



Instituto Tecnológico de Costa Rica
Campus Tecnológico Central Cartago
Escuela de Ingeniería en Computación

Proyecto 2: Simulación de sistema distribuido

Principios de Sistemas Operativos - Grupo 2
Prof. Kenneth Roberto Obando Rodriguez

Daniel Granados Retana, carné 2022104692
Diego Manuel Granados Retana, carné 2022158363
David Fernández Salas, carné 2022045079

22 de Noviembre del 2024

IIS 2024

Tabla de contenidos

Introducción	2
Arquitectura del programa	5
Comunicación de procesos	6
Recursos compartidos	8
Clase Node	10
Clase Master	11
Pruebas y resultados	14
Prueba 1: Asignación de procesos y balanceo de carga	14
Prueba 2: Sincronización de recursos compartidos	17
Prueba 3: Manejo de fallos	21
Ejemplo 1	21
Ejemplo 2	23
Ejemplo 3	24
Prueba 4: Escalabilidad del sistema	26
Prueba 5: Redistribución automática de procesos	29
Bibliografía	32

Introducción

Este proyecto tiene el propósito de abordar los desafíos de la gestión eficiente de recursos y procesos en entornos distribuidos, los cuales deben garantizar la sincronización, tolerancia a fallos y escalabilidad. Un sistema distribuido es una forma de implementar la arquitectura de un sistema donde varias tareas se desarrollan en forma conjunta siempre con un enfoque cooperativo o competitivo (Arredondo et al., 1995). Para esto se crea una emulación de un sistema distribuido que simula un entorno de nodos interconectados. Cada nodo actúa como una instancia básica capaz de gestionar procesos y recursos, en la cual se interactúa con otros nodos para los cambios o fallos en la red.

El objetivo principal del proyecto es aplicar conceptos fundamentales de sistemas distribuidos para diseñar e implementar mecanismos que permitan la comunicación, sincronización y balanceo de carga entre nodos, garantizando al mismo tiempo la escalabilidad y la capacidad de recuperación ante fallos. Este enfoque permite emular escenarios reales de redes distribuidas y evaluar su desempeño bajo diferentes condiciones.

A través de esta simulación, se busca la comprensión de conceptos como:

- La asignación dinámica de procesos.
- La gestión coordinada de recursos compartidos.
- La tolerancia a fallos en sistemas distribuidos.
- La escalabilidad y el balanceo de carga en redes de nodos.

La presente documentación detalla el diseño, implementación y evaluación del emulador, destacando sus funcionalidades principales, casos de uso, pruebas realizadas y resultados obtenidos. Este trabajo no solo permite explorar los fundamentos de los sistemas distribuidos, sino que también fomenta el desarrollo de habilidades prácticas aplicables en el diseño de sistemas reales. La arquitectura fundamental con la cual se implementó el sistema es la de Master/Slave (o Worker), también conocida como Primary/Secondary.

Se adjunta un video que explica los casos de uso solicitados y las pruebas realizadas:

<https://www.youtube.com/watch?v=TC1m2JYcBTY>

Capítulos:

- 0:00 Introducción
- 0:10 Arquitectura
- 1:40 Ejecución
- 1:50 Master
- 5:05 Node
- 8:23 Caso de Uso 1: Asignación
- 10:55 Caso de Uso 2: Recursos compartidos
- 14:30 Caso de Uso 3: Manejo de fallos
- 17:45 Prueba 1: Asignación y Balanceo
- 21:26 Prueba 2: Sincronización de recursos
- 25:00 Prueba 3: Manejo de fallos
- 29:40 Prueba 4: Escalabilidad
- 31:18 Prueba 5: Redistribución

Ejecución del programa

Para automatizar la configuración del sistema distribuido para las pruebas, se implementó el manejo de la configuración a partir de un archivo YAML. Con esto, se busca emular la forma en que funciona Kubernetes. El YAML debe seguir una estructura como la siguiente:

```
system:
  node_quantity: 3
  node_capacities:
    - 3
    - 4
    - 5
  shared_resources:
    - name: total
      type: Integer
      values:
        initial_value: 0
  requests:
    - endpoint: http://localhost:5000/send
```

```

times: 11
method: POST
sleep: 0
body:
  type: python
  command: "def fibonacci(n):\n\tif n == 0:\n\t\treturn
0\n\telif n == 1:\n\t\treturn 1\n\telse:\n\t\treturn fibonacci(n-1) +
fibonacci(n-2)\n\nresponse = fibonacci(45)"

```

Así, en el YAML se puede especificar la cantidad de nodos trabajadores con el que inicia el sistema y la capacidad máxima de cada uno de los nodos. También, se pueden inicializar recursos compartidos en el sistema distribuido, como un número entero. Por último, se puede especificar la secuencia de solicitudes HTTP que se le van a enviar al nodo Master, ya que el protocolo de comunicación de los usuarios con el nodo Master es HTTP. Para cada request, se especifica el endpoint al cual se le va a enviar la solicitud, la cantidad de veces que se envía la solicitud al nodo, el método HTTP, el tiempo que se queda durmiendo, y por último el cuerpo de la instrucción. Este se codifica como un JSON donde hay un campo para un tipo, el cual puede ser “python” o “shell”, y un campo para el “command”, el cual contiene el código en Python o el comando de shell a ejecutar. Por ejemplo, en esta solicitud, el comando es una implementación “naive” para calcular el número n-ésimo de la secuencia de Fibonacci. Esta implementación es ineficiente a propósito, pero así se simula una carga computacional real en cada proceso.

Para ejecutar el programa solo se requiere correr el archivo main con el YAML específico el cual se recibe como input al correrlo. Además el programa cuenta con un logger para modificar la forma de hacer los prints del programa. El logger puede ser más o menos detallado. Cuenta con los siguientes modos del más detallado al menos detallado “DEBUG, INFO, WARNING, ERROR, CRITICAL”.

```

14 # Configure the logger
15 logging.basicConfig(
16     level=logging.DEBUG,
17     format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
18     datefmt="%Y-%m-%d %H:%M:%S.%f",
19 )

```

Arquitectura del programa

La arquitectura del sistema implementado se basa en el modelo de Master-Slave o Primary-Secondary. En este patrón de diseño, se tiene un nodo maestro y varios nodos “esclavos” o secundarios. El nodo master es el encargado de administrar los nodos trabajadores, distribuir las tareas y escalar el sistema (Sharma, 2023). Los nodos trabajadores son los que ejecutan los procesos que reciben. El nodo primario también se encarga de monitorear el estado de los nodos secundarios y de recuperar el sistema de las fallas cuando ocurran. Algunas de las principales ventajas de esta arquitectura son la posibilidad de procesamiento paralelo, la escalabilidad del sistema y la tolerancia a fallas. En el sistema, cada nodo es representado por su propio proceso de Python. Python, al no tener hilos puros debido al GIL y a que solo puede correr un solo hilo de ejecución a la vez (Python, s.f.b), el paralelismo puro se obtiene por medio de la creación de nuevos procesos. Así, para enviar información entre procesos, se usaron métodos de comunicación entre procesos, como variables de memoria compartida y pipes.

El sistema fue desarrollado en Python con el objetivo de aprovechar su sintaxis clara y legible, lo que facilita la comprensión del código y promueve una experiencia de aprendizaje accesible para quienes exploran el funcionamiento de los sistemas distribuidos. Otra ventaja es el uso de bibliotecas como multiprocessing, que permite gestionar múltiples hilos y procesos de manera eficiente. Esto es fundamental en la simulación de sistemas distribuidos, ya que facilita la comunicación entre nodos y la coordinación de las tareas asignadas a cada uno de ellos. Además, multiprocessing soporta la paralelización y sincronización, asegurando que los recursos compartidos sean manejados de forma segura y evitando condiciones de carrera.

Otra característica destacada de Python es su capacidad para ejecutar dinámicamente el código recibido a través de solicitudes (requests). Esto permite implementar un sistema flexible donde los nodos pueden procesar instrucciones directamente solo con el comando `exec()`.

La estructura de datos más importante usada en el sistema es un “deque”, o una cola de doble extremo. Las colas de tareas en todos los nodos son en realidad un

deque. Esto es para poder eliminar o añadir tareas de ambos extremos eficientemente. Se utilizó la implementación del módulo estándar de Python: *collections*. Estos objetos mantienen el control de su capacidad máxima, por lo que no es necesario guardar las capacidades de los nodos en una lista separada (Python, s.f.c). Asimismo, la documentación indica que son “thread-safe”, lo cual es especialmente importante en una aplicación multi hilos como este sistema para prevenir condiciones de carrera. Se puede hacer operaciones de “push” y “pop” en tiempo constante en ambos extremos, de manera segura para hilos y haciendo un uso eficiente de la memoria.

Comunicación de procesos

El rendimiento de un sistema distribuido depende ampliamente de la eficiencia en la comunicación de procesos. Esto implica implementar medidas entre los nodos para garantizar que los datos se puedan transmitir de manera rápida y sin pérdida. Además es sumamente importante considerar que deben de tener una correcta sincronización entre procesos para evitar condiciones de carrera y sobrecarga que puedan generar errores en la ejecución (Arredondo et al., 1995).

En la simulación del sistema la mayoría de la comunicación se hace mediante pipes que vienen de la librería de Python llamada “multiprocessing”. El módulo multiprocessing permite la creación y administración de procesos independientes por medio de pipes que permiten una comunicación entre procesos (Python, s.f.a). Cabe destacar que dichos pipes son bidireccionales, es decir, pueden intercambiar datos en ambas direcciones. Además, al crear un pipe se obtienen 2 extremos, uno siendo el `parent_pipe` y el otro siendo el `child_pipe`. Incluso dicho módulo tiene la posibilidad de crear nuevos procesos con el comando “process”, el cual es una instancia en ejecución de un programa, donde cada uno tiene su propio espacio de memoria Python (s.f.a). Es decir, los procesos son independientes de otros procesos, lo que los diferencia de un hilo. Además, al crear un proceso, este tiene un ciclo de vida separado al principal.

Los pipes se usan en dos puntos principales: para enviar instrucciones del nodo master al nodo trabajador y para enviar una solicitud para recibir el acceso a un recurso compartido. En el primer caso de uso, el nodo master y el nodo hijo ambos tienen un extremo del pipe. Cuando el nodo master detecta que un nodo está inactivo, le envía

un objeto de clase *Instruction* por el pipe. Estos pipes de la biblioteca *multiprocessing* son altamente usables debido a que serializan los objetos automáticamente por medio de la biblioteca de *pickle*. El nodo trabajador recibe el objeto *Instruction*, lo interpreta y ejecuta el comando. En el segundo caso de uso, los nodos trabajadores envían una solicitud para acceder a un recurso compartido a través del pipe. El nodo master está constantemente revisando estos pipes de acceso a recursos para atender las solicitudes que reciba. Cuando se recibe una solicitud, revisa si el recurso está disponible para enviar la referencia del mismo a través del pipe. Aquí, el nodo solicitante se queda esperando hasta que le respondan, ya que los pipes también pueden bloquear la ejecución.

Otro recurso utilizado de la biblioteca *multiprocessing* es el *Event*. Es una primitiva de sincronización que es en esencia una bandera que puede estar activada o desactivada (Python, s.f.b). Cada nodo tiene una bandera que desactiva cuando va a iniciar la ejecución de una tarea y reactiva cuando termina la ejecución. De acuerdo con este indicador, el nodo master sabe si enviarle una tarea para procesar o no.

En nuestra implementación de los sistemas distribuidos un claro ejemplo de comunicación entre procesos es el heartbeat, el cual se utiliza con el fin de conocer el estado de los nodos hijos. Este funciona con base en el principio de “heartbeat”, donde el hijo envía señales de vida periódicamente y si el padre no recibe estas señales en un tiempo definido lo reinicia (Silberschatz et al., 1991). En el sistema, el mecanismo de envío de heartbeats está implementado por medio de una variable de memoria compartida. En Python, la biblioteca *multiprocessing* ofrece esta funcionalidad. Con la función *Value()*, se puede declarar una variable compartida de un tipo específico compatible con los tipos propios al lenguaje C. La ventaja de usar esta biblioteca es que ya tiene un candado o lock incorporado en el objeto para asegurar la exclusión mutua entre procesos. En este caso, el value es de tipo *double* y almacena el resultado de *time.time()*. Esto es un número flotante de la cantidad de segundos que han pasado desde un punto base, o época. Así, el master valida que el último tiempo registrado por los nodos trabajadores no exceda un intervalo de 10 segundos. Si el valor compartido excede este tiempo máximo, genera un *timeout* y asume que el nodo está muerto. Redistribuye la carga y reinicia el nodo.

Otro punto a considerar es cómo el paralelismo influye en la eficiencia del sistema, permitiendo la ejecución de un gran número de computaciones de manera independiente en cada procesador.

Recursos compartidos

Los recursos compartidos son elementos que pueden ser accedidos, utilizados o manipulados de manera simultánea por múltiples procesos o hilos (Jiménez et al., 2017). En los sistemas distribuidos, su importancia radica en su capacidad para facilitar la coordinación entre nodos y optimizar la eficiencia del sistema. La implementación de recursos compartidos es fundamental, ya que permite sincronizar los procesos, garantizando una colaboración efectiva y evitando el desperdicio de recursos. Esto contribuye a maximizar su utilización, mejorando así el rendimiento global del sistema (Jiménez et al., 2017).

En nuestra implementación, utilizamos el módulo `multiprocessing.Manager`, que permite crear recursos compartidos para que múltiples procesos en Python puedan acceder a ellos y modificarlos de forma segura. Este módulo ofrece la capacidad de gestionar estructuras de datos simples como cadenas, enteros y flotantes mediante el uso del comando `manager.Value`. Además, extiende su funcionalidad a elementos más complejos, como listas, diccionarios y archivos, lo que incrementa su versatilidad en aplicaciones concurrentes.

Cada recurso compartido está protegido con un `ReadWriteLock`, una herramienta crucial para prevenir problemas de sincronización, los cuales son frecuentes al trabajar en entornos paralelos. Este mecanismo garantiza que las operaciones de lectura y escritura sobre los recursos se realicen de forma controlada y sin interferencias. Este candado permite que haya varios procesos compartiendo el recurso compartido en modo lectura. Por ejemplo, es posible que varios procesos operen sobre el mismo arreglo simultáneamente. También, el modo de escritura sí es exclusivo, lo cual garantiza la exclusión mutua al momento de escribir.

Para gestionar el acceso a los recursos compartidos, se han implementado funciones específicas tanto para la lectura como para la escritura:

1. Lectura de recursos:

- a. La función `request_read_resource` permite que un nodo adquiera un bloqueo de lectura sobre un recurso específico. Además, registra el recurso como en uso dentro del diccionario de recursos asignados al nodo. Del lado del nodo trabajador, envía una solicitud al master por medio de un pipe. Del lado del nodo master, es representado por un hilo que se queda esperando a que pueda adquirir el control sobre el recurso compartido.
 - b. La función `release_read_resource` libera el bloqueo de lectura y actualiza el registro de los recursos que están asignados, asegurando que el recurso quede disponible para otros nodos. Del lado del nodo, simplemente envía un mensaje diciendo que ya se puede liberar su control sobre el recurso compartido. Del lado del nodo master, actualiza el estado del recurso en uso.
2. Escritura de recursos:
- a. La función `request_write_resource` permite que un nodo adquiera un bloqueo de escritura sobre un recurso, lo que asegura que solo ese nodo pueda modificarlo durante el tiempo en que el bloqueo esté activo. También registra esta acción en el sistema y genera un registro en el log para mantener un control claro de los accesos.
 - b. La función `release_write_resource` libera el bloqueo de escritura y actualiza los registros correspondientes, dejando el recurso listo para ser utilizado por otros procesos.

Para una gestión efectiva de solicitudes de acceso se han desarrollado dos funciones adicionales: `handle_read_request` y `handle_write_request`. Estas funciones supervisan las solicitudes de lectura y escritura, verificando el estado del nodo solicitante antes de autorizar el acceso. En el caso de que exista un error con el nodo este se encarga de liberar el recurso para que no suceda un bloqueo indefinido y mantener la integridad del sistema.

Clase Node

Esta clase está diseñada para simular el comportamiento de un nodo, el cual es capaz de escuchar instrucciones, ejecutar comandos dinámicamente, gestionar los recursos compartidos y enviar los “heartbeat”. El nodo funciona como una unidad autónoma dentro de una red de varios nodos, los cuales interactúan con el maestro. El rol de dicho nodo es ejecutar las instrucciones recibidas, ya sean de Python o de shell. Estas instrucciones se reciben con un pipe con el nodo master. También, la clase Node cuenta con un objeto Event para indicar cuando está ocupado con una tarea o cuando está ocioso.

Además, el nodo debe solicitar acceso a los recursos compartidos por medio de métodos específicos, donde se coordinan los permisos de lectura y escritura para evitar conflictos. Finalmente, mantiene la comunicación activa con su padre “máster” donde envía señales periódicas, o “heartbeats”, para indicar que sigue activo.

Entre los métodos principales está listen(), el cual es un método que escucha continuamente instrucciones a través del pipe. Este puede ejecutar las instrucciones de tipo Stop, python y shell. La función de send_heartbeat() es un hilo en segundo plano que envía una señal de heartbeat al proceso maestro cada 5 segundos, actualizando la variable compartida heartbeat con el tiempo actual.

Para la gestión de recursos hay varios métodos, entre ellos request_read_resource, request_write_resource, release_read_resource y release_write_resource. Estos métodos permiten al nodo interactuar con recursos compartidos de forma sincronizada. La comunicación entre el nodo master y los nodos trabajadores para la solicitud y recepción de recursos compartidos se realiza por medio de un pipe bidireccional.

Dicha clase Node tiene siempre dos hilos siempre activos siendo el principal y el hilo que envía el heartbeat. El hilo principal es el ciclo que se llama listen el cual se llama al crear un nodo, dicho hilo es infinito por lo cual permanece escuchando instrucciones provenientes del pipe asignado. El segundo hilo con la función de heartbeat se encarga de siempre enviar las actualizaciones sobre el estado del nodo al master.

Clase Master

La función principal de la clase Master es coordinar la ejecución de múltiples nodos mediante procesos y recursos compartidos. Una de sus primeras responsabilidades es la gestión de nodos, lo cual realiza con la función `initialize_node`. Esta se encarga de inicializar cada nodo, que actúa como un proceso independiente encargado de ejecutar tareas. Al crear un nuevo nodo se le asigna un identificador único, un pipe de comunicación, otro pipe para los heartbeats, otro pipe para los recursos y una cola de tareas. Esta cola de tareas es un deque con una longitud máxima.

La función `__requeue_tasks` es clave para manejar la redistribución de tareas en caso de fallos o cambios en el sistema. Esta función extrae todas las tareas asignadas a un nodo específico y las reubica en la cola del máster, para que este se encargue de reasignarlas a otros nodos disponibles. La cola del máster es un deque sin límite de tamaño.

El monitoreo de los nodos se realiza con la función `__check_nodes_status()`, que verifica periódicamente los heartbeats. Esta función ejecuta un hilo continuo encargado de asegurar que todos los nodos envían señales de vida cada 5 segundos. Si un nodo no envía señales durante 10 segundos, el máster lo considera inactivo e inicia el mecanismo de recuperación. En este caso lo que tiene que hacer el master es notificar que un hijo no responde, colocarlo como un nodo inactivo, reasignar las tareas que están pendientes, liberar los recursos reservados por el proceso y finalmente crear nuevamente el mismo hijo. Para reasignar las tareas pendientes, las coloca al inicio de la cola maestra para que tengan prioridad a la hora de ser distribuidas. Esto se ejecuta en la función `__requeue_tasks()`. Marcar el nodo como inactivo lo excluye de la rotación de round-robin. Hace que se lo salten en el proceso de `__assign_tasks()` y `select_next_node()` hasta que vuelva a ser marcado como activo y tenga su nueva cola. También, la liberación de los recursos compartidos es posible porque se mantiene un conjunto de los recursos a los cuales cada nodo tiene acceso, por lo que se itera por esta colección y se van liberando los modos de acuerdo con su modo de acceso. Esto se realiza en la función `release_resources`. Por último, se vuelve a crear un proceso

para ese mismo identificador con la misma capacidad máxima con el método de `__create_node()`. Ya en este momento puede atender instrucciones nuevamente.

La función `select_next_node` determina a qué nodo se le asignará la próxima tarea, implementando un algoritmo de selección circular “round-robin”. Este algoritmo considera el tamaño de la cola de cada nodo y su estado (disponible u ocupado). Si un nodo no está operativo, las tareas pendientes se reasignan automáticamente a otros nodos activos. Asimismo, si la cola del nodo está llena, también se lo salta de la ronda de asignación.

La clase master gestiona varios hilos donde uno de ellos es el `__check_nodes_status` el cual se encarga de verificar periódicamente el estado de los nodos y reinicia aquellos que no están respondiendo. Otro hilo es `__distribute_tasks` el cual asigna áreas desde la cola maestra a los nodos disponibles. Existe otro hilo que está estrechamente relacionado, el cual se llama `__assign_tasks`, que envía tareas específicas a nodos individuales y gestiona el trabajo entre nodos. Después, recibe las respuestas de los nodos. También implementa el algoritmo de “work-stealing”, que ocurre cuando un nodo está ocioso y hay otro nodo con más de 1 tarea en su cola. El nodo ocioso le roba la última tarea de la cola al otro nodo. Finalmente está el hilo que se encarga de gestionar el acceso a recursos compartidos por los nodos utilizando bloqueos de lectura y escritura para evitar condiciones de carrera y la pérdida de información.

Para que el sistema pueda interactuar con solicitudes HTTP se usan rutas por medio de un API RESTful con Flask. Dentro de las opciones de rutas están enviar instrucciones específicas a un nodo o al maestro para que éste las asigne. Además se puede detener un nodo específico más que todo con el fin de casos de prueba. Incluso se pueden agregar nodos al sistema o crear los recursos compartidos para los nodos.

Para recibir las instrucciones de los usuarios, el master tiene un servidor HTTP. Este servidor solo recibe métodos POST. Las rutas o endpoints que hay son los siguientes:

1. `/send/<node_id>`: Recibe una instrucción en el cuerpo del request y la asigna a la cola del nodo mencionado en la ruta

POST localhost:5000/send/0 Send

Query Headers² Auth **Body¹** Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```

1  {
2    "type": "python",
3    "command": "def fibonacci(n):\n\tif n == 0:\n\t\treturn 0\n\telif n == 1\n\t\t:\n\t\t\treturn 1\n\telse:\n\t\t\treturn fibonacci(n-1) + fibonacci(n-2\n\t)\nresponse = fibonacci(42)"
4  }
5

```

2. /send: Recibe una instrucción y la agrega a la cola que mantiene el nodo Master, para que este luego las asigne a un nodo.

POST localhost:5000/send Send

Query Headers² Auth **Body¹** Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

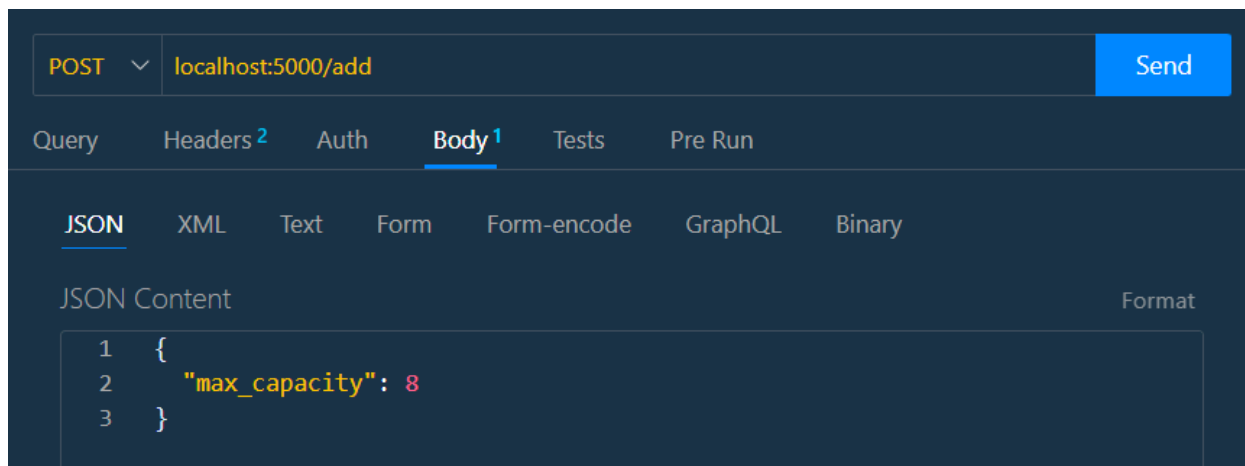
```

1  {
2    "type": "python",
3    "command": "import time\nshared_list = request_read_resource\n('shared_list')\nresponse = 0\nprint(shared_list)\nif shared_list\n:\n\t\ttry:\n\t\t\tfor i in range(3):\n\t\t\t\t\tresponse +=\n\t\t\t\t\tshared_list[i]\n\t\t\t\t\ttime.sleep(30)\n\t\t\t\t\tfinally\n\t\t\t\t\t:\n\t\t\t\t\t\trelease_read_resource('shared_list')\n\t\t\t\t\t\telse:\n\t\t\t\t\t\tresponse =\n\t\t\t\t\t\t'Resource acquisition failed'"
4  }
5

```

3. /resource: Se usa para añadir recursos compartidos. Recibe el nombre ("name"), el tipo ("type") y el valor inicial ("values": "initial_value:"). El tipo puede ser un archivo, string, entero, número de punto flotante, booleano, lista, diccionario o tupla.

4. /add: Se usa para añadir un nodo al sistema. En el cuerpo del request debe venir la capacidad máxima de la cola de procesos de ese nodo.



5. /stop/<node_id>: Se usa para ordenar la terminación de un nodo. Por ejemplo, para terminar el Nodo 2, se usaría la siguiente ruta:



Pruebas y resultados

Prueba 1: Asignación de procesos y balanceo de carga

Esta prueba tiene como propósito verificar que el sistema asigna los procesos al nodo correcto. En nuestro sistema, para la asignación de procesos implementamos un mecanismo de Round Robin, donde los nodos se asignan en orden. Un proceso nuevo no necesariamente se asigna al que tenga menos procesos encolados, sino al que sigue en el orden y tiene espacio en su cola.

A continuación está la configuración de la prueba que ejecutamos:

```

Prueba1.yaml
1  system:
2    node_quantity: 3
3    node_capacities:
4      - 3
5      - 4
6      - 5
7    shared_resources:
8      - name: total
9        type: Integer
10       values:
11         initial_value: 0
12    requests:
13      - endpoint: http://localhost:5000/send
14        times: 11
15        method: POST
16        sleep: 0
17        body:
18          type: python
19          command: "def fibonacci(n):\n\tif n == 0:\n\t\treturn 0\n\telif n == 1:\n\t\treturn 1\n\telse:\n\t\treturn fibonacci(n-1) + fibonacci(n-2)\n\nresponse = fibonacci(45)"

```

En esta prueba, el sistema va a tener tres nodos con capacidades de 3, 4 y 5 procesos. Luego, se ejecuta una operación costosa de calcular: el Fibonacci número 45 de la secuencia. Esta operación se envía 11 veces, por lo que el Nodo 0 debería tener 3 procesos en su cola, el Nodo 1, 4 y el Nodo 2, 4. Esto se usa para demostrar que se respeta el tamaño de la cola de cada Nodo, ya que el Nodo 0 debería recibir el décimo proceso, pero lo va a recibir el Nodo 1 porque en el 0 ya no cabe.

```

2024-11-16 11:28:51 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-16 11:28:53 - Node 0-33020 - DEBUG - Heartbeat sent
2024-11-16 11:28:53 - Node 1-72924 - DEBUG - Heartbeat sent
2024-11-16 11:28:53 - Node 2-63076 - DEBUG - Heartbeat sent
2024-11-16 11:28:53 - werkzeug - INFO - 127.0.0.1 - - [16/Nov/2024 11:28:53] "POST /send HTTP/1.1" 200 -
2024-11-16 11:28:53 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/11" 200 35
- Código de respuesta: 200
- Respuesta: {"message": "Instruction received"}

2024-11-16 11:28:53 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-16 11:28:53 - Master - INFO - Queueing instruction to Node 0
2024-11-16 11:28:53 - Master - INFO - Status: True
2024-11-16 11:28:53 - Master - INFO - Node 0's queue length: 1
2024-11-16 11:28:53 - Master - INFO - Sending instruction to node 0
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
response = fibonacci(45)
2024-11-16 11:28:55 - werkzeug - INFO - 127.0.0.1 - - [16/Nov/2024 11:28:55] "POST /send HTTP/1.1" 200 -
2024-11-16 11:28:55 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/11" 200 35
- Código de respuesta: 200
- Respuesta: {"message": "Instruction received"}

2024-11-16 11:28:55 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-16 11:28:55 - Master - INFO - Queueing instruction to Node 1
2024-11-16 11:28:55 - Master - INFO - Status: True
2024-11-16 11:28:55 - Master - INFO - Node 1's queue length: 1
2024-11-16 11:28:55 - Master - INFO - Sending instruction to node 1

```



```

2024-11-16 11:28:57 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-16 11:28:57 - Master - INFO - Queueing instruction to Node 2
2024-11-16 11:28:57 - Master - INFO - Status: True
2024-11-16 11:28:57 - Master - INFO - Node 2's queue length: 1
2024-11-16 11:28:57 - Master - INFO - Sending instruction to node 2

```

Aquí se puede ver la asignación Round Robin de los procesos. Primero se envía el proceso al Nodo 0, luego al 1 y por último al 2.

```

2024-11-16 11:29:05 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-16 11:29:05 - Master - INFO - Queueing instruction to Node 0
2024-11-16 11:29:05 - Master - INFO - Status: True
2024-11-16 11:29:05 - Master - INFO - Node 0's queue length: 3
2024-11-16 11:29:07 - werkzeug - INFO - 127.0.0.1 - - [16/Nov/2024 11:29:07] "POST /send HTTP/1.1" 200 -
2024-11-16 11:29:07 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/1.1" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

```

Aquí se evidencia cuando la cola del Nodo 0 se llena con tres procesos.

```

2024-11-16 11:29:09 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-16 11:29:09 - Master - INFO - Queueing instruction to Node 2
2024-11-16 11:29:09 - Master - INFO - Status: True
2024-11-16 11:29:09 - Master - INFO - Node 2's queue length: 3
2024-11-16 11:29:11 - werkzeug - INFO - 127.0.0.1 - - [16/Nov/2024 11:29:11] "POST /send HTTP/1.1" 200 -
2024-11-16 11:29:11 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/1.1" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

2024-11-16 11:29:11 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-16 11:29:11 - Master - INFO - Queueing instruction to Node 1
2024-11-16 11:29:11 - Master - INFO - Status: True
2024-11-16 11:29:11 - Master - INFO - Node 1's queue length: 4
2024-11-16 11:29:13 - Node 0-33020 - DEBUG - Heartbeat sent
2024-11-16 11:29:13 - Node 1-72924 - DEBUG - Heartbeat sent
2024-11-16 11:29:13 - Node 2-63076 - DEBUG - Heartbeat sent
2024-11-16 11:29:13 - werkzeug - INFO - 127.0.0.1 - - [16/Nov/2024 11:29:13] "POST /send HTTP/1.1" 200 -
2024-11-16 11:29:13 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/1.1" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

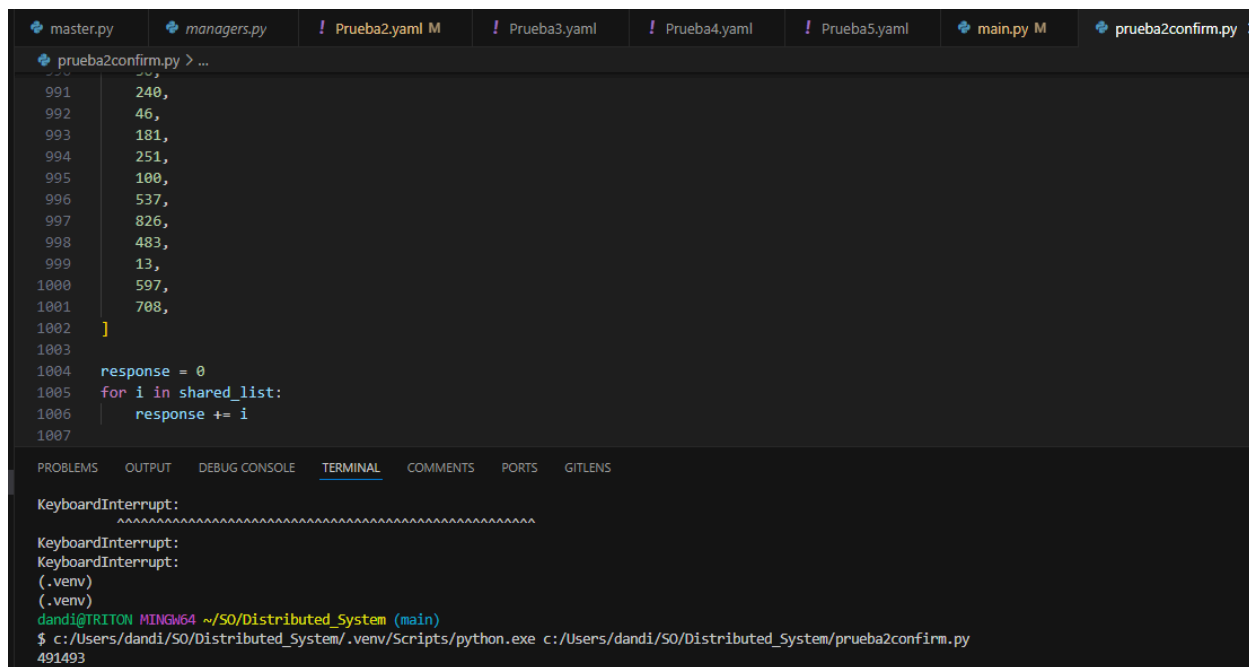
2024-11-16 11:29:13 - Master - INFO - Queueing instruction to Node 2
2024-11-16 11:29:13 - Master - INFO - Status: True
2024-11-16 11:29:13 - Master - INFO - Node 2's queue length: 4

```

Luego, podemos ver que el Nodo 2 tiene 3 procesos en su cola, por lo que ese era el proceso número 9. El siguiente tiene que ser el décimo proceso, que debería ir al Nodo 0. No obstante, como el Nodo 0 está lleno, el proceso se envía al Nodo 1. Posteriormente, podemos ver que el siguiente Nodo al que se le asigna un proceso es el 2, siguiendo correctamente el orden Round Robin.

Prueba 2: Sincronización de recursos compartidos

Esta prueba consiste en comprobar que los nodos acceden a los recursos compartidos de forma sincronizada. Para esta prueba, se va a realizar la suma de una lista de elementos de forma paralelizada.



The screenshot shows a code editor with several tabs: master.py, managers.py, Prueba2.yaml M, Prueba3.yaml, Prueba4.yaml, Prueba5.yaml, main.py M, and prueba2confirm.py. The active file is prueba2confirm.py, which contains the following code:

```

991     240,
992     46,
993     181,
994     251,
995     100,
996     537,
997     826,
998     483,
999     13,
1000    597,
1001    788,
1002 ]
1003
1004 response = 0
1005 for i in shared_list:
1006     response += i
1007

```

The terminal output shows the following:

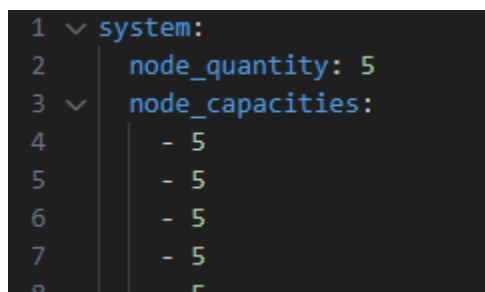
```

KeyboardInterrupt:
~~~~~
KeyboardInterrupt:
KeyboardInterrupt:
(.venv)
(.venv)
dandi@TRITON MINGW64 ~/SO/Distributed_System (main)
$ c:/Users/dandi/SO/Distributed_System/.venv/Scripts/python.exe c:/Users/dandi/SO/Distributed_System/prueba2confirm.py
491493

```

Este es el resultado que se debe dar si la suma se realiza secuencialmente.

Está la configuración de la prueba:



The screenshot shows a YAML configuration file with the following content:

```

1 system:
2   node_quantity: 5
3   node_capacities:
4     - 5
5     - 5
6     - 5
7     - 5
8     - 5

```

```

9  shared_resources:
10 - name: shared_list
11   type: List
12   values:
13     initial_value: [302, 498, 100, 227, 44, 493, 157, 761, 708, 873, 789, 729, 749, 701, 311, 473, 842, 71, 429, 686, 437, 782, 7, 357, 531, 781]
14 - name: total
15   type: Integer
16   values:
17     initial_value: 0
18 requests:
19 - endpoint: http://localhost:5000/send
20   times: 1
21   method: POST
22   sleep: 0
23   body:
24     type: python
25     command: "import time\nshared_list = request_read_resource('shared_list')\nresponse = 0\nprint(shared_list)\nif shared_list:\n\ttry:\n\t\tfor i in
26 - endpoint: http://localhost:5000/send
27   times: 1
28   method: POST
29   sleep: 0
30   body:
31     type: python
32     command: "import time\nshared_list = request_read_resource('shared_list')\nresponse = 0\nprint(shared_list)\nif shared_list:\n\ttry:\n\t\tfor i in
33 - endpoint: http://localhost:5000/send
34   times: 1
35   method: POST
36   sleep: 0
37   body:
38     type: python
39     command: "import time\nshared_list = request_read_resource('shared_list')\nresponse = 0\nprint(shared_list)\nif shared_list:\n\ttry:\n\t\tfor i in
40 - endpoint: http://localhost:5000/send
41   times: 1
42   method: POST
43   sleep: 0
44   body:
45     type: python
46     command: "import time\nshared_list = request_read_resource('shared_list')\nresponse = 0\nprint(shared_list)\nif shared_list:\n\ttry:\n\t\tfor i in

1049 - endpoint: http://localhost:5000/send
1050   times: 1
1051   method: POST
1052   sleep: 20
1053   body:
1054     type: python
1055     command: "import time\nshared_list = request_read_resource('shared_list')\nresponse = 0\nprint(shared_list)\nif shared_list:\n\ttry:\n\t\tfor i in
range(800, 1000):\n\t\t\tresponse += shared_list[i]\n\tfinally:\n\t\t\trelease_read_resource('shared_list')\n\telse:\n\t\t\tresponse = 0\n\ttotal =
request_write_resource('total')\n\ttry:\n\t\t\ttotal.value += response\n\tfinally:\n\t\t\trelease_write_resource('total')\n"
1056 - endpoint: http://localhost:5000/send
1057   times: 1
1058   method: POST
1059   sleep: 0
1060   body:
1061     type: python
1062     command: "total = request_read_resource('total')\nresponse = total.value\nprint(response)\n"

```

Hay cinco nodos y dos recursos compartidos. `shared_list` es la lista que contiene los números por sumar y `total` es la variable compartida donde se almacenará el resultado de las sumas. `shared_list` es una secuencia de 1000 números aleatorios. Los primero cinco requests son los que solicitan la suma de un rango de los números, como los de la posición 0 a 200 y así sucesivamente hasta terminar toda la lista. El último request solicita la impresión del resultado de las sumas.

```

2024-11-15 19:18:07 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-15 19:18:08 - Master - INFO - Queueing instruction to Node 0
2024-11-15 19:18:08 - Master - INFO - Status: True
2024-11-15 19:18:08 - Master - INFO - Node 0's queue length: 1
2024-11-15 19:18:08 - Master - INFO - Sending instruction to node 0
import time
shared_list = request_read_resource('shared_list')
response = 0
print(shared_list)
if shared_list:
    try:
        for i in range(200):
            response += shared_list[i]
        finally:
            release_read_resource('shared_list')
    else:
        response = 0
total = request_write_resource('total')
try:
    total.value += response
finally:
    release_write_resource('total')

```

Aquí se ve que las instrucciones se envían correctamente a los nodos.

```

2024-11-15 19:18:09 - Master - INFO - Node 0 has acquired read access to resource shared_list
[302, 498, 100, 227, 44, 493, 157, 761, 708, 873, 789, 729, 749, 701, 311, 473, 842, 71, 429, 686, 437, 782, 7, 357, 531, 781, 120, 794, 557, 355, 34, 817, 171, 252, 798,
347, 696, 513, 564, 178, 94, 845, 615, 238, 731, 464, 380, 522, 111, 893, 459, 236, 104, 225, 292, 538, 177, 224, 857, 87, 336, 502, 991, 30, 23, 799, 60, 901, 307, 196,
69, 44, 393, 384, 294, 265, 758, 834, 584, 528, 74, 951, 303, 515, 252, 633, 34, 873, 464, 41, 787, 96, 89, 425, 291, 928, 203, 810, 909, 74, 206, 317, 725, 537, 688, 7,
326, 447, 78, 878, 867, 159, 773, 623, 546, 634, 401, 167, 660, 988, 724, 58, 935, 652, 465, 597, 543, 567, 808, 328, 140, 452, 120, 813, 990, 464, 161, 465, 419, 443, 3,
5, 593, 95, 73, 790, 266, 441, 982, 390, 615, 999, 156, 850, 918, 9, 939, 504, 475, 946, 513, 709, 868, 874, 416, 838, 886, 251, 125, 227, 21, 397, 663, 237, 980, 496, 34,
8, 608, 4, 895, 135, 648, 354, 846, 210, 608, 887, 557, 162, 246, 977, 24, 597, 96, 961, 942, 404, 748, 53, 585, 832, 939, 39, 959, 844, 950, 678, 269, 836, 572, 167, 271,
, 31, 952, 338, 400, 174, 694, 545, 24, 521, 33, 580, 37, 207, 828, 196, 177, 444, 978, 492, 954, 258, 79, 679, 628, 645, 135, 768, 235, 199, 829, 117, 310, 34, 508, 909,
959, 312, 71, 804, 379, 261, 68, 585, 594, 239, 331, 816, 272, 818, 942, 49, 833, 89, 685, 403, 597, 14, 334, 269, 71, 203, 730, 575, 545, 38, 374, 76, 761, 579, 980, 75,
7, 170, 184, 557, 524, 184, 920, 632, 423, 349, 620, 158, 539, 776, 905, 328, 487, 340, 968, 156, 982, 428, 828, 358, 118, 614, 24, 720, 584, 53, 655, 918, 324, 399, 657,
883, 412, 763, 986, 512, 244, 922, 785, 564, 929, 288, 788, 58, 144, 977, 455, 513, 589, 940, 619, 307, 368, 757, 773, 686, 89, 558, 301, 925, 370, 908, 739, 128, 464, 7,
55, 978, 436, 754, 518, 372, 537, 858, 778, 907, 55, 726, 477, 204, 386, 285, 778, 547, 879, 564, 679, 299, 320, 444, 919, 311, 299, 170, 473, 618, 271, 958, 605, 260, 63,
8, 131, 891, 929, 944, 794, 260, 781, 965, 446, 291, 460, 322, 664, 952, 407, 885, 771, 730, 342, 997, 16, 290, 492, 74, 354, 282, 364, 739, 25, 124, 64, 649, 27, 346, 59,
5, 367, 643, 887, 697, 685, 510, 724, 224, 433, 383, 902, 606, 120, 522, 968, 134, 795, 231, 891, 334, 119, 81, 698, 835, 804, 443, 95, 188, 520, 105, 134, 237, 538, 50,
36, 296, 522, 83, 457, 986, 814, 807, 870, 367, 579, 87, 737, 84, 184, 312, 894, 959, 609, 321, 495, 781, 680, 930, 892, 689, 173, 514, 280, 442, 445, 669, 30, 97, 785, 9,
, 86, 297, 711, 366, 969, 907, 566, 577, 254, 775, 359, 726, 222, 676, 747, 914, 823, 600, 935, 546, 247, 146, 669, 464, 932, 514, 218, 469, 204, 803, 736, 628, 734, 749,
111, 773, 671, 29, 503, 989, 499, 204, 407, 564, 162, 375, 869, 451, 199, 386, 498, 635, 835, 79, 758, 210, 888, 536, 171, 866, 862, 66, 516, 127, 313, 152, 646, 300, 69,
8, 633, 877, 294, 919, 41, 576, 822, 801, 854, 876, 989, 633, 623, 156, 884, 47, 298, 48, 79, 127, 440, 606, 441, 951, 13, 625, 235, 366, 667, 669, 658, 850, 851, 325, 91,
7, 481, 803, 463, 64, 883, 997, 399, 399, 619, 175, 293, 573, 291, 868, 529, 556, 580, 749, 554, 825, 275, 2, 181, 150, 378, 915, 5, 616, 416, 681, 308, 469, 700, 561, 32,
7, 793, 105, 603, 945, 683, 57, 486, 126, 92, 729, 327, 224, 705, 931, 508, 584, 292, 727, 484, 870, 410, 837, 115, 592, 488, 196, 830, 834, 322, 273, 451, 477, 366, 256,
933, 417, 982, 912, 311, 357, 336, 582, 850, 923, 605, 397, 60, 761, 793, 670, 160, 33, 15, 829, 390, 763, 303, 137, 613, 624, 327, 781, 815, 845, 328, 425, 729, 322, 68,
5, 862, 344, 675, 645, 324, 278, 895, 999, 935, 745, 567, 746, 572, 851, 655, 77, 492, 522, 962, 682, 5, 30, 201, 879, 836, 87, 475, 310, 492, 49, 183, 602, 862, 368, 417,
, 428, 257, 169, 761, 731, 182, 389, 996, 702, 430, 317, 189, 363, 63, 112, 735, 801, 335, 629, 383, 546, 354, 474, 687, 533, 365, 534, 615, 409, 515, 239, 158, 60, 10, 1,
53, 99, 401, 249, 813, 885, 351, 653, 241, 678, 567, 567, 355, 430, 859, 801, 709, 599, 30, 800, 622, 893, 973, 340, 578, 553, 84, 403, 546, 638, 417, 641, 994, 412, 502,
713, 819, 463, 380, 718, 318, 597, 729, 957, 924, 980, 799, 46, 158, 384, 607, 401, 253, 939, 475, 597, 774, 210, 33, 592, 674, 919, 880, 493, 255, 287, 448, 944, 174, 1,
31, 796, 221, 361, 464, 252, 532, 316, 883, 106, 924, 788, 587, 90, 465, 314, 194, 245, 332, 653, 561, 383, 292, 520, 698, 467, 396, 631, 545, 28, 492, 39, 39, 267, 59, 2,
21, 409, 675, 372, 438, 394, 859, 150, 72, 551, 931, 479, 288, 873, 578, 112, 100, 358, 342, 729, 687, 821, 268, 540, 180, 748, 65, 547, 344, 568, 59, 486, 36, 3, 27, 429,
, 512, 214, 975, 584, 510, 683, 807, 103, 330, 343, 255, 78, 950, 280, 93, 842, 186, 731, 4, 591, 147, 844, 440, 876, 701, 766, 371, 573, 592, 295, 295, 454, 517, 33, 179,
, 299, 548, 409, 405, 978, 647, 834, 983, 567, 350, 768, 337, 884, 592, 833, 344, 565, 745, 332, 470, 933, 197, 223, 383, 633, 617, 655, 140, 229, 613, 350, 187, 71, 103,
826, 139, 45, 420, 284, 813, 804, 380, 859, 557, 736, 518, 36, 240, 46, 181, 251, 100, 537, 826, 483, 13, 597, 708]
2024-11-15 19:18:09 - werkzeug - INFO - 127.0.0.1 - - [15/Nov/2024 19:18:09] "POST /send HTTP/1.1" 200 -

```

Primero, adquieren el permiso de lectura de la lista e imprimen los números.

```

2024-11-15 19:18:10 - Master - INFO - Node 0 has released read access to resource shared_list

```

Luego, adquieren el permiso de escritura de la variable compartida total.

```

2024-11-15 19:18:11 - Master - INFO - Node 0 has acquired write access to resource total

```

```
2024-11-15 19:18:12 - Master - INFO - Node 0 has released write access to resource total
2024-11-15 19:18:12 - Node 0-51368 - INFO - 95651
2024-11-15 19:18:12 - Master - INFO - Node 1 has acquired write access to resource total
```

Finalmente, el nodo imprime su resultado. Aquí podemos ver que, luego de que el nodo 0 liberara el permiso de escritura, el nodo 1 lo pudo adquirir.

```
2024-11-15 19:18:13 - Master - INFO - Node 0 response: 95651
```

Este es el resultado que obtuvo el nodo 0.

```
2024-11-15 19:18:14 - Master - INFO - Node 1 response: 102736
```

Este es el resultado que obtuvo el nodo 1.

```
2024-11-15 19:18:16 - Master - INFO - Node 2 response: 100683
```

Este es el resultado que obtuvo el nodo 2.

```
2024-11-15 19:18:18 - Master - INFO - Node 3 response: 100062
```

Este es el resultado que obtuvo el nodo 3.

```
2024-11-15 19:18:20 - Master - INFO - Node 4 response: 92361
```

Este es el resultado que obtuvo el nodo 4.

```
2024-11-20 14:54:36 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-20 14:54:38 - werkzeug - INFO - 127.0.0.1 - - [20/Nov/2024 14:54:38] "POST /send HTTP/1.1" 200 -
2024-11-20 14:54:38 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/1.1" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

2024-11-20 14:54:38 - Master - INFO - Queueing instruction to Node 0
2024-11-20 14:54:38 - Master - INFO - Node 0's queue length: 1
2024-11-20 14:54:38 - Master - INFO - Sending instruction to node 0
total = request_read_resource('total')
response = total.value
print(response)

2024-11-20 14:54:38 - Master - INFO - Node 0 has acquired read access to resource total
491493
2024-11-20 14:54:38 - Node 0-9996 - INFO - 491493
2024-11-20 14:54:38 - Node 0-9996 - DEBUG - Heartbeat sent
2024-11-20 14:54:38 - Node 1-12020 - DEBUG - Heartbeat sent
2024-11-20 14:54:38 - Node 2-30888 - DEBUG - Heartbeat sent
2024-11-20 14:54:38 - Node 3-8684 - DEBUG - Heartbeat sent
2024-11-20 14:54:38 - Node 4-82180 - DEBUG - Heartbeat sent
2024-11-20 14:54:39 - Master - INFO - Node 0 response: 491493
2024-11-20 14:54:39 - Master - INFO - Releasing resources held by Node 0
2024-11-20 14:54:39 - Master - INFO - Node 0 has released read access to resource total
```

Aquí podemos ver que el último request imprime el resultado de la operación, el cual es correcto. Así, se puede comprobar que se coordina efectivamente el acceso a

recursos compartidos en el sistema distribuido. Adicionalmente, se liberan los recursos no liberados por el proceso.

Prueba 3: Manejo de fallos

Ejemplo 1

Esta prueba se encarga de verificar que el sistema redistribuye correctamente los procesos en caso de fallo de un nodo. En este caso, le vamos a dar una tarea a un nodo y después lo vamos a matar para visualizar cómo las tareas pendientes se encolan y se asignan a los nodos restantes. En este caso el archivo YAML se encarga de enviar 10 tareas de Fibonacci a tres nodos con una capacidad máxima de 5 tareas en la cola. Después, la última instrucción usa el endpoint de “/stop” para interrumpir la ejecución del nodo 0.

```

1 system:
2   node_quantity: 3
3   node_capacities:
4     - 5
5     - 5
6     - 5
7   shared_resources:
8     - name: shared_list
9       type: list
10      values:
11        initial_value: [1, 2, 3, 4, 5]
12  requests:
13    - endpoint: http://localhost:5000/send
14      times: 10
15      method: POST
16      sleep: 0
17      body:
18        type: python
19        command: "def fibonacci(n):\n\tif n == 0:\n\t\treturn 0\n\telif n == 1:\n\t\treturn 1\n\telse:\n\t\treturn fibonacci(n-1) + fibonacci(n-2)\n\nresponse = f
20    - endpoint: http://localhost:5000/stop/0
21      times: 1
22      method: POST
23      sleep: 0

```

En la siguiente imagen, se muestra cómo las tareas se van asignando en orden round-robin entre los nodos.

```

2024-11-19 12:47:03 - werkzeug - INFO - 127.0.0.1 - - [19/Nov/2024 12:47:03] "POST /send HTTP/1.1" 200 -
2024-11-19 12:47:03 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/1.1" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

2024-11-19 12:47:03 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-19 12:47:04 - Master - INFO - Queueing instruction to Node 0
2024-11-19 12:47:04 - Master - INFO - Node 0's queue length: 1
2024-11-19 12:47:04 - Master - INFO - Sending instruction to node 0
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
response = fibonacci(40)
2024-11-19 12:47:06 - werkzeug - INFO - 127.0.0.1 - - [19/Nov/2024 12:47:06] "POST /send HTTP/1.1" 200 -
2024-11-19 12:47:06 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/1.1" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

2024-11-19 12:47:06 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-19 12:47:06 - Master - INFO - Queueing instruction to Node 1
2024-11-19 12:47:06 - Master - INFO - Node 1's queue length: 1
2024-11-19 12:47:06 - Master - INFO - Sending instruction to node 1
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
response = fibonacci(40)
2024-11-19 12:47:06 - Node 0-21396 - DEBUG - Heartbeat sent
2024-11-19 12:47:06 - Node 1-28416 - DEBUG - Heartbeat sent
2024-11-19 12:47:06 - Node 2-32564 - DEBUG - Heartbeat sent
2024-11-19 12:47:08 - werkzeug - INFO - 127.0.0.1 - - [19/Nov/2024 12:47:08] "POST /send HTTP/1.1" 200 -
2024-11-19 12:47:08 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/1.1" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

2024-11-19 12:47:08 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-19 12:47:08 - Master - INFO - Queueing instruction to Node 2
2024-11-19 12:47:08 - Master - INFO - Node 2's queue length: 1
2024-11-19 12:47:08 - Master - INFO - Sending instruction to node 2

```

Después de enviarle varias instrucciones a los nodos, la cola del Nodo 0 tiene una longitud de 3 al momento en que se recibe el comando para terminar al Nodo 0:

```

2024-11-19 12:47:22 - Master - INFO - Queueing instruction to Node 0
2024-11-19 12:47:22 - Master - INFO - Node 0's queue length: 3
2024-11-19 12:47:24 - werkzeug - INFO - 127.0.0.1 - - [19/Nov/2024 12:47:24] "POST /stop/0 HTTP/1.1" 200 -
2024-11-19 12:47:24 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /stop/0 HTTP/1.1" 200 42
- Código de respuesta: 200
- Respuesta: {"message":"Node 0 stopped successfully"}

2024-11-19 12:47:27 - Node 2-32564 - DEBUG - Heartbeat sent
2024-11-19 12:47:27 - Node 1-28416 - DEBUG - Heartbeat sent
2024-11-19 12:47:32 - Node 2-32564 - DEBUG - Heartbeat sent
2024-11-19 12:47:32 - Node 1-28416 - DEBUG - Heartbeat sent
2024-11-19 12:47:32 - Master - INFO - Node 0 is not responding.
2024-11-19 12:47:32 - Master - INFO - Requeueing tasks from Node 0
2024-11-19 12:47:32 - Master - INFO - Node 0's queue length before requeueing: 3
2024-11-19 12:47:32 - Master - INFO - Finished requeueing for Node 0: length 0
2024-11-19 12:47:32 - Master - INFO - Node 0 activated.
2024-11-19 12:47:32 - Master - INFO - Queueing instruction to Node 1
2024-11-19 12:47:32 - Master - INFO - Node 1's queue length: 3
2024-11-19 12:47:32 - Master - INFO - Queueing instruction to Node 2
2024-11-19 12:47:32 - Master - INFO - Node 2's queue length: 3
2024-11-19 12:47:32 - Master - INFO - Queueing instruction to Node 0
2024-11-19 12:47:32 - Master - INFO - Node 0's queue length: 1
2024-11-19 12:47:32 - Master - INFO - Node 0 restarted.
2024-11-19 12:47:32 - Node 0-77412 - INFO - Node 0 created: 77412 with capacity of 5
2024-11-19 12:47:32 - Node 0-77412 - DEBUG - Heartbeat sent
2024-11-19 12:47:33 - Node 1-28416 - INFO - 102334155

```


Acá se puede visualizar donde se muestra que el primer nodo para de manera correcta. Finalmente, también se puede visualizar que, como el primer nodo terminó, el master no detectó el heartbeat por lo que inició el mecanismo de recuperación, donde se vuelven a encolar las tareas. Así, las tres tareas se reasignan a la cola maestra y el hilo de distribución de tareas las empieza a reasignar según el orden round-robin. En el proceso de levantamiento del nuevo proceso para el Nodo 0, se crea una cola nueva y se marca el nodo como activo. Así, se le empiezan a encolar tareas al Nodo 0 aunque todavía no se haya impreso el mensaje de que el Nodo 0 fue reiniciado. Esto no es problema, debido a que el nodo empieza a atender las solicitudes apenas esté disponible. Lo importante es que no se pierden tareas:

```
2024-11-19 12:47:32 - Node 0-77412 - INFO - Node 0 created: 77412 with capacity of 5
2024-11-19 12:47:32 - Node 0-77412 - DEBUG - Heartbeat sent
2024-11-19 12:47:33 - Node 1-28416 - INFO - 102334155
2024-11-19 12:47:33 - Master - INFO - Sending instruction to node 0
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Ejemplo 2

En este ejemplo, el nodo se detiene con un código de Python que termina la ejecución del programa. La configuración de la prueba es la siguiente:

```
system:
  node_quantity: 5
  node_capacities:
    - 5
    - 5
    - 5
    - 5
    - 5
  shared_resources:
    - name: shared_list
      type: list
      values:
        initial_value: [1, 2, 3, 4, 5]
    - name: shared_dict
      type: dict
      values:
        initial_value: { 1: 1, 2: 2, 3: 3, 4: 4, 5: 5 }
  requests:
    - endpoint: http://localhost:5000/send
      times: 6
      method: POST
      sleep: 2
      body:
        type: python
        command: "def fibonacci(n):\n\tif n == 0:\n\t\treturn 0\n\telif n == 1:\n\t\treturn 1\n\telse:\n\t\treturn fibonacci(n-1) + fibonacci(n-2)\nresponse = fibonacci(40)"
    - endpoint: http://localhost:5000/send
      times: 1
      method: POST
      sleep: 0
      body:
        type: python
        command: "if self.node_id == 1:\n\tprint(\"Me voy a suicidar\")\n\texit()\nelse:\n\tprint(\"Sobreviví\")"
```


Primero, se envían seis requisitos para calcular un Fibonacci complejo. Luego, se envía al Nodo 1 la instrucción de terminar. Esta valida si el nodo en la que se está ejecutando es el 1 y si es, el programa imprime que se va a terminar y se detiene la ejecución. Si no es, imprime que el nodo sobrevivió. Como sí se va a enviar al nodo 1, el nodo se va a morir.

```

2024-11-19 11:45:55 - Master - INFO - Sending instruction to node 1
if self.node_id == 1:
    print("Me voy a suicidar")
    exit()
else:
    print("Sobreviví ")
Me voy a suicidar
2024-11-19 11:45:58 - Node 2-69212 - INFO - 102334155
2024-11-19 11:45:58 - Master - INFO - Node 2 response: 102334155
2024-11-19 11:45:59 - Node 2-69212 - DEBUG - Heartbeat sent
2024-11-19 11:45:59 - Node 4-78612 - DEBUG - Heartbeat sent
2024-11-19 11:45:59 - Node 3-64156 - DEBUG - Heartbeat sent
2024-11-19 11:45:59 - Node 0-76656 - DEBUG - Heartbeat sent
2024-11-19 11:46:02 - Node 3-64156 - INFO - 102334155
2024-11-19 11:46:02 - Master - INFO - Node 3 response: 102334155
2024-11-19 11:46:04 - Node 2-69212 - DEBUG - Heartbeat sent
2024-11-19 11:46:04 - Node 3-64156 - DEBUG - Heartbeat sent
2024-11-19 11:46:04 - Node 4-78612 - DEBUG - Heartbeat sent
2024-11-19 11:46:04 - Node 0-76656 - DEBUG - Heartbeat sent
2024-11-19 11:46:04 - Master - INFO - Node 1 is not responding.
2024-11-19 11:46:04 - Master - INFO - Requeueing tasks from Node 1
2024-11-19 11:46:04 - Master - INFO - Finished requeueing for 1: length 0
2024-11-19 11:46:04 - Master - INFO - Queueing instruction to Node 2
2024-11-19 11:46:04 - Master - INFO - Status: True
2024-11-19 11:46:04 - Master - INFO - Node 2's queue length: 1
2024-11-19 11:46:04 - Master - INFO - Sending instruction to node 2
if self.node_id == 1:
    print("Me voy a suicidar")
    exit()
else:
    print("Sobreviví ")
Sobreviví
2024-11-19 11:46:04 - Node 2-69212 - INFO - No result returned.
2024-11-19 11:46:04 - Master - INFO - Node 1 restarted.
2024-11-19 11:46:05 - Node 4-78612 - INFO - 102334155
2024-11-19 11:46:05 - Master - INFO - Node 2 response: No result returned.
2024-11-19 11:46:05 - Master - INFO - Node 4 response: 102334155
2024-11-19 11:46:06 - Node 1-20340 - INFO - Node 1 created: 20340 with capacity of 5
2024-11-19 11:46:06 - Node 1-20340 - DEBUG - Heartbeat sent
2024-11-19 11:46:09 - Node 2-69212 - DEBUG - Heartbeat sent
2024-11-19 11:46:09 - Node 3-64156 - DEBUG - Heartbeat sent
2024-11-19 11:46:09 - Node 4-78612 - DEBUG - Heartbeat sent

```

Aquí podemos ver que luego de que se le enviara el código al Nodo 1, se imprimió “Me voy a suicidar”. Luego, se evidencia que el Nodo 1 no envía heartbeats y no está respondiendo, por lo que se mueven sus tareas al Nodo 2. En el Nodo 2, se vuelve a ejecutar el código que termina el proceso, pero como este no es el Nodo 1, imprime “Sobreviví” y el nodo no se muere. Posteriormente, se puede apreciar que el Nodo 1 se reinició con un nuevo proceso y con la capacidad correcta.

Ejemplo 3

En el siguiente ejemplo, se demuestra cómo se liberan los recursos a los cuales un nodo tenía acceso cuando se detecta una falla. En el programa que se le envía

como tarea, se realiza el cálculo de Fibonacci sobre el primer elemento de la lista compartida. Después, se instruye la terminación del Nodo 0:

```

1  system:
2      node_quantity: 5
3      node_capacities:
4          - 5
5          - 5
6          - 5
7          - 5
8          - 5
9      shared_resources:
10         - name: shared_list
11           type: List
12           values:
13             initial_value: [40, 2, 3, 4, 5]
14      requests:
15         - endpoint: http://localhost:5000/send
16           times: 1
17           method: POST
18           sleep: 3
19           body:
20             type: python
21             command: "def fibonacci(n):\n\tif n == 0:\n\t\treturn 0\n\telif n == 1:\n\t\treturn 1\n\telse:\n\t\treturn fibonacci(n-1) + fibonacci(n-2)\n\nshared_list = request_read_resource('shared_list')\nresponse = fibonacci(shared_list[0])\nrelease_read_resource('shared_list')"
```

En la ejecución, se le envía la instrucción al Nodo 0. Se aprecia que los 5 nodos envían su heartbeat:

```

2024-11-19 17:09:48 - Master - INFO - Queueing instruction to Node 0
2024-11-19 17:09:48 - Master - INFO - Node 0's queue length: 1
2024-11-19 17:09:48 - Master - INFO - Sending instruction to node 0
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
shared_list = request_read_resource('shared_list')
response = fibonacci(shared_list[0])
release_read_resource('shared_list')
2024-11-19 17:09:48 - Master - INFO - Node 0 has acquired read access to resource shared_list
2024-11-19 17:09:50 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-19 17:09:50 - Node 0-72484 - DEBUG - Heartbeat sent
2024-11-19 17:09:50 - Node 1-45460 - DEBUG - Heartbeat sent
2024-11-19 17:09:50 - Node 2-12300 - DEBUG - Heartbeat sent
2024-11-19 17:09:50 - Node 3-67836 - DEBUG - Heartbeat sent
2024-11-19 17:09:50 - Node 4-9624 - DEBUG - Heartbeat sent
```

Después de abortar el Nodo 0, ya no recibe el heartbeat y se inicia el mecanismo de recuperación del nodo. Se reasignan las tareas y se liberan los recursos compartidos.

```

2024-11-19 17:09:55 - Node 1-45460 - DEBUG - Heartbeat sent
2024-11-19 17:09:55 - Node 2-12300 - DEBUG - Heartbeat sent
2024-11-19 17:09:55 - Node 3-67836 - DEBUG - Heartbeat sent
2024-11-19 17:09:55 - Node 4-9624 - DEBUG - Heartbeat sent
2024-11-19 17:10:00 - Node 1-45460 - DEBUG - Heartbeat sent
2024-11-19 17:10:00 - Node 2-12300 - DEBUG - Heartbeat sent
2024-11-19 17:10:00 - Node 3-67836 - DEBUG - Heartbeat sent
2024-11-19 17:10:00 - Node 4-9624 - DEBUG - Heartbeat sent
2024-11-19 17:10:01 - Master - INFO - Node 0 is not responding.
2024-11-19 17:10:01 - Master - INFO - Requeueing tasks from Node 0
2024-11-19 17:10:01 - Master - INFO - Node 0's queue length before requeueing: 1
2024-11-19 17:10:01 - Master - INFO - Finished requeueing for Node 0: length 0
2024-11-19 17:10:01 - Master - INFO - Releasing resources held by Node 0
2024-11-19 17:10:01 - Master - INFO - Node 0 has released read access to resource shared_list
2024-11-19 17:10:01 - Master - INFO - Node 0 activated.
2024-11-19 17:10:01 - Master - INFO - Node 0 restarted.
2024-11-19 17:10:01 - Node 0-86864 - INFO - Node 0 created: 86864 with capacity of 5
2024-11-19 17:10:01 - Node 0-86864 - DEBUG - Heartbeat sent

```

La tarea se le asigna al Nodo 1:

```

2024-11-19 17:10:02 - Master - INFO - Queueing instruction to Node 1
2024-11-19 17:10:02 - Master - INFO - Node 1's queue length: 1
2024-11-19 17:10:02 - Master - INFO - Sending instruction to node 1
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
shared_list = request_read_resource('shared_list')
response = fibonacci(shared_list[0])
release_read_resource('shared_list')
2024-11-19 17:10:02 - Master - INFO - Node 1 has acquired read access to resource shared_list

```

El Nodo 1 eventualmente retorna la respuesta:

```

2024-11-19 17:10:15 - Node 4-9624 - DEBUG - Heartbeat sent
2024-11-19 17:10:16 - Node 0-86864 - DEBUG - Heartbeat sent
2024-11-19 17:10:17 - Master - INFO - Node 1 has released read access to resource shared_list
2024-11-19 17:10:17 - Node 1-45460 - INFO - 102334155
2024-11-19 17:10:18 - Master - INFO - Node 1 response: 102334155
2024-11-19 17:10:20 - Node 2-12300 - DEBUG - Heartbeat sent

```

Prueba 4: Escalabilidad del sistema

Esta prueba se encarga de verificar que el sistema siga funcionando correctamente cuando se le añaden nodos nuevos. La prueba la codificamos de la siguiente forma:

```

Prueba4.yaml
You, 2 hours ago | 1 author (You)
1 system:
2   node_quantity: 5
3   node_capacities:
4     - 5
5     - 5
6     - 5
7     - 5
8     - 5
9   shared_resources:
10    - name: shared_list
11      type: List
12      values:
13        initial_value: [1, 2, 3, 4, 5]
14    - name: shared_dict
15      type: Dict
16      values:
17        initial_value: { 1: 1, 2: 2, 3: 3, 4: 4, 5: 5 }
18   requests:
19    - endpoint: http://localhost:5000/send
20      times: 6
21      method: POST
22      sleep: 2
23      body:
24        type: python
25        command: "def fibonacci(n):\n\tif n == 0:\n\t\treturn 0\n\telif n == 1:\n\t\treturn 1\n\telse:\n\t\treturn fibonacci(n-1) + fibonacci(n-2)\n\nresponse = fibonacci(40)"
26    - endpoint: http://localhost:5000/add
27      times: 2
28      method: POST
29      sleep: 0
30      body:
31        max_capacity: 5
32    - endpoint: http://localhost:5000/send
33      times: 6
34      method: POST
35      sleep: 2
36      body:
37        type: python
38        command: "def fibonacci(n):\n\tif n == 0:\n\t\treturn 0\n\telif n == 1:\n\t\treturn 1\n\telse:\n\t\treturn fibonacci(n-1) + fibonacci(n-2)\n\nresponse = fibonacci(40)"

```

En esta prueba, tenemos 5 nodos con una capacidad de 5 procesos en la cola. Luego, se realizan 6 requests. Primero, se pone en la cola de cada nodo un cálculo de Fibonacci de 40 para que dure mucho la operación. Luego, se añaden 2 nodos con los requests de add. Estos tienen una capacidad de 5 procesos también en la cola. Finalmente, se añaden más operaciones de Fibonacci para que se envíen a los nuevos nodos.

```

2024-11-15 17:07:19 - werkzeug - INFO - 127.0.0.1 - - [15/Nov/2024 17:07:19] "POST /add HTTP/1.1" 200 -
2024-11-15 17:07:19 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /add HTTP/11" 200 40
- Código de respuesta: 200
- Respuesta: {"message": "Node 5 added successfully"}

2024-11-15 17:07:19 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-15 17:07:20 - Node 5-70732 - INFO - Node 5 created: 70732 with capacity of 5
2024-11-15 17:07:20 - Node 5-70732 - DEBUG - Heartbeat sent
2024-11-15 17:07:20 - Node 1-70408 - INFO - 102334155
2024-11-15 17:07:21 - werkzeug - INFO - 127.0.0.1 - - [15/Nov/2024 17:07:21] "POST /add HTTP/1.1" 200 -
2024-11-15 17:07:21 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /add HTTP/11" 200 40
- Código de respuesta: 200
- Respuesta: {"message": "Node 6 added successfully"}

2024-11-15 17:07:21 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-15 17:07:21 - Master - INFO - Node 1 response: 102334155
2024-11-15 17:07:22 - Node 6-77696 - INFO - Node 6 created: 77696 with capacity of 5

```

Aquí vemos que se crean los dos nuevos nodos adicionales correctamente.

```

2024-11-15 17:07:22 - Node 5-09080 - DEBUG - Heartbeat sent
2024-11-15 17:07:23 - werkzeug - INFO - 127.0.0.1 - - [15/Nov/2024 17:07:23] "POST /send HTTP/1.1" 200 -
2024-11-15 17:07:23 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/11" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

```

Aquí podemos ver que luego de añadir los nodos, el sistema sigue recibiendo solicitudes correctamente.

```

2024-11-15 17:07:39 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/11" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

2024-11-15 17:07:39 - Master - INFO - Queueing instruction to Node 5
2024-11-15 17:07:39 - Master - INFO - Status: True
2024-11-15 17:07:39 - Master - INFO - Node 5's queue length: 1
2024-11-15 17:07:39 - Master - INFO - Sending instruction to node 5
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
response = fibonacci(40)

```

Aquí podemos ver cómo se le asignan operaciones al nodo 5.

```

2024-11-15 17:07:43 - werkzeug - INFO - 127.0.0.1 - - [15/Nov/2024 17:07:43] "POST /send HTTP/1.1" 200 -
2024-11-15 17:07:43 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send HTTP/11" 200 35
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received"}

2024-11-15 17:07:43 - Master - INFO - Queueing instruction to Node 6
2024-11-15 17:07:43 - Master - INFO - Status: True
2024-11-15 17:07:43 - Master - INFO - Node 6's queue length: 1
2024-11-15 17:07:43 - Master - INFO - Sending instruction to node 6
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
response = fibonacci(40)

```

Aquí podemos ver cómo se le asignan operaciones al nodo 6.

```

2024-11-15 17:07:59 - Node 4-75004 - INFO - 102334155
2024-11-15 17:07:59 - Master - INFO - Node 4 response: 102334155
2024-11-15 17:08:00 - Node 5-70732 - DEBUG - Heartbeat sent
2024-11-15 17:08:02 - Node 6-77696 - DEBUG - Heartbeat sent
2024-11-15 17:08:02 - Node 0-77316 - DEBUG - Heartbeat sent
2024-11-15 17:08:02 - Node 1-70408 - DEBUG - Heartbeat sent
2024-11-15 17:08:02 - Node 2-78836 - DEBUG - Heartbeat sent
2024-11-15 17:08:02 - Node 4-75004 - DEBUG - Heartbeat sent
2024-11-15 17:08:02 - Node 3-69080 - DEBUG - Heartbeat sent
2024-11-15 17:08:03 - Node 5-70732 - INFO - 102334155
2024-11-15 17:08:03 - Master - INFO - Node 5 response: 102334155
2024-11-15 17:08:05 - Node 5-70732 - DEBUG - Heartbeat sent
2024-11-15 17:08:06 - Node 6-77696 - INFO - 102334155
2024-11-15 17:08:07 - Node 6-77696 - DEBUG - Heartbeat sent
2024-11-15 17:08:07 - Master - INFO - Node 6 response: 102334155
2024-11-15 17:08:07 - Node 0-77316 - DEBUG - Heartbeat sent

```

Aquí podemos ver cómo los nodos nuevos, el 5 y el 6, pueden recibir y ejecutar correctamente instrucciones.

Prueba 5: Redistribución automática de procesos

Esta prueba tiene como objetivo demostrar que el sistema redistribuye de manera automática los procesos cuando un nodo alcanza su capacidad máxima de carga. Para ello, se utiliza el siguiente archivo en formato YAML, diseñado para sobrecargar intencionalmente el primer nodo asignándole más tareas de las que puede manejar. En este escenario, se envían 7 tareas de larga duración al primer nodo, cuya capacidad máxima es de 5 tareas en cola, mientras que los demás nodos permanecen inactivos y no reciben ninguna carga inicial.

```

1  system:
2    node_quantity: 5
3    node_capacities:
4      - 5
5      - 5
6      - 5
7      - 5
8      - 5
9    shared_resources:
10     - name: shared_list
11       type: List
12       values:
13         initial_value: [1, 2, 3, 4, 5]
14     - name: shared_dict
15       type: Dict
16       values:
17         initial_value: { 1: 1, 2: 2, 3: 3, 4: 4, 5: 5 }
18   requests:
19     - endpoint: http://localhost:5000/send/0
20       times: 7
21       method: POST
22       sleep: 1
23       body:
24         type: python
25         command: "def fibonacci(n):\n\tif n == 0:\n\t\treturn 0\n\telif n == 1:\n\t\treturn 1\n\telse:\n\t\treturn fibonacci(n-1) + fibonacci(n-2)\n\nresponse = fibonacci(40)"

```

Estos son los resultados donde primero se puede visualizar cómo el primer nodo recibe todas las instrucciones respectivas

```
2024-11-15 17:04:55 - urllib3.connectionpool - DEBUG - http://localhost:5000 "POST /send/0 HTTP/1.1" 200 46
- Código de respuesta: 200
- Respuesta: {"message":"Instruction received for node 0"}

2024-11-15 17:04:56 - Master - INFO - Sending instruction to node 0
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
response = fibonacci(40)
2024-11-15 17:04:56 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
```

Después explica cómo el segundo nodo le quita trabajo al primer nodo y muestra la instrucción que toma.

```
2024-11-15 17:04:59 - Master - CRITICAL - Node 1 is stealing work from Node 0!
2024-11-15 17:04:59 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-15 17:05:00 - Master - INFO - Sending instruction to node 1
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
response = fibonacci(40)
```

Igualmente el tercer nodo hace lo mismo al quitarle trabajo al primer nodo donde muestra la instrucción que toma.

```
2024-11-15 17:05:02 - Master - CRITICAL - Node 2 is stealing work from Node 0!
2024-11-15 17:05:03 - urllib3.connectionpool - DEBUG - Starting new HTTP connection (1): localhost:5000
2024-11-15 17:05:03 - Master - INFO - Sending instruction to node 2
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
response = fibonacci(40)
```

Así sucesivamente con todas la tareas que este tiene hasta que finalmente empiezan a terminar donde se muestran los resultados de la siguiente manera

```
2024-11-15 17:05:18 - Node 1-38356 - INFO - 102334155
2024-11-15 17:05:19 - Master - INFO - Node 1 response: 102334155
```

```

2024-11-15 17:05:21 - Node 2-40272 - INFO - 102334155
2024-11-15 17:05:21 - Master - INFO - Node 2 response: 102334155
2024-11-15 17:05:22 - Node 2-40272 - DEBUG - Heartbeat sent
2024-11-15 17:05:22 - Node 4-33072 - DEBUG - Heartbeat sent
2024-11-15 17:05:22 - Node 3-40552 - DEBUG - Heartbeat sent
2024-11-15 17:05:22 - Node 1-38356 - DEBUG - Heartbeat sent
2024-11-15 17:05:22 - Node 0-40516 - DEBUG - Heartbeat sent
2024-11-15 17:05:24 - Node 3-40552 - INFO - 102334155
2024-11-15 17:05:25 - Master - INFO - Node 3 response: 102334155
2024-11-15 17:05:27 - Node 2-40272 - DEBUG - Heartbeat sent
2024-11-15 17:05:27 - Node 4-33072 - DEBUG - Heartbeat sent
2024-11-15 17:05:27 - Node 3-40552 - DEBUG - Heartbeat sent
2024-11-15 17:05:27 - Node 1-38356 - DEBUG - Heartbeat sent
2024-11-15 17:05:27 - Node 0-40516 - DEBUG - Heartbeat sent
2024-11-15 17:05:27 - Node 4-33072 - INFO - 102334155
2024-11-15 17:05:28 - Master - INFO - Node 4 response: 102334155
2024-11-15 17:05:32 - Node 2-40272 - DEBUG - Heartbeat sent
2024-11-15 17:05:32 - Node 3-40552 - DEBUG - Heartbeat sent
2024-11-15 17:05:32 - Node 4-33072 - DEBUG - Heartbeat sent
2024-11-15 17:05:32 - Node 1-38356 - DEBUG - Heartbeat sent
2024-11-15 17:05:32 - Node 0-40516 - DEBUG - Heartbeat sent
2024-11-15 17:05:32 - Node 0-40516 - INFO - 102334155
2024-11-15 17:05:32 - Master - INFO - Node 0 response: 102334155
2024-11-15 17:05:37 - Node 2-40272 - DEBUG - Heartbeat sent
2024-11-15 17:05:37 - Node 3-40552 - DEBUG - Heartbeat sent
2024-11-15 17:05:37 - Node 4-33072 - DEBUG - Heartbeat sent
2024-11-15 17:05:37 - Node 1-38356 - DEBUG - Heartbeat sent
2024-11-15 17:05:37 - Node 0-40516 - DEBUG - Heartbeat sent
2024-11-15 17:05:38 - Node 1-38356 - INFO - 102334155
2024-11-15 17:05:38 - Master - INFO - Node 1 response: 102334155

```

Esto funciona debido a que se implementó un algoritmo para “robar” el trabajo de los nodos. Cuando el master detecta que un nodo está ocioso, revisa las colas de los otros nodos para ver si estos tienen más de una tarea en su cola. Si esto es cierto, el master le quita la última tarea de la cola y se la pasa a la cola del nodo que está ocioso. Así, el nodo inactivo empieza a trabajar y se optimiza el uso de los recursos computacionales y se mejora el tiempo de respuesta. De esta manera, no se redistribuyen las hasta que un nodo alcance su capacidad máxima. Sino los nodos disponibles van agarrando las tareas en la cola de los otros nodos. Se logra que la mayoría de los nodos estén trabajando sin tener que realizar el cálculo de distribuir las tareas para que los nodos tengan colas de longitudes similares.


```

295
296         if queue:
297             instruction = queue[0]
298             self.logger.info(f"Sending instruction to node {node_id}")
299             self.node_pipes[node_id].send(instruction)
300         else:
301             # Steal work from other queues
302             for other_node, other_queue in self.node_queues.items():
303                 if other_node != node_id and len(other_queue) > 1:
304                     self.logger.critical(
305                         f"Node {node_id} is stealing work from Node {other_node}!"
306                     )
307                     instruction = other_queue.pop()
308                     self.logger.info(
309                         f"Node {other_node}'s queue length: {len(other_queue)}"
310                     )
311                     queue.append(instruction)
312                     break

```

Bibliografía

- Arredondo, D., Printista, A. M., & Gallard, R. H. (1995). Un sistema distribuido para el procesamiento paralelo de algoritmos genéticos. Congreso Argentino de Ciencias de la Computación. https://sedici.unlp.edu.ar/bitstream/handle/10915/24294/Documento_completo.pdf?sequence=1&isAllowed=y
- Jiménez, L. M., Puerto, R., & Payá, L. (2017). Sistemas distribuidos: Arquitectura y aplicaciones. Universidad Miguel Hernández.
- Python. (s.f.a). Multiprocessing — Paralelismo basado en procesos — Documentación de Python. <https://docs.python.org/es/3.9/library/multiprocessing.html>
- Python. (s.f.b). Threading — Thread-based parallelism. Python Documentation. Retrieved November 17, 2024, from <https://docs.python.org/3/library/threading.html>
- Python. (s.f.c). *collections* — *Container datatypes*. Python Documentation. Retrieved November 20, 2024, from <https://docs.python.org/3/library/collections.html#collections.deque>
- Sharma, S. (2023, May 24). *Unveiling the Power and Pitfalls of Master-Slave Architecture*. Medium. <https://medium.com/@cpsupriya31/understanding-master-slave-architecture-use-s-and-challenges-2acc907de7c4>

Silberschatz, A., Peterson, J. L., & Galvin, P. B. (1991). Operating system concepts.
Addison-Wesley Longman Publishing Co., Inc.