Programación Concurrente

Trabajo Práctico N°3

GRUPO: We are Teapots

INTEGRANTES:

Tracchia, Belén

Manterola, Lucas

Carrizo, Aixa M.

Fernández, Juan Ignacio

Ferrari, Ivo

ÍNDICE

Introducción

Enunciado

Se debe implementar un simulador de un procesador con dos núcleos. A partir de la red de Petri de la figura 1, la cual representa a un procesador mono núcleo, se deberá extender la misma a una red que modele un procesador con dos núcleos. Además, se debe implementar una Política que resuelva los conflictos que se generan con las transiciones que alimentan los buffers de los núcleos (CPU_Buffer).

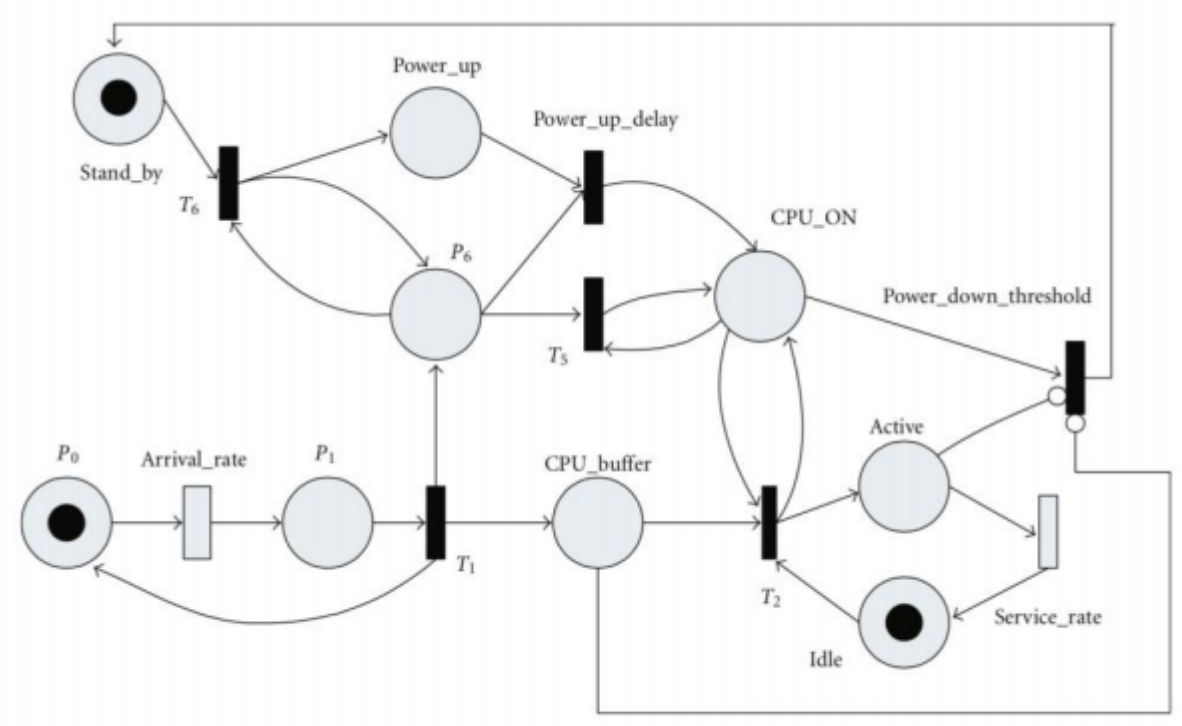


Figura 1

Nota: Las transiciones “Arrival_rate” y “Service_rate” son temporizadas y representan el tiempo de arribo de una tarea y el tiempo de servicio para concluir una tarea.

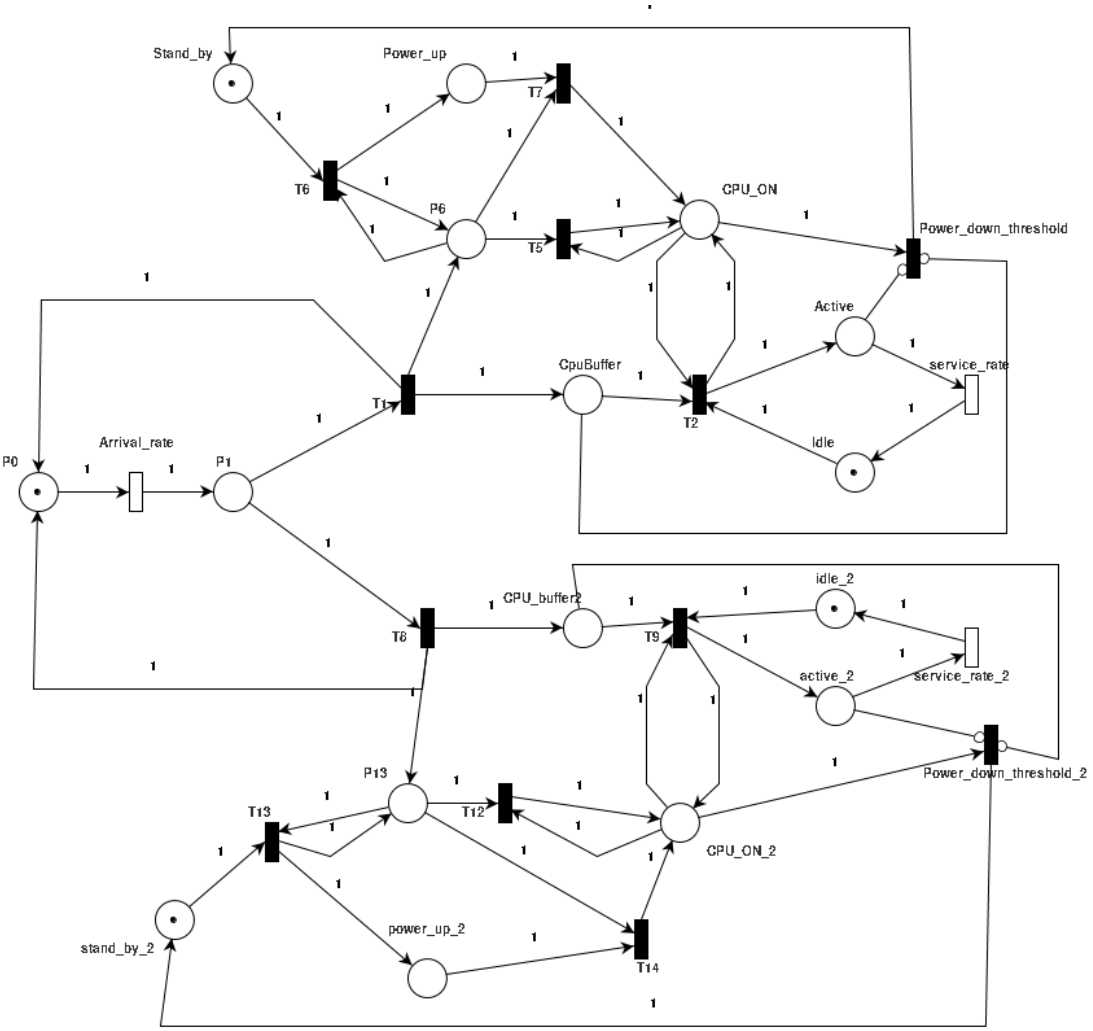
Requerimientos

1) Modelar el sistema de un procesador con dos núcleos, extendiendo la red de Petri de la figura 1. Verificar todas sus propiedades haciendo uso de la herramienta PIPE o Petrinator.
2) Hacer el diagrama de clases que modele el sistema.
3) Implementar un objeto Política que resuelva el conflicto entre las transiciones sensibilizadas que alimentan los buffers de los núcleos, manteniendo la carga de la CPU equitativa.
4) Hacer el diagrama de secuencia que muestre el disparo exitoso de una de las transiciones que alimenta a uno de los buffers de la CPU, mostrando el uso de la política.
5) Indicar la cantidad de hilos necesarios para la ejecución y justificar.
6) Modelar el sistema con objetos en Java.
7) Realizar las siguientes ejecuciones con 1000 tareas completadas (para cada caso):
a) Ambos núcleos con el mismo tiempo de “service_rate”.
b) Un núcleo con el doble de tiempo de “service_rate” que el otro.
c) Un núcleo con el triple de tiempo de “service_rate” que el otro.
8) Verificar los resultados haciendo uso de un archivo de log, verificando el cumplimiento de los invariantes.
9) Debe hacer una clase Main que al correrla, inicie el programa

Red de Petri - Descripción, propiedades e invariantes.

Diagrama de la red de Petri

La red de Petri modelada en Pipe es la siguiente:



Propiedades de la red (Deadlock, acotación, seguridad):

Luego de cargar la red de petri del punto anterior en la herramienta PIPE, se procedió a verificar la información dada en la opción “State Space Analysis” :

Petri net state space analysis results

Bounded	false
Safe	false
Deadlock	false

Gracias a esto se pudieron obtener las siguientes conclusiones:

- La red no es **acotada**, puesto que los buffers pueden contener infinita cantidad de tokens.
- Dado que no es acotada, tampoco es **segura**.
- Tampoco hay **deadlock**, ya que en ninguna marca se presenta la imposibilidad de disparar al menos una transición.

Clasificación

Para comprobar la clasificación se procedió a verificar la información dada en la opción “Classification” :

Petri net classification results

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

Gracias a esto se pudieron obtener las siguientes conclusiones:

- **State Machine - False**: Debido a que hay más de 1 token en la red y plazas con más de 1 entrada y/o salida.
- **Marked Graph - False**: Debido a que hay plazas con más de una entrada/salida (P0, P1, P6, CPU_ON, etc)

- **Free Choice Net - False**: Debido a que hay transiciones que tienen un conflicto y poseen otra entrada de otra plaza. Por ejemplo: T2, T7, etc
- **Extended Free Choice Net - False**: Similar al punto anterior, sumado a que no tienen el mismo conjunto de lugares de entrada.
- **Simple Net - False**: Debido a que hay más de un conflicto que afecta a algunas transiciones (por ejemplo, T5 está en conflicto con T7 por los tokens en P6 y con power_down por el token de CPU_ON).
- **Extended Simple Net - False**: Porque hay transiciones involucradas en más de un conflicto que no tienen conjuntos de entrada iguales o de forma que uno incluya al otro.

Invariantes de plaza y transición de la red

A continuación se muestran los invariantes de marcas y de transición, los cuales se analizarán más adelante.

Petri net invariant analysis results

T-Invariants

Arrival_rate	Power_down_threshold	Power_down_threshold_2	service_rate	service_rate_2	T1	T12	T13	T14	T2	T5	T6	T7	T8	T9
1	1	0	1	0	1	0	0	0	1	0	1	1	0	0
1	0	1	0	1	0	0	1	1	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0	0	0	0	0	1	1
1	0	0	1	0	1	0	0	0	1	1	0	0	0	0

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

P-Invariants

Active	active_2	CPU_buffer	CPU_buffer2	CPU_ON	CPU_ON_2	Idle	idle_2	P0	P1	P13	P6	Power_up	power_up_2	Stand_by	stand_by_2
0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0

The net is not covered by positive P-Invariants, therefore we do not know if it is bounded.

P-Invariant equations

M(active_2) + M(idle_2) = 1
M(CPU_ON) + M(Power_up) + M(Stand_by) = 1
M(CPU_ON_2) + M(power_up_2) + M(stand_by_2) = 1
M(Active) + M(Idle) = 1
M(P0) + M(P1) = 1

Analysis time: 0.001s

Política implementada

La política implementada está orientada a distribuir equitativamente entre ambos CPUs las tareas generadas,colocando la tarea en aquél buffer que se encuentre menos ocupado. En caso de que ambas colas tengan la misma cantidad de tareas en espera de ser atendidas, se asigna la tarea al buffer del CPU2.

Considerando esto, a la hora de decidir qué hilo se despertará luego de un disparo exitoso, se le dá prioridad a los hilos dormidos en las colas de las transiciones que asignan las tareas a su buffer respectivo (T1 y T8), y luego a los que se encuentran en las demás, en el orden en que aparecen indexadas en las matrices generadas en PIPE.

Es decir, si no hay hilos esperando en T1 o T8, se intentará activar a algún hilo siguiendo la siguiente secuencia:

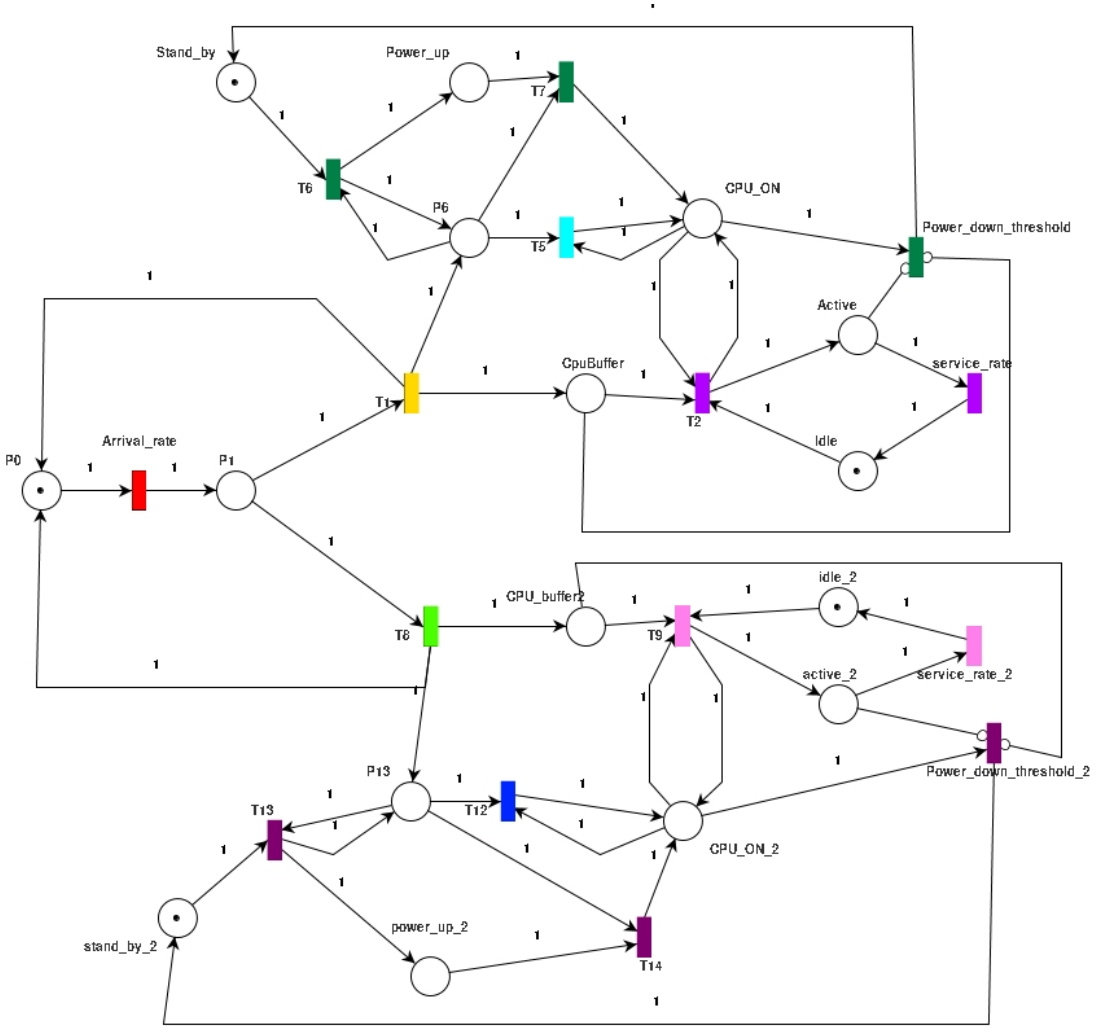
Arrival_rate, power_down1, power_down2, service_rate1, service_rate2, T12, T13, T14, T2, T5, T6, T7, T8, T9.

Descripción y análisis de los hilos utilizados

Para la ejecución necesitaremos 9 hilos, cada uno de los cuales dispara:

1. T5
2. T12
3. T0
4. T1
5. T8
6. T2 y service_rate1
7. T9 y service_rate2
8. T6, T7 y power_down1
9. T13, T14 y power_down2

En la siguiente imagen podemos observar, con distintos colores, que transiciones pertenecen a cada hilo (las transiciones pintadas del mismo color se ejecutan en orden secuencial por el mismo hilo)



La cantidad de hilos a utilizar se deduce a partir de los T - invariantes y P - invariantes.

De estos primeros podemos distinguir 4 secuencias de operaciones sobre la red:

1. Generación de la tarea, asignación de la misma al CPU1, proceso de encendido, atención de la tarea y apagado del CPU1.

2. Generación de la tarea, asignación de la misma al CPU2, proceso de encendido, atención de la tarea y apagado del CPU2.
3. Generación de la tarea, asignación de la misma al CPU2, y atención de la tarea.
4. Generación de la tarea, asignación de la misma al CPU1, y atención de la tarea.

A su vez, consideramos los invariantes de plaza:

Active_2 + Idle_2 = 1

CPU_ON + Power_up + Stand_by = 1

CPU_ON2 + Power_up2 + Stand_by2 = 1

Active + Idle = 1

P0 + P1 = 1

En los cuales también se puede distinguir una separación en las secuencias de encendido y apagado de los CPUs, generación de tareas y consumo de las mismas. Por lo tanto, el análisis de ambos tipos de invariantes nos permite identificar las transiciones que son secuenciales estrictamente respecto a otras.

Es por esto que las operaciones de la red (las cuales son realizadas por los distintos hilos) se han dividido en los procesos que realizan:

- Generación de tareas (hilo 3)
- Asignación de tareas a los buffers (hilos 4 y 5)
- Sustracción de tareas de los buffers y atención de las mismas (hilos 6 y 7)
- Encendido y apagado de los CPUs (hilos 8 y 9)
- Consumo de tokens sobrantes en encendido de CPU/llegada de nuevas tareas (hilos 1 y 2).

Análisis de los logs obtenidos.

Se realizaron los logs de 1000 tareas producidas con un valor de arrival_rate de 25 ms, asumiendo que el tiempo que tardan en llegar las tareas siempre será menor que el tiempo que demoran en ser atendidas (es decir, solo el retardo generado por el servicio de la tarea, sin considerar el tiempo que espera en el buffer). A partir de este supuesto, se produjeron los siguientes resultados, según los valores de service_rate de los CPUs.

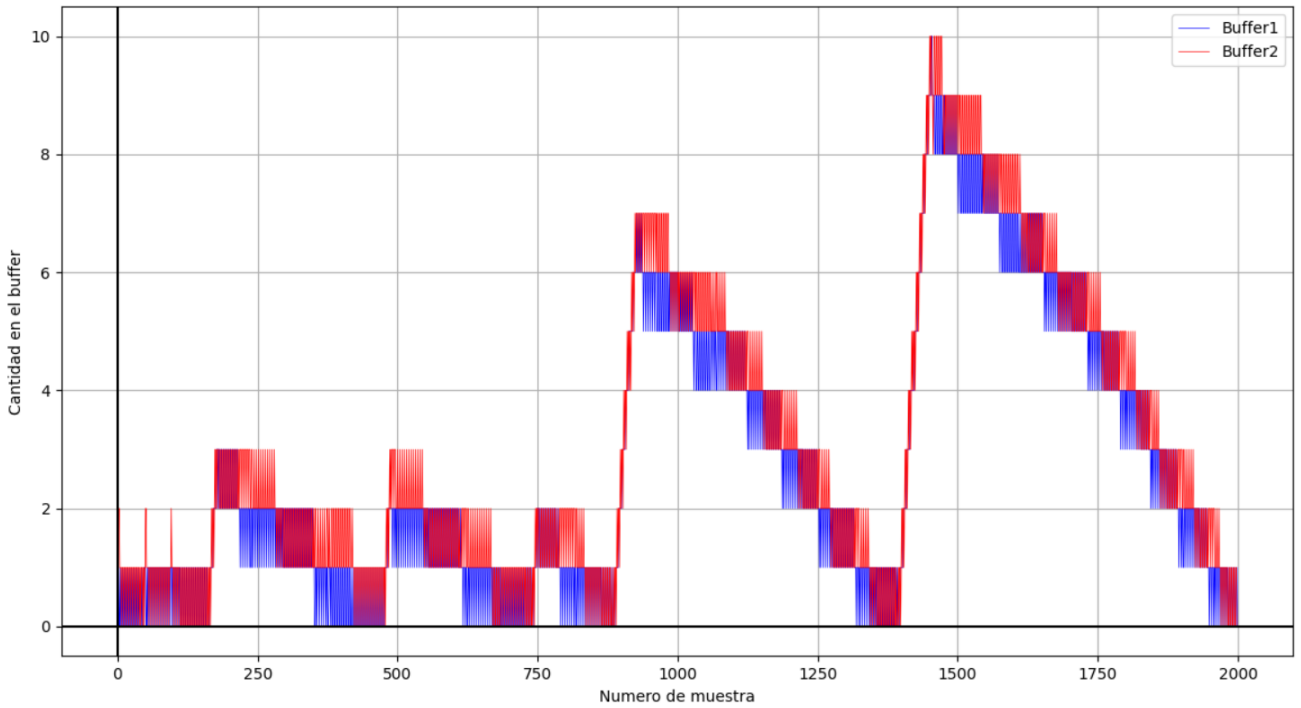
Caso 1

Primeramente, se procedió a realizar la ejecución con un service_rate de 50ms para ambos CPUs.

	CPU1	CPU2
srv_rate (ms)	50	50
Tareas atendidas (promedio de 5 muestras)	475	525

Tiempo de ejecución : ~28 segundos

El gráfico a continuación ilustra la variación de los tamaños de las colas en los CPUs a lo largo de la ejecución:



La cantidad acumulada de tareas atendidas por cada uno de los CPUs muestra una distribución del total relativamente uniforme. La diferencia entre los valores encontrados puede atribuirse al hecho de que, estando los dos CPUs con igual cantidad de tareas encoladas en su buffer correspondiente, la próxima tarea será asignada al buffer del CPU2, según la política implementada.

En cuanto a la carga dinámica de los buffers durante la ejecución, podemos decir que la misma es aproximadamente equitativa, dado que la curva que representa la cantidad de paquetes encolados en el buffer 1 se superpone o se encuentra muy cercana a la del buffer 2. Gracias a esto, podemos decir que el comportamiento real se acerca al deseado, puesto que no se presenta una sobrecarga de ninguna de las colas respecto de la otra.

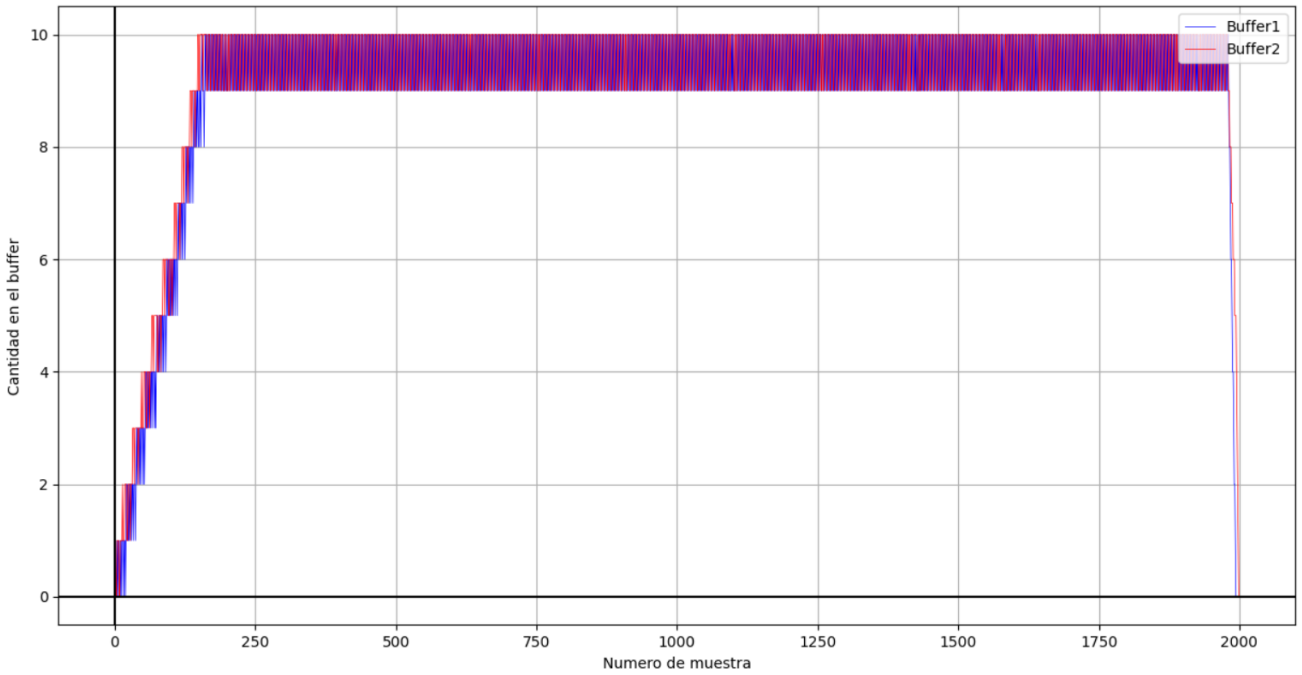
Caso 2

Seguidamente, repetimos el estudio, aumentando la tasa de servicio del CPU2 a 100ms. El comportamiento esperado es un mayor número de tareas

atendidas por el CPU1, y una carga dinámica de los buffers similar al caso anterior (es decir, se espera que la cantidad de los paquetes encolados en los buffers tienda a ser igual durante toda la ejecución).

	CPU1	CPU2
srv_rate (ms)	50	100
Tareas atendidas (promedio de 5 muestras)	660	340

Tiempo de ejecución: ~35 segundos



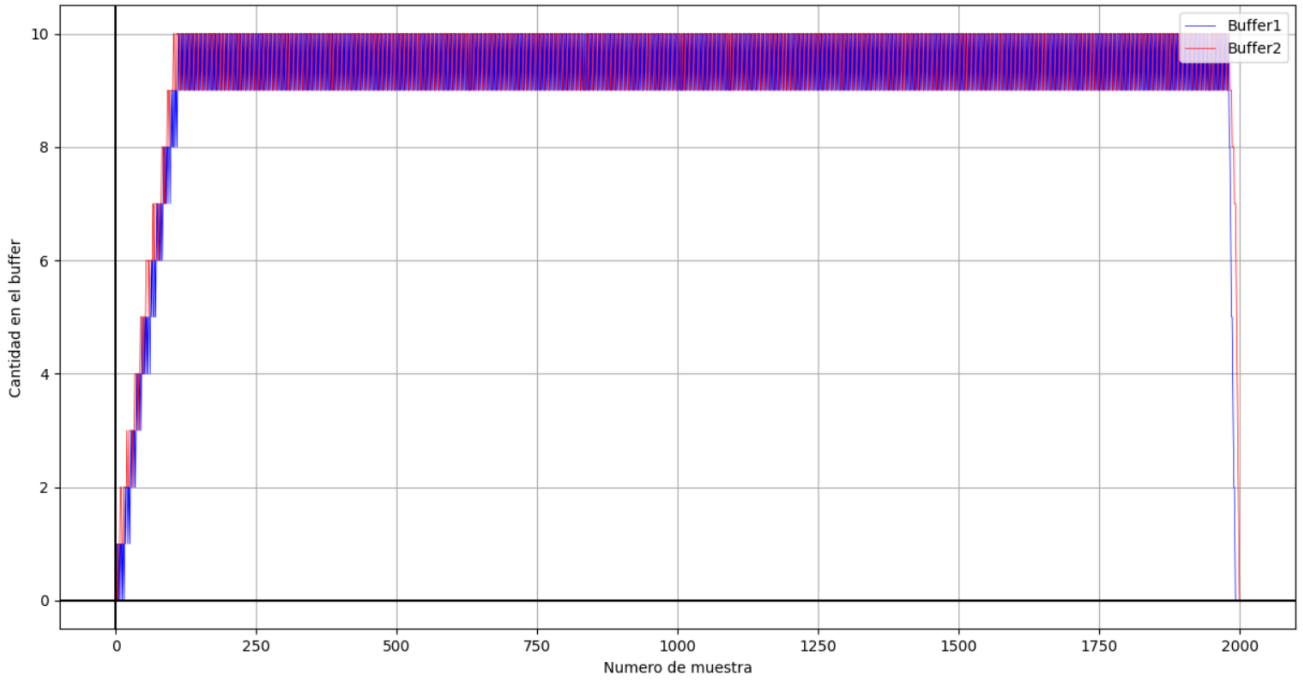
Como esperábamos, la cantidad total de paquetes atendidos por el CPU1 fue mayor. Congruentemente con este hecho, en el gráfico podemos observar a simple vista que el color azul (representando la acumulación de tareas en el buffer 1) predomina, lo que es causado por las variaciones (entrada/salida) de tareas en la cola del CPU1, dado que éste atiende las tareas y se desocupa con mayor velocidad que el buffer de CPU2.

Caso 3

Finalmente, realizaremos una última prueba, esta vez, con un service_rate2 de 150 ms.

	CPU1	CPU2
srv_rate (ms)	50	150
Tareas atendidas (promedio de 5 muestras)	739	261

Tiempo de ejecución: ~40 segundos



En este caso, las variaciones de carga del buffer 1 son más evidentes, lo que coincide con la mayor cantidad de tareas atendidas que se obtuvo para el CPU1.

Conclusiones del análisis

Con estos 3 casos estudiados, podemos afirmar que:

- Existe una relación proporcional entre la diferencia de tareas atendidas totales atendidas por cada CPU y la diferencia entre sus tasas de servicio, de forma que:

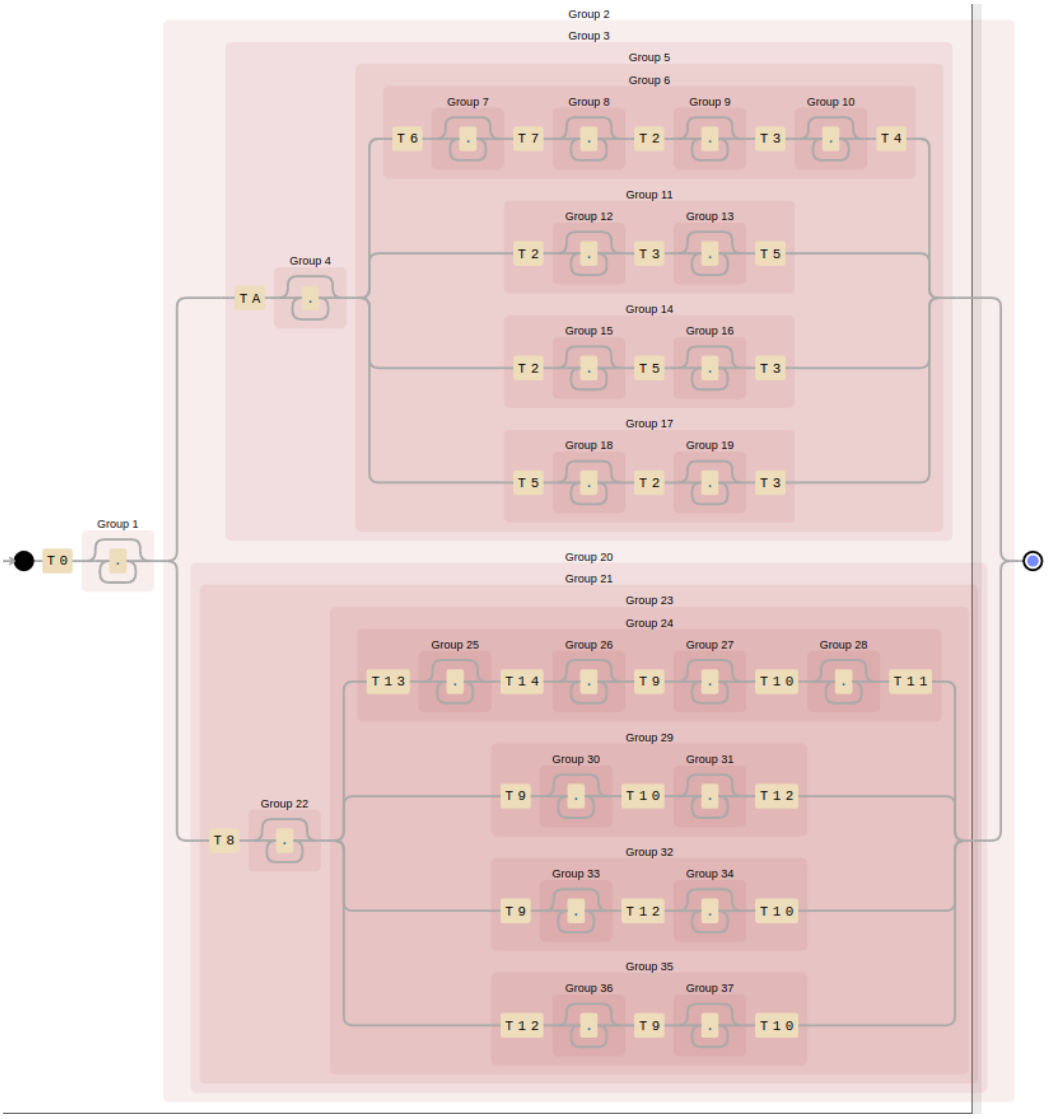
$$\frac{serviceRate2}{serviceRate1} \alpha \frac{Total\ De\ Tareas\ CPU1}{Total\ De\ Tareas\ CPU2}$$

- Se cumple el requerimiento de balanceo de carga en los buffers.
- Mientras más aumenta la carga sobre uno de los CPUs, el tiempo total necesario para atender una determinada cantidad de tareas también se hace mayor.

Verificación de invariantes de transición y de plaza.

Invariantes de transición

Para verificar el cumplimiento de los invariantes de transición, se empleó un script en Python en el cual, mediante una expresión regular, se eliminan los invariantes encontrados, de forma que, si la salida al final es un string vacío, se puede afirmar el cumplimiento de los mismos durante toda la ejecución.
La expresión regular utilizada es la siguiente:
T0(.*?)((TA(.*?)((T6(.*?)T7(.*?)T2(.*?)T3(.*?)T4)|(T2(.*?)T3(.*?)T5)|(T2(.*?)T5(.*?)T3)|(T5(.*?)T2(.*?)T3)))|((T8(.*?)((T13(.*?)T14(.*?)T9(.*?)T10(.*?)T11)|(T9(.*?)T10(.*?)T12)|(T9(.*?)T12(.*?)T10)|(T12(.*?)T9(.*?)T10)))))



En la figura anterior se muestra una representación gráfica de la expresión regular y cómo se buscan las combinaciones asociadas a ésta (y que representan a los 4 invariantes de transición). Dichas secuencias se encuentran en el orden en el que deberían ocurrir, y son las siguientes:

- T0TAT6T7T2T3T4
- T0T8T13T14T9T10T11
- T0T8T12T9T10
- T0TAT5T2T3

Cabe aclarar que:

1. La expresión “(.*?)” hace referencia a cualquier conjunto de transiciones entre la transición anterior y posterior, permitiendo encontrar invariantes entremezclados.
2. Se utiliza TA en vez de T1. Esto se debe a que la expresión regular podría confundir la transición 11 (T11) con la transición 1 más un 1 (T1 + 1), dando un resultado erróneo.

La imagen a continuación muestra la aplicación del script a la secuencia de transiciones obtenida de una de las pruebas realizadas anteriormente.

```
~/facu/concurrente/TP3Concurrente2021v2 main* › python test.py
Test de invariantes de transicion:
El test ha finalizado exitosamente. Enhorabuena!
```

Si la secuencia de transiciones no cumple con los T-Invariantes establecidos, veremos una salida similar a la siguiente:

```
~/facu/concurrente/TP3Concurrente2021v2 main* › python test.py
Test de invariantes de transicion:
Error de T-Invariantes
T13T14T8T0T9T10
```

En la misma se muestra el mensaje que indica el fallo, junto a la secuencia de transiciones que no corresponde a ningún invariante. En este caso, la misma se obtuvo borrando T11 del final del archivo obtenido originalmente.
Cabe aclarar que el error detectado por el script no siempre es a causa de un T-invariante incorrecto, sino que puede darse el caso en el que las transiciones disparadas correspondan con uno de los T-invariantes, pero se encuentren en un orden diferente a alguno de los contemplados por la expresión regular.

Invariantes de plaza

Para controlar los invariantes de plaza, luego de cada disparo que se realiza, antes de que el hilo abandone el monitor, el mismo ejecuta una rutina para verificar el cumplimiento de los mismos.

```
/*
INVARIANTES DE PLAZA
m1 + m7 = 1
m4 + m12 + m14 = 1
m5 + m13 + m15 = 1
m0 + m6 = 1
m8 + m9 = 1
*/

private boolean verifyMInvariants () throws Exception {
    int mark[] = pn.getMarkVector ();

    if (((mark[1] + mark[7]) == 1) && ((mark[4] + mark[12] + mark[14]) == 1) &&
            ((mark[5] + mark[13] + mark[15]) == 1) && ((mark[0] + mark[6]) == 1) && ((mark[8] + mark[9])
        return true;
    else {
        throw new Exception ("Fallo en invariantes de plaza");
    }
}
```

En el caso de que resulte exitoso, la ejecución continuará normalmente. Si se encuentra algún error, se detendrá la ejecución y el mensaje obtenido será similar al siguiente:

```
java.lang.Exception: Fallo en invariantes de plaza
        at MonitorV2.verifyMInvariants(MonitorV2.java:67)
        at MonitorV2.shoot(MonitorV2.java:154)
        at CPUProcess.run(CPUProcess.java:23)
```

Conclusiones

Durante la realización del presente trabajo, nos encontramos con diferentes inconvenientes relacionados tanto al diseño del sistema como a la implementación del mismo. A continuación, describiremos los más importantes y que, con un abordaje correcto, nos permitieron lograr un resultado satisfactorio.

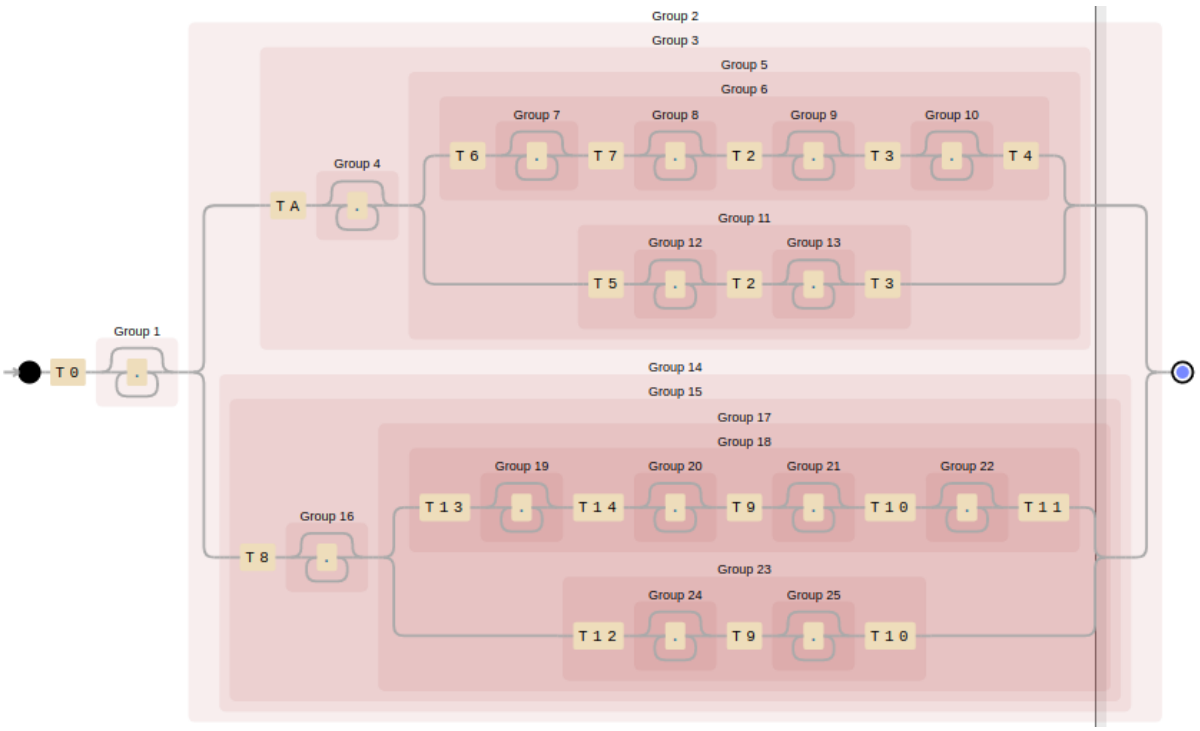
Mala interpretación de la consigna

Primeramente, la solución encontrada para la resolución de la consigna se encontraba fuertemente ligada a la red de Petri, de forma que, si se deseaba realizar algún cambio sobre la misma, la repercusión en el código fuente del monitor era significativa. Por ello, se replanteó la totalidad del diseño para el sistema, de forma que la lógica de funcionamiento se encuentra ahora en la red de Petri y en la política.

Inconveniente con expresión regular

Asimismo, esta nueva implementación acarreó problemas a la hora de realizar el análisis de invariantes de transición. La dificultad para predecir con exactitud el momento en el que las transiciones T5 y T12 se disparan provoca que la expresión regular no detecte algunos invariantes, aunque los mismos sean correctos. Como la cadena resultante en caso de falla es pequeña en relación al total, se optó por revisar estos restos de forma manual.

Para confirmar que el problema presente en la verificación de los invariantes de transición se debe al orden en que aparecen T5 y T12 y no a otro factor, se procedió a agregar una lógica de prioridad para que T5 y T12 fuesen, después de T1 y T8, las transiciones con mayor privilegio de ejecución. En otras palabras, se redujo la expresión regular a:



y se agregó el código encuadrado en rojo en la clase Política:

```
public int signalPolitic (boolean[] boolQuesWait) {
    aux = pn.getSensitized ();
    if (boolQuesWait[addBuffer[0]] && boolQuesWait[addBuffer[1]] && aux[addBuffer[0]] == 1) {
        markVector = pn.getMarkVector ();

        if (markVector[listBuff[0]] < markVector[listBuff[1]]) {
            return addBuffer[0];
        } else if (markVector[listBuff[0]] >= markVector[listBuff[1]]) {
            return addBuffer[1];
        }

    }
    if(aux[6] == 1 && boolQuesWait[6])
        return 6;
    if(aux[10]== 1 && boolQuesWait[10])
        return 10;

    for (int i = 0; i < 15; i++) {
        if (aux[i] == 1 && boolQuesWait[i] && i != 5 && i != 13) {
            return i;
        }
    }
    return -1;
}
```

Como se esperaba, el resultado obtenido luego de realizar estas modificaciones fue que en todos los casos observados, la detección de invariantes fue correcta. Aún así, estos cambios no se aplicaron en el sistema final dado que no encontramos correcta la implementación arbitraria de prioridades para T5 y T12 solo para asegurar la detección de invariantes. Esto

que exponemos puede verificarse descomentando las líneas explicitadas en la clase Política y modificando ligeramente la expresión regular de test.py (ver comentario en test.py).

Política de CPUs

Por último, aunque la política no garantiza una distribución absolutamente equitativa de tareas acumuladas entre los CPUs, encontramos que la solución propuesta se ajusta a lo requerido, puesto que en ningún momento se encuentra un CPU sobrecargado estando el otro ocioso, de forma que el proceso de apagado de los mismos solo inicia cuando ya se han consumido todas las tareas producidas.