0.1 Sistema: n servidores en serie con retroceso

Este proyecto se enfoca en la creación de una simulación de eventos discretos con el fin de analizar y comprender diversos fenómenos. Nuestro objetivo es aplicar los principios de esta simulación para modelar y experimentar con dichos fenómenos, con el propósito de obtener resultados que guíen nuestras decisiones de manera informada.

Los clientes llegan a un sistema que tiene n servidores, y las llegadas distribuyen M. Cada cliente que llega debe ser atendido primero por el servidor 1 y, al completar el servicio en el servidor 1, el cliente pasa al servidor 2.

Cuando un cliente llega, entra en servicio con el servidor 1 si ese servidor está libre, o se une a la cola del servidor 1 en caso contrario. De manera similar, cuando el cliente completa el servicio en el servidor 1, entra en servicio con el servidor 2 si ese servidor está libre, o se une a su cola y así sucesivamente. Después de ser atendido en el servidor n, el cliente abandona el sistema.

El scliente en el servidor i con una probabilidad p puede saltar a la cola del servidor j.

Los tiempos de servicio en el servidor i tienen la distribución Gi

0.1.1 Variables

Tiempo t: tiempo general
t_A: tiempo de arribo del próximo cliente
t_Di: tiempo de salida del servidor i

Contadoras n_A: cantidad de arribos
n_D: cantidad de salidas
Q_i: arribos a la cola del servidor i
A_i: arribos al servidor i
D_j: salidas del servidor i

Estado $Queue_i$: cola de espera del servidor i

0.1.2 Implementación

Se define un enum llamado EventType, el cual representa los distintos tipos de eventos que pueden ocurrir en el sistema simulado. Cada tipo de evento tiene asignado un valor entero único.

ASIGNED: Este evento se refiere a la asignación de un cliente a la cola de un servidor.

ARRIVAL: Representa la llegada de un cliente a un servidor.

FINISH: Indica la finalización o conclusión del servicio al cliente.

[1]:
```
from enum import auto, Enum


class EventType(Enum):
    ASIGNED = auto()
    ARRIVAL = auto()
    FINISH = auto()
```

```
    def __lt__(self, other):
        return self.value < other.value
```

Para simular servidores informáticos, hemos empleado una distribución exponencial con distintos valores de lambda. Estos valores de lambda representan diversas velocidades de procesamiento, que varían según la calidad y la generación de los servidores. Al utilizar distribuciones exponenciales, modelamos el tiempo que tarda cada servidor en completar una tarea, teniendo en cuenta la tasa promedio de llegada de solicitudes.

La selección de distribuciones y valores específicos de lambda o parámetros de distribución se basa en la necesidad de modelar con precisión las características y variaciones de tiempo en cada contexto simulado, lo que nos permite obtener resultados realistas y significativos para la toma de decisiones y la optimización de los sistemas.

0.1.3 Servidores informáticos

[2]:
```
from queue import PriorityQueue as pq
import random
from tabulate import tabulate
import numpy as np

def servers_simulation(time, n, lambda_arrival_time, mu_wait_time, p,␣
↪finish=True):
    """
    Este método simula un sistema de servidores para atender solicitudes de␣
↪clientes.

    Args:
        time (int): Duración máxima de la simulación
        n (int): El número de servidores en el sistema.
        lambda_arrival_time (float): La tasa promedio de llegada de solicitudes␣
↪al sistema.
        mu_wait_time list(float): La tasa promedio de tiempo de espera en el␣
↪sistema.
        p (float): La probabilidad de que una solicitud sea asignada a un␣
↪servidor adyacente.

    Returns:
        t: Tiempo del último evento procesado.
        n: Número de servidores del sistema.
        n_a (int): Número de arribos al sistema.
        n_d (int): Número de clientes atendidos en el sistema.
        q (list(dict)): Por cada par {server, client} tenemos los tiempos de␣
↪llegada a la cola del servidor de dicho cliente.
        a (list(dict)): Por cada par {server, client} tenemos los tiempos de␣
↪llegada al servidor de dicho cliente.
```

```
        d (list(dict)): Por cada par {server, client} tenemos los tiempos de␣
↪salida del servidor de dicho cliente.
    """

    #Inicialización de variables
    t = 0
    events = pq()
    # Events: cola con prioridad, que se organizará en base al tiempo, de modo␣
↪que siempre esté arriba el evento más cercano a ocurrir.
    n_a = 0
    n_d = 0
    a = [{} for _ in range(n)]
    q = [{} for _ in range(n)]
    d = [{} for _ in range(n)]
    queue = [[] for _ in range(n)]

    # LLegada del primer cliente al sistema
    events.put((t + random.expovariate(lambda_arrival_time), (EventType.ASIGNED,␣
↪0, 0)))

    while((finish or t < time ) and not events.empty()):
        #Analizamos el próximo evento, el evento con menor t, y por tanto el␣
↪primero en la cola de prioridad.
        (t, (e, c, i)) = events.get()

        if e == EventType.ASIGNED:
            # Evento: ASIGEND (llegó un cliente nuevo al servidor {i})
            index = c if i !=0 else n_a

            q_t = q[i].get(index, list())
            q_t.append(t)
            q[i][index] = q_t

            if len(queue[i]) == 0:
                #Si la cola está vacía el servidor lo atiende inmediatamente
                #Por lo que se pone en la cola de eventos en este mismo tiempo␣
↪con el estado ARRIVAL
                events.put((t, (EventType.ARRIVAL, index, i)))

            queue[i].append(index)

            if i == 0:
                # Si el servidor es el 0, significa que el cliente recién arribó␣
↪al sistema
                n_a += 1
```

```
                # Entonces generamos el tiempo de llegada del próximo cliente
                t_next_a = random.expovariate(lambda_arrival_time)
                if t + t_next_a < time:
                    events.put((t + t_next_a, (EventType.ASIGNED, 0, 0)))

        elif e == EventType.ARRIVAL:

            # Asignamos el tiempo de llegada al servidor {i} del cliente numero␣
↪{c}
            a_t = a[i].get(c, list())
            a_t.append(t)
            a[i][c] = a_t
            # Generamos el tiempo que demorará en ser atendido
            duration = random.expovariate(mu_wait_time[i])
            # Añadimos el evento finish del client en el servidor {i} teniendo␣
↪en cuenta la duración generada.
            events.put((t + duration, (EventType.FINISH, c, i)))


        elif e == EventType.FINISH:
            if len(queue[i]) > 0:
                # En la punta de la cola siempre está el número del cliente que␣
↪se está atendiendo en ese momento.
                queue[i].pop(0)

            if len(queue[i]) > 0:
                # Si después de sacar de la cola el que se terminó de atender en␣
↪el t actual quedan clientes,
                # entonces creamos el evento arrival para el siguiente en la cola
                events.put((t, (EventType.ARRIVAL, queue[i][0], i)))

            # Asignamos el tiempo de salida del servidor {i} del cliente número␣
↪{c}
            d_t = d[i].get(c, list())
            d_t.append(t)
            d[i][c] = d_t

            if i == n-1:
                # Si el cliente estaba en el último servidor aumentamos el␣
↪número de clientes que han salido del sistema
                n_d += 1
                continue

            # Asignamos el cliente que recién termina en el servidor {i} al␣
↪siguiente servidor.
            # Este puede ser {i+1} o un servidor anterior con probabilidad p.
```

```
            next_server = i+1 if i == 0 or random.uniform(0,1) > p[i] else␣
↪random.choice([j for j in range(i)])
            events.put((t, (EventType.ASIGNED, c, next_server)))

    return t, n, n_a, n_d, q, a, d
```

0.1.4 Hallazgos y Experimentos

Primero preparamos algunos métodos que nos harán más fácil experimentar con la simulación desarollada

[3]:
```
class Metrics:
    def __init__(self, time, n_d, wait_time, use_time, system_time) -> None:
        self.time = time
        self.n_d = n_d
        self.wait_time = wait_time
        self.use_time = use_time
        self.system_time = system_time
```

[4]:
```
def metrics_by_simulation(time, n, lambda_arrival_time, mu_wait_time, p, finish):

    # Ejecutamos la simulación del servidor para obtener las métricas relevantes.
    t, n, n_a, n_d, q, a, d = servers_simulation(time, n, lambda_arrival_time,␣
↪mu_wait_time, p, finish)

    wait_time = []
    use_time = []
    system_time = [0 for _ in range(n_a)]
    on_queue = [0 for _ in range(n)]
    processing = [0 for _ in range(n)]

    # Iteramos sobre los servidores para calcular las métricas.
    for s in range(n):

        wait_time_per_client = []
        use_time_per_client = []

        for c in range(n_a):

            q_t = q[s].get(c, list())
            a_t = a[s].get(c, list())
            d_t = d[s].get(c, list())

            if len(q_t) == 0:
                continue
```

```
            if len(a_t) == 0:
                wait_time_per_client.append(t - q_t[0])
                on_queue[s] += 1
                continue

            if len(d_t) == 0:
                w_t = sum(a - q for q, a in zip(q_t, a_t)) / len(a_t)
                wait_time_per_client.append(w_t)

                processing[s] + 1
                continue

            # Calculamos los tiempos de espera y uso promedio por cliente.
            w_t = sum(a - q for q, a in zip(q_t, a_t)) / len(a_t)
            u_t = sum(d - a for a, d in zip(a_t, d_t)) / len(a_t)
            wait_time_per_client.append(w_t)
            use_time_per_client.append(u_t)

            # Actualizamos el tiempo del sistema si es el primer o último␣
↪servidor.
            if s == 0 and c < n_d:
                system_time[c] -= q_t[0]
            if s == n-1 and c < n_d:
                system_time[c] += d_t[-1]

        wait_time.append(np.mean(wait_time_per_client))
        use_time.append(np.mean(use_time_per_client))

    return Metrics(t, n_d, wait_time, use_time, system_time)


def run_simulation(time, n, lambda_arrival_time, mu_wait_time, p, finish):

    # Definimos los parámetros para el control de convergencia.
    min_iterations = 1000
    max_iterations = 5000
    convergence_threshold = 0.001
    iterations = 0

    system_time = []
    wait_time = []
    use_time = []

    while True:
        metrics = metrics_by_simulation(time, n, lambda_arrival_time,␣
↪mu_wait_time, p, finish)
```

```
        system_time.append(np.mean(metrics.system_time))
        wait_time.append(np.mean(metrics.wait_time))
        use_time.append(np.mean(metrics.use_time))

        iterations += 1

        if iterations >= max_iterations:
            break

        # Verificamos la convergencia después de alcanzar el número mínimo de␣
↪iteraciones.
        if iterations >= min_iterations:
            standard_deviation = np.std(system_time)

            if standard_deviation/len(system_time) < convergence_threshold:
                break


    return {
        "Wait Time": (np.var(wait_time), np.mean(wait_time)),
        "Use Time": (np.var(use_time), np.mean(use_time)),
        "System Time": (np.var(system_time), np.mean(system_time))
    }
```

0.1.5 Convergencia

La verificación de la convergencia en las simulaciones es un aspecto crucial para garantizar la fiabilidad y precisión de los resultados obtenidos. En este proyecto, se implementó un método de convergencia basado en la observación de métricas clave a lo largo de múltiples iteraciones de simulación.

Parámetros de Convergencia Para controlar la convergencia de las simulaciones, se definieron los siguientes parámetros:

min_iterations: Número mínimo de iteraciones antes de verificar la convergencia. max_iterations: Número máximo de iteraciones permitidas. convergence_threshold: Umbral de convergencia, que determina cuándo se considera que la simulación ha convergido.

El proceso de convergencia se llevó a cabo mediante un bucle while que ejecuta iteraciones de la simulación hasta que se cumplan las condiciones de convergencia. Durante cada iteración, se recopilaron métricas relevantes, incluyendo el tiempo en el sistema (system_time), el tiempo de espera (wait_time), y el tiempo de uso (use_time).

La convergencia se verifica después de alcanzar el número mínimo de iteraciones (min_iterations). Se calcula la desviación estándar de los tiempos del sistema y se compara con el umbral de convergencia (convergence_threshold). Si la desviación estándar dividida por la cantidad de simulaciones es menos que el umbral se considera que la simulación ha convergido y el bucle se detiene. También se considera que se deben tener las simulaciones después de un número máximo de iteraciones.

Una vez que se alcanza la convergencia, se recopilan las métricas finales, incluyendo la varianza y la media de los tiempos de espera, uso del sistema y tiempo del sistema. Estas métricas proporcionan una visión completa del desempeño del sistema bajo las condiciones simuladas.

0.1.6 Experimento 1

Un único servidor con un procesamiento promedio de 256 solicitudes procesadas y 150 recibidas por segundo, sin retroceso y las simulaciones tendrán una duración de 5s = 5000ms.

[5]:
```
results = run_simulation(5000, 1, 1/7, [1/4], [0], True)

rows = [[key, results[key][0], results[key][1]] for key in results.keys()]

print(tabulate(rows, headers=["Metric", "Variance", "Expected value"],␣
 ↪tablefmt="grid"))
```

```
+-------------+------------+------------------+
| Metric      |   Variance |   Expected value |
+=============+============+==================+
| Wait Time   |   1.5084   |          5.31645 |
+-------------+------------+------------------+
| Use Time    |   0.021088 |          3.99982 |
+-------------+------------+------------------+
| System Time |   1.72313  |          9.31627 |
+-------------+------------+------------------+
```

0.1.7 Experimento 2

Un único servidor con un procesamiento promedio de 512 solicitudes procesadas y 150 recibidas por segundo, sin retroceso y las simulaciones tendrán una duración de 5s = 5000ms.

[6]:
```
results = run_simulation(5000, 1, 1/7, [1/2], [0], True)

rows = [[key, results[key][0], results[key][1]] for key in results.keys()]

print(tabulate(rows, headers=["Metric", "Variance", "Expected value"],␣
 ↪tablefmt="grid"))
```

```
+-------------+------------+------------------+
| Metric      |   Variance |   Expected value |
+=============+============+==================+
| Wait Time   | 0.0184834  |          0.79052 |
+-------------+------------+------------------+
| Use Time    | 0.00538621 |          1.99254 |
+-------------+------------+------------------+
| System Time | 0.0344657  |          2.78306 |
+-------------+------------+------------------+
```

0.1.8 Experimento 3

Un único servidor con un procesamiento promedio de 1024 solicitudes procesadas y 150 recibidas por segundo, sin retroceso y las simulaciones tendrán una duración de 5s = 5000ms.

[7]:
```
results = run_simulation(5000, 1, 1/7, [1], [0], True)

rows = [[key, results[key][0], results[key][1]] for key in results.keys()]

print(tabulate(rows, headers=["Metric", "Variance", "Expected value"],␣
 ↪tablefmt="grid"))
```

```
+-------------+------------+------------------+
| Metric      |   Variance |   Expected value |
+=============+============+==================+
| Wait Time   | 0.00104274 |         0.167856 |
+-------------+------------+------------------+
| Use Time    | 0.0012394  |         1.00035  |
+-------------+------------+------------------+
| System Time | 0.00325681 |         1.1682   |
+-------------+------------+------------------+
```

0.1.9 Experimento 4

Tres servidores con un procesamiento promedio de 256, 512 y 1024 solicitudes procesadas y 150 recibidas por segundo, sin retroceso y las simulaciones tendrán una duración de 5s = 5000ms.

[8]:
```
results = run_simulation(5000, 3, 1/7, [1/4, 1/2, 1], [0, 0, 0], True)

rows = [[key, results[key][0], results[key][1]] for key in results.keys()]

print(tabulate(rows, headers=["Metric", "Variance", "Expected value"],␣
 ↪tablefmt="grid"))
```

```
+-------------+------------+------------------+
| Metric      |   Variance |   Expected value |
+=============+============+==================+
| Wait Time   | 0.14684    |          2.07912 |
+-------------+------------+------------------+
| Use Time    | 0.00339538 |          2.33611 |
+-------------+------------+------------------+
| System Time | 1.5352     |         13.2457  |
+-------------+------------+------------------+
```

0.1.10 Experimento 5

Tres servidores con un procesamiento promedio de 1024, 512 y 256 solicitudes procesadas y 150 recibidas por segundo, sin retroceso y las simulaciones tendrán una duración de 5s = 5000ms. En este caso pusimos primero los servidores más rápidos para verificar si había alguna diferencia.

[9]:
```
results = run_simulation(5000, 3, 1/7, [1, 1/2, 1/4], [0, 0, 0], True)

rows = [[key, results[key][0], results[key][1]] for key in results.keys()]

print(tabulate(rows, headers=["Metric", "Variance", "Expected value"],␣
 ↪tablefmt="grid"))
```

```
+-------------+------------+------------------+
| Metric      |   Variance |   Expected value |
+=============+============+==================+
| Wait Time   | 0.161482   |          2.09725 |
+-------------+------------+------------------+
| Use Time    | 0.00310491 |          2.33576 |
+-------------+------------+------------------+
| System Time | 1.70726    |         13.299   |
+-------------+------------+------------------+
```

0.1.11 Experimento 6

Tres servidores con un procesamiento promedio de 1024, 512 y 256 solicitudes procesadas y 150 recibidas por segundo, con retroceso y las simulaciones tendrán una duración de una 5s = 5000ms.

[10]:
```
results = run_simulation(5000, 3, 1/7, [1/4, 1/2, 1], [0, 0.1, 0.1], True)

rows = [[key, results[key][0], results[key][1]] for key in results.keys()]

print(tabulate(rows, headers=["Metric", "Variance", "Expected value"],␣
 ↪tablefmt="grid"))
```

```
+-------------+-------------+------------------+
| Metric      |    Variance |   Expected value |
+=============+=============+==================+
| Wait Time   | 9.50275e+06 |      28736.7     |
+-------------+-------------+------------------+
| Use Time    | 4.49179e-05 |          2.33322 |
+-------------+-------------+------------------+
| System Time | 5.60385e+07 |      69266.6     |
+-------------+-------------+------------------+
```

0.1.12 Interpretación de los resultados

A medida que se incrementa la capacidad de procesamiento del servidor, se observa una mejora significativa en los tiempos de espera y de uso, lo que indica una mayor eficiencia en el manejo de las solicitudes.

La introducción de múltiples servidores con diferentes capacidades de procesamiento aumenta la complejidad del sistema y resulta en una variabilidad más alta en los tiempos del sistema.

Los resultados sugieren que la eficiencia y el rendimiento del sistema de servidores en serie están influenciados principalmente por la capacidad de procesamiento de los servidores individuales y la

complejidad introducida por la presencia de múltiples servidores.

Los experimentos que involucran múltiples servidores muestran que el orden en que se disponen los servidores no tiene un impacto significativo en los tiempos de espera, de uso y del sistema. Esto sugiere que, en configuraciones con múltiples servidores, el rendimiento del sistema no está influenciado por el orden en que se distribuyen las capacidades de procesamiento de los servidores.

Se observa que aumentar ligeramente la probabilidad de reciclado resulta en un aumento significativo en el tiempo del sistema y en la duración total de la simulación. Esto indica que incluso pequeñas variaciones en la probabilidad de reciclado pueden tener un efecto considerable en el rendimiento del sistema y en el tiempo necesario para completar las simulaciones.

0.1.13 Modelo Matemático

Supuestos

- Se asume que las colas son potencialmente infinitas, es decir, pueden almacenar un número ilimitado de clientes en espera.
- No se considera prioridad en el servicio; el principio de primero en entrar, primero en salir (FIFO) se sigue en todas las colas.
- Se supone que los clientes llegan al sistema de uno en uno, no se permiten llegadas simultáneas de varios clientes.
- No se tienen en cuenta averías ni fallos en los servidores, se asume un funcionamiento continuo y sin interrupciones.
- No se contempla el abandono de la cola por parte de los clientes, todos los clientes que ingresan a la cola eventualmente serán atendidos.
- La simulación se detiene cuando todos los clientes han salido de la cola, pero a partir de un tiempo determinado ya no se permiten nuevas entradas de clientes al sistema.

Descripción Inicialmente, nos enfocaremos en el caso donde existe un único servidor y por tanto no hay retroceso.

En el problema de los servidores informáticos usaremos la distribución exponencial para las llegadas y salidas del sistema.

Consideramos que los servidores pueden procesar en promedio 256 request por segundo, lo que implica un tiempo promedio de respuesta de 1 request cada 4 ms aproximadamente. Por tanto, usaremos la distribución exponencial con $\mu = \frac{1}{4}$ para simular las salidas. Siguiendo esta misma lógica, si consideramos que llegan alrededor de 150 solicitudes por segundo, entonces para las llegadas usaremos la distribución exponencial con $\lambda = \frac{1}{7}$.

Entonces la tasa de llegada sería: $a(t) = \lambda e^{-\lambda t} = 1/7e^{-1/7t}$ y la tasa de salida $d(t) = \mu e^{-\mu t} = 1/4e^{-1/4t}$

Dada la teoria explicada en [1], podemos considerar el número promedio de clientes en la cola como:

$L = (1-\rho)\rho \sum_{n=0}^{\infty} n\rho^{n-1} \quad donde : \sum_{n=0}^{\infty} n\rho^{n-1} = \frac{1}{(1-\rho)^2} \quad por, tanto : L = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu-\lambda} = \frac{1/7}{1/4-1/7} = 1.33$

De este modo podemos calcular también el tiempo medio de espera de los clientes en cola como:

$W = \frac{L}{\lambda} = \frac{1}{\mu-\lambda} = 9.33$

Lo que se aproxima a los resultados obtenidos en el experimento # 1.

Metric	Variance	Expected value
System Time	1.58985	9.3543

Ahora consideremos el caso de n servidores, con capacidad infinita y sin retroceso.

Partiendo de que la capacidad es infinita podemos analizar cada servidor por separado y apoyarnos en el análisis inicial.

La distribución de la salida de los clientes de un sistema M/M/1/∞ tiene una distribución idéntica a la de la entrada.

Entonces la probabilidad de que en un instante dado haya n_1 clientes en el servidor 1, n_2 en el 2 y n_k en el k es $P_{n1,n2,..,nk} = P_{n1}P_{n2}..P_{nk}$

Entonces apoyados en lo descrito anteriormente, si tenemos 3 servidores con:

$\mu_1 = 1/4$, $\mu_2 = 1/2$ *y* $\mu_3 = 1$ $\lambda = 1/7$

Se pueden calcular los tiempos de espera promedio de los clientes en cada cola como: $W_1 = 9.33, W_2 = 2.8 y W_3 = 1.16$, por lo que el tiempo de espera total será de aproximadamente $W = 13.29$

Lo que se aproxima a los resultados obtenidos

Experimento # 4

Metric	Variance	Expected value
System Time	1.68916	13.2794

Experimento # 5

Metric	Variance	Expected value
System Time	1.88207	13.2927

Consideremos ahora el caso de n servidores con capacidad infinita y retroceso.

Todos los servidores tienen un servicio exponencial de media μ_i.

De cada etapa i un cliente se mueve a otra etapa con probabilidad r_{ij} donde para $i >= j$ se sabe que $r_i j = 0$.

Entonces la llegada a los servidores se define como $\lambda_i = \lambda_{i-1} + \sum_{j=i+1}^{n} p_{ij}\lambda_j$

Además sabemos por [1] que el comportamiento del valor medio permite seguir considerando cada cola como una M/M/1 independiente.

Entonces:

$L_i = \frac{\rho_i}{1-\rho_i} = \frac{\lambda_i}{\mu-\lambda_i}$

$W_i = \frac{L_i}{\lambda_i}$

$W = \sum W_i$

En este caso, después de largos días de búsqueda e investigación, no encontramos en la bibliografía proporcionada para el curso, ni en libros consultados en internet, una forma precisa de calcular la intensidad de llegada a cada servidor.

La fórmula expuesta anteriormente fue deducida por nosotros dado el conocimiento adquirido durante la investigación y la realización de este trabajo, pero no es posible calcularla, puesto que tiene recursividad hacia delante y hacia atrás. Por favor, si es tan amable de compartirnos bibliografía donde podamos encontrar una solución al problema propuesto sería de mucha ayuda.

Nos topamos con las series abiertas de Jackson, que se parecen en cierta medida, pero no encontramos evidencia que soportara que las mismas propiedades se cumplían, ya que en este caso solo el primer servidor recibe información del exterior y el reciclado o repetición de servidores solo es en retroceso.

Además encontramos algunos usos de fórmulas de Earlang, pero tampoco encontramos evidencia que sustentara su validez en el problema propuesto.

Referencias: [1] José Pedro García Sabater, Aplicando Teorıa de Colas en Direccion de Operaciones, 2015.