquier dato que esté en el medio entre, por ejemplo, T0 y el **primer** T1 siguiente.

Por otra parte, el uso de la construcción "?!\d" implementa un *control* que verifica que después de una transición como T1 no haya un dígito, garantizando así que el patrón distinga correctamente entre T1 y transiciones como T10, T11, etc. Que poseen un inicio "T1" seguido de un digito. Por eso:

- T1(?!\d): Coincide con **T1 solo si T1 no está seguido de un digito.**

Ya con nuestra REGEX establecida, pasaremos a verificar el funcionamiento de la misma en nuestro sistema, para ello, usaremos una librería “re” específica para Python y definiremos una función **analyzewithRegex**:

Pseudo Codigo

```
funcion analizarSecuencia(secuencia):

    definir patronRegex      // patrón que reconoce un ciclo T12..T11
    definir reemplazo        // grupos usados para reconstruir la
                                parte "no invariante"

    remaining = secuencia
    iteracion = 0

    repetir indefinidamente:

        iteracion = iteracion + 1

        (cleaned, count) = aplicarSubstitucionRegex(patronRegex,
reemplazo, remaining)

        // re.subn hace dos cosas:
        //   - residual → la nueva secuencia tras sustituir los
            matches del patrón
        //   - count   → cuántas veces se encontró el patrón (cuántos
            ciclos T12..T11 detectó)

        si count == 0:
            salir del bucle   // no se encontraron más invariantes

        remaining = residual   // actualizo y sigo limpiando

    // fin del bucle

    si remaining está vacío (o solo espacios):
        imprimir "Secuencia vacia, invariantes validas”
    sino:
        imprimir "Quedaron transiciones fuera de lugar"
        mostrar remaining
```

Internamente, el patrón está dividido en **grupos capturados** mediante paréntesis, de modo que cada bloque T12 hasta T11 se separa en sus transiciones principales y en el texto intermedio que las rodea. El reemplazo no borra todo el match, sino que reconstruye la secuencia usando solo algunos de esos grupos (\g<2>, \g<5>, etc.), preservando el contenido “suelto” entre invariantes y descartando las transiciones correctamente formadas. Si al finalizar el proceso no queda contenido residual, se concluye que toda la secuencia se descompuso en

invariantes de transición válidos. Si no, el remanente indica transiciones fuera de orden o errores en el patrón regex cargado.

Salida de nuestro programa:

```
[Regex] Iteraciones realizadas: 6
[Regex] La secuencia quedó vacía. Todas las transiciones cumplen el patrón.

Log generado correctamente en: invariantes_2025-12-05_02-54-03.log
```

En Log "invariantes":

```
Archivo analizado: secuencia_2025-12-15_18-03-03.log

[INVARIANTES POR TIPO]
T12-T0-T1-T2-T3-T4-T11: 43
T12-T0-T1-T5-T6-T11: 76
T12-T0-T1-T7-T8-T9-T10-T11: 81

Total de invariantes detectados: 200

No quedó nada sin filtrar. Todos los invariantes fueron clasificados correctamente.
```

Invariantes de Plazas

Una **invariante de plaza** es una propiedad de la red de Petri que se mantiene constante para todo marcado alcanzable. Matemáticamente, se expresa como una combinación lineal de plazas cuya suma de tokens permanece inalterada sin importar qué secuencia de transiciones se dispare. Expresarlas en forma de ecuación permite identificar recursos que nunca se crean ni destruyen, sino que únicamente se redistribuyen entre diferentes estados del sistema. En nuestra red, estas invariantes verifican que las etapas de cargado (Loader) y de procesado de los datos (Processor), respeten su capacidad máxima, garantizando que no se carguen y/o procesen más datos de los permitidos, que no existan situaciones de desbordamiento ni pérdida de tokens.

N.º	Invariante de Plazas (ecuaciones)
IP1	$M(P1) + M(P2) = 1$
IP2	$M(P10) + M(P4) + M(P5) + M(P6) + M(P7) + M(P8) + M(P9) = 1$

Con esas ecuaciones podemos observar que:

I1: $M(P1) + M(P2) = 1$

Representa la etapa del Loader, entre P1 y P2 existe exactamente un token, esto nos denota que P2 es una plaza impuesta para limitar la capacidad de P1.

I2: $M(P10) + M(P4) + M(P5) + M(P6) + M(P7) + M(P8) + M(P9) = 1$

Este marcado nos da a entender que la etapa de procesado solo se puede llevar a cabo en un solo modo de procesado a la vez y con únicamente un dato a procesar.

Análisis de hilos

Hilos mínimos por etapa según Caso 1 y Caso 2

A partir de los invariantes de transición obtenidos en el análisis anterior y aplicando las reglas establecidas en los casos 1 y 2, es posible determinar la cantidad mínima de hilos para aprovechar el mayor paralelismo posible.

Caso 1

El Caso 1 establece que:

"Si un invariante de transición presenta un conflicto con otro invariante, debe existir un hilo encargado de las transiciones anteriores al conflicto, y un hilo por cada rama o camino alternativo generado por dicho conflicto."

En nuestra RdP, se identificó un único conflicto:

Conflicto: T2 / T5 / T7 (etapa Processor)

Los invariantes se dividen en tres familias una vez que el dato llegue a P3:

- IT1 continua por T2
- IT2 continua por T5
- IT3 continua por T7

Por lo tanto, como mínimo, la presencia de 3 hilos Processor, uno para cada rama del conflicto.

Caso 2

El Caso 2 establece que:

"Cuando dos o más invariantes presentan un join en una misma plaza, las transiciones posteriores al join deben ser ejecutadas por tantos hilos como tokens simultáneos existan en dicha plaza. Si la RdP solo admite un token simultáneo en ese punto, el mínimo estructural es 1 hilo."

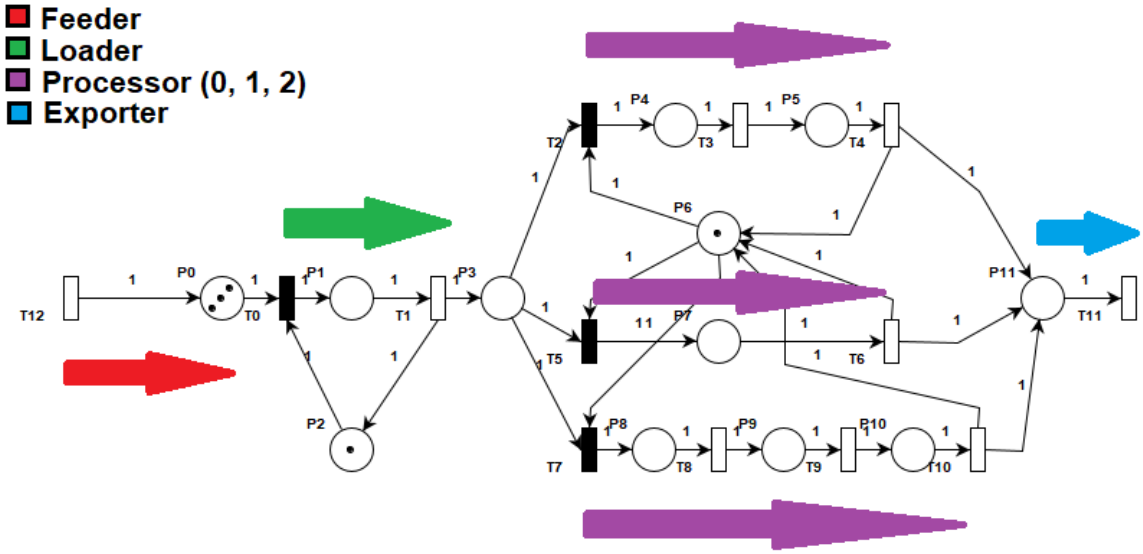
Al igual que en el caso 1, solo una plaza posee esta propiedad de que dos o más invariantes realizan un join, se trata de **P11,** esta plaza cumple el rol de **recibir el dato ya procesado** (De la etapa Processor respectivamente), restaurando un camino único para luego proceder a la etapa de exportado.

Estos joins **no generan nuevos hilos por sí mismos**, sino que su efecto se limita a:

- Reunir en un punto, lo diversificado en ramas abiertas originados en conflictos previos (Caso 1)

- Y, definir puntos donde, un único hilo podría ser suficiente para continuar, siempre que solo exista un token simultáneo en dichas plazas.

En particular, el join en **P11** respalda la decisión de utilizar un **único hilo *Exporter*** como mínimo estructural, dado que la etapa de procesamiento previa garantiza que los datos ingresan de manera secuencial a dicha plaza. Como consecuencia, P11 no puede acumular múltiples tokens simultáneamente, lo que hace innecesaria la utilización de más de un hilo para la etapa de exportado, sin afectar el comportamiento ni el rendimiento del sistema.



*** Los hilos Processor se responsabilizan exclusivamente a llevar a cabo la etapa de procesado de datos, no poseen ningun modo inherente a ellos.**

Al analizar la estructura de los invariantes de transición y aplicar los criterios de segmentación por **forks** y **joins**, la red se divide naturalmente en tramos de ejecución que son asignados a hilos específicos. Cada sector de la misma queda bajo la responsabilidad de un grupo de hilos claramente delimitado:

- En la entrada, **Feeder-0** se encarga exclusivamente del segmento inicial **T12**, que es común a todos los invariantes y precede a la etapa de Loader.
- Continuando el flujo de la red, pasamos al Loader, llevado a cabo específicamente por el hilo **Loader-0,** sigue siendo común a todos los invariantes, antecede a la etapa del primer y único conflicto de la red.
- Desde este punto, en la etapa perteneciente a Processor, aparece el único fork de la red (T2/T5/T7). En este tramo la responsabilidad la llevan a cabo los hilos **Processor-0, Processor-1, Processor-2, sin tener ningún modo arraigado a un hilo especifico**:
  - Esta elección de diseño permite modelar la etapa de procesamiento mediante una única clase, teniendo una estructura más simple, la cual nos brinda una menor fragmentación de la implementación, además de mantenerla uniforme (no habría más clases que sustancialmente hagan lo “mismo”). Por lo tanto, el hilo que continúa con el procesamiento es determinado dinámicamente, mientras que el modo de procesamiento ejecutado depende de la transición disparada.

- Asimismo, dado que la estructura de la red garantiza que solo un token puede atravesar la etapa de procesamiento en un instante dado, no existe paralelismo de datos entre los distintos modos. En consecuencia, asignar hilos específicos a cada modo de procesamiento no aportaría mejoras en el rendimiento, ya que el procesamiento es secuencial respecto a los datos.

- Finalmente, el camino hacia la exportación es único y lineal. La transición final (T11) está a cargo del hilo **Exporter-0**, que completa la salida de los datos del sistema.

De este modo, la red queda organizada en **hilos especializados**, cada uno responsable de un segmento definido, lo cual elimina decisiones internas dentro del hilo. Tal como se explica en el algoritmo 4.2 del siguiente [paper].

Conclusión

A lo largo del proyecto fuimos combinando muchas herramientas y conceptos distintos, casi como si cada etapa nos empujara a aprender algo nuevo para poder avanzar. Empezamos programando en Java, primero con lo básico y después metiéndonos de lleno en hilos, sincronización y distintas políticas de ejecución. Para ordenar mejor toda esa arquitectura usamos diagramas UML, tanto de clases como de secuencia, que nos ayudaron a visualizar cómo se relacionaban entre sí los componentes del sistema. También incorporamos Python para automatizar análisis, procesar logs y aplicar expresiones regulares, lo que facilitó validar invariantes de transición y detectar patrones en la ejecución. Paralelamente, tuvimos que profundizar en los conceptos relaciones de Redes de Petri, entender su dinámica, analizar propiedades estructurales, verificar invariantes y aplicar políticas como Round Robin para resolver conflictos. En conjunto, el proyecto terminó siendo un recorrido amplio e intenso, donde cada herramienta y concepto aportó algo distinto para construir una solución funcional, coherente y bien fundamentada.