Algoritmos distribuidos

Los algoritmos distribuidos son un tipo específico de algoritmo diseñado para ejecutarse en sistemas distribuidos, es decir, en sistemas compuestos por múltiples computadoras interconectadas. Estos algoritmos se caracterizan por:

- Ejecución descentralizada: Cada computadora del sistema distribuido ejecuta una parte del algoritmo de forma independiente, sin necesidad de un control centralizado.
- Comunicación y coordinación: Las computadoras del sistema distribuido se comunican entre sí para intercambiar información y coordinar sus acciones, asegurando que el algoritmo se ejecute de manera coherente y eficiente.
- Tolerancia a fallos: Los algoritmos distribuidos deben ser tolerantes a fallos, ya que es posible que algunas computadoras del sistema fallen durante la ejecución del algoritmo. Esto significa que el algoritmo debe ser capaz de continuar su ejecución incluso si algunas computadoras no están disponibles.
- Escalabilidad: Los algoritmos distribuidos deben ser escalables, es decir, deben poder ejecutarse de manera eficiente en sistemas con un gran número de computadoras.

Los algoritmos distribuidos se utilizan en una amplia variedad de aplicaciones, incluyendo:

- Redes de computadoras: Enrutamiento de datos, control de acceso, gestión de recursos.
- Sistemas de almacenamiento distribuido: Bases de datos distribuidas, sistemas de archivos distribuidos.
- Computación en la nube: Distribución de cargas de trabajo, escalabilidad de servicios.
- Internet de las cosas: Recolección y procesamiento de datos de dispositivos distribuidos.
- Inteligencia artificial distribuida: Aprendizaje automático distribuido, procesamiento del lenguaje natural distribuido.

Existen diferentes tipos de algoritmos distribuidos, cada uno con sus propias características y aplicaciones específicas.

Algoritmos de consenso

- Algoritmo de Paxos
- Raft

Algoritmos de elección de líder

- Algoritmo de Bully
- Tolerancia a fallos bizantina (Byzantine fault tolerance)
- Detección de terminación de procesos
  - Algoritmo de Dijkstra-Scholten

  - Algoritmo de Huang

Algoritmos de relojes lógicos

- Sincronización de relojes
- Algoritmo de Berkeley
- Algoritmo de Cristian
- Algoritmo de Intersección
- Algoritmo de Marzullo
- Ordenamiento de Lamport
- Relojes vectoriales (Vector clocks)

Algoritmos de distribución de recursos

- Exclusión mutua
- Algoritmo de exclusión mutua distribuida de Lamport
- Algoritmo log(n) de Naimi-Trehel
- Algoritmo de Maekawa
- Algoritmo de Raymond
- Algoritmo de Ricart-Agrawala

Algoritmo de toma de instantáneas

- Algoritmos de toma de instantáneas
- Algoritmo de Chandy-Lamport

Algoritmos de gestión de memoria

- Algoritmos de asignación y liberación de memoria
- Asignación de memoria Buddy
- Recolectores de basura
  - Algoritmo de Cheney
  - Recolector de basura generacional
  - Algoritmo de marcado-compacto (Mark-compact algorithm)
  - Marcado y barrido (Mark and sweep)
  - Recolector de semiespacio

Algoritmos de consenso

En sistemas distribuidos, lograr el consenso entre múltiples nodos es fundamental para asegurar la consistencia y la confiabilidad de los datos. Dos de los algoritmos más reconocidos para resolver problemas de consenso en redes de procesadores poco fiables son el Algoritmo de Paxos y Raft. Ambos algoritmos abordan el desafío del consenso de maneras diferentes, pero con el mismo objetivo de asegurar que todos los nodos en un sistema distribuido acuerden un mismo valor. Este informe proporciona una visión detallada de estos algoritmos, incluyendo sus principios, funcionamiento y ejemplos de implementaciones en Python.

Algoritmo de Paxos

El Algoritmo de Paxos, desarrollado por Leslie Lamport, es una familia de protocolos diseñados para resolver problemas de consenso en redes de procesadores poco fiables. Paxos asegura que un conjunto de nodos puede llegar a un acuerdo sobre un único valor, incluso en presencia de fallos en algunos nodos.

Los principales componentes del Algoritmo de Paxos son:

- Proposers (Proponentes): Nodos que proponen valores a ser consensuados.
- Acceptors (Aceptantes): Nodos que reciben las propuestas y deciden si aceptarlas.
- Learners (Aprendices): Nodos que aprenden el valor consensuado una vez que se ha decidido.

El protocolo de Paxos se puede dividir en varias fases:

1. Fase de Preparación (Prepare Phase):
- El proponente elige un número de propuesta (proposal number) único y envía una solicitud de preparación (prepare request) a una mayoría de aceptantes.
- Cada aceptante compara el número de la propuesta con los números de propuestas anteriores y responde con la promesa de no aceptar propuestas con números menores, además de enviar la última propuesta aceptada (si existe).
1. Fase de promesa (Promise Phase):
- Si el proponente recibe promesas de una mayoría de aceptantes, envía una propuesta (proposal) con el valor más alto de las últimas propuestas aceptadas que ha recibido.
- Los aceptantes que reciben la propuesta comparan el número de la propuesta con sus promesas previas y, si es válido, aceptan la propuesta y notifican al proponente.
1. Fase de aceptación (Accept Phase):
- Si el proponente recibe respuestas de aceptación de una mayoría de aceptantes, se considera que la propuesta ha sido aceptada, y los aprendices son notificados del valor consensuado.

Implementación del algoritmo de Paxos

Aquí hay un ejemplo simplificado de implementación del Algoritmo de Paxos en Python:

```
class Proposer:
    def __init__(self, proposer_id, acceptors):
        self.proposer_id = proposer_id
        self.acceptors = acceptors
        self.proposal_number = 0
        self.proposal_value = None

    def prepare(self):
        self.proposal_number += 1
        promises = []
        for acceptor in self.acceptors:
            promise = acceptor.receive_prepare(self.proposal_number)
            if promise:
                promises.append(promise)
        if len(promises) > len(self.acceptors) / 2:
```

```
            self.propose()

    def propose(self):
        for acceptor in self.acceptors:
            acceptor.receive_propose(self.proposal_number,
self.proposal_value)

class Acceptor:
    def __init__(self, acceptor_id):
        self.acceptor_id = acceptor_id
        self.promised_proposal_number = 0
        self.accepted_proposal_number = 0
        self.accepted_value = None

    def receive_prepare(self, proposal_number):
        if proposal_number > self.promised_proposal_number:
            self.promised_proposal_number = proposal_number
            return (self.accepted_proposal_number,
self.accepted_value)
        return None

    def receive_propose(self, proposal_number, value):
        if proposal_number >= self.promised_proposal_number:
            self.promised_proposal_number = proposal_number
            self.accepted_proposal_number = proposal_number
            self.accepted_value = value
            return True
        return False

# Ejemplo de uso
acceptors = [Acceptor(i) for i in range(5)]
proposer = Proposer(1, acceptors)
proposer.prepare()
```

Este código demuestra los componentes básicos del Algoritmo de Paxos, incluyendo la preparación y la proposición de valores.

Algoritmo Raft

Raft es un algoritmo de consenso diseñado por Diego Ongaro y John Ousterhout como una alternativa a Paxos, con el objetivo de ser más comprensible y fácil de implementar. Raft divide el problema de consenso en tres subproblemas principales:

- Elección de líder: Un nodo es elegido como líder por un periodo de tiempo (termo) y es responsable de gestionar las entradas de log.
- Replicación de log: El líder replica las entradas de log a los seguidores para mantener un estado consistente.
- Compromiso de log: Las entradas de log se consideran comprometidas una vez que se han replicado en una mayoría de nodos.

Los estados de los nodos en Raft son:

1. Líder: El nodo que gestiona las entradas de log y coordina la replicación.
2. Seguidor: Nodo que replica las entradas de log del líder.
3. Candidato: Nodo que se postula para ser líder durante una elección.

El proceso de elección de líder en Raft incluye:

1. Solicitar votos: Un candidato incrementa su término y envía solicitudes de votos a otros nodos.
2. Votación: Los nodos responden a las solicitudes de voto. Si un candidato recibe votos de una mayoría de nodos, se convierte en líder.
3. Periodo de liderazgo: El líder envía latidos (heartbeats) periódicos para mantener su liderazgo y replicar nuevas entradas de log.

Implementación del algoritmo Raft

A continuación, se presenta un ejemplo simplificado de implementación del Algoritmo Raft en Python:

```
import threading
import time

class Node:
    def __init__(self, node_id, peers):
        self.node_id = node_id
        self.peers = peers
        self.term = 0
        self.voted_for = None
        self.state = "follower"
        self.log = []
        self.commit_index = 0
        self.last_applied = 0

    def start_election(self):
        self.state = "candidate"
        self.term += 1
        self.voted_for = self.node_id
        votes = 1
        for peer in self.peers:
            if peer.request_vote(self.term, self.node_id):
                votes += 1
        if votes > len(self.peers) / 2:
            self.state = "leader"
            self.lead()

    def request_vote(self, term, candidate_id):
        if term > self.term:
            self.term = term
            self.voted_for = candidate_id
```

```
            return True
        return False

    def lead(self):
        while self.state == "leader":
            self.send_heartbeats()
            time.sleep(1)

    def send_heartbeats(self):
        for peer in self.peers:
            peer.receive_heartbeat(self.term, self.node_id)

    def receive_heartbeat(self, term, leader_id):
        if term >= self.term:
            self.term = term
            self.state = "follower"

# Ejemplo de uso
nodes = [Node(i, []) for i in range(5)]
for node in nodes:
    node.peers = [n for n in nodes if n != node]

leader = nodes[0]
thread = threading.Thread(target=leader.start_election)
thread.start()
```

Este código muestra los conceptos básicos de la elección de líder y el envío de latidos en el algoritmo Raft.

Los algoritmos de Paxos y Raft son soluciones robustas y probadas para el problema del consenso en sistemas distribuidos. Paxos, a pesar de su complejidad, ofrece una alta tolerancia a fallos, mientras que Raft proporciona una alternativa más fácil de entender y de implementar.

Ambos algoritmos juegan un papel crucial en el mantenimiento de la consistencia y la confiabilidad en sistemas distribuidos, y sus implementaciones en Python pueden ser utilizadas como base para desarrollar sistemas distribuidos resilientes y eficientes.

Algoritmos de elección de líder

En sistemas distribuidos, la elección de un líder es crucial para coordinar tareas, gestionar recursos y mantener la consistencia del sistema. Los algoritmos de elección de líder permiten a los nodos de un sistema distribuido ponerse de acuerdo sobre quién debería ser el coordinador o líder, incluso en presencia de fallos.

Algoritmo de Bully

El Algoritmo de Bully es un método para seleccionar dinámicamente un coordinador en un sistema distribuido. Funciona bajo el supuesto de que cada nodo tiene un identificador único y que los nodos pueden fallar, pero eventualmente se recuperan.

El proceso del algoritmo de Bully es el siguiente:

1. Cuando un nodo detecta que el coordinador ha fallado, inicia una elección enviando mensajes de elección a todos los nodos con identificadores más altos.
2. Si ningún nodo responde, el nodo iniciador se declara líder.
3. Si uno o más nodos responden, estos nodos inician sus propias elecciones.
4. El proceso continúa hasta que un nodo con el identificador más alto se declara líder.

Implementación del algoritmo de Bully

```
class Node:
    def __init__(self, node_id, nodes):
        self.node_id = node_id
        self.nodes = nodes
        self.leader = None

    def start_election(self):
        print(f"Nodo {self.node_id} inicia una eleccion.")
        higher_nodes = [node for node in self.nodes if node.node_id >
self.node_id]
        if not higher_nodes:
            self.become_leader()
        else:
            for node in higher_nodes:
                if node.alive:
                    node.receive_election_message(self.node_id)

    def receive_election_message(self, sender_id):
        print(f"Nodo {self.node_id} recive mensaje de eleccion desde
el Nodo {sender_id}.")
        if self.node_id > sender_id:
            self.start_election()

    def become_leader(self):
        print(f"Nodo {self.node_id} llega a ser el lider the leader.")
        self.leader = self.node_id
        for node in self.nodes:
            if node.node_id != self.node_id:
                node.leader = self.node_id

    @property
    def alive(self):
        return True

nodes = [Node(i, []) for i in range(5)]
for node in nodes:
    node.nodes = [n for n in nodes if n != node]

nodes[2].start_election()
```

Este código demuestra cómo los nodos se comunican entre sí para elegir un líder utilizando el Algoritmo de Bully.

Algoritmo de Chang y Roberts

El algoritmo de Chang y Roberts es un algoritmo de elección de líderes diseñado para sistemas distribuidos donde los nodos están organizados en un anillo lógico. Este algoritmo es especialmente adecuado para entornos donde la estructura del anillo es natural o fácil de implementar. A continuación, se explica en detalle su funcionamiento, características, ventajas y limitaciones.

Descripción del algoritmo

El algoritmo de Chang y Roberts funciona de la siguiente manera:

1. Inicialización:
   - Supongamos que cada nodo en el sistema tiene un identificador único.
   - Los nodos están conectados en un anillo lógico, lo que significa que cada nodo puede comunicarse directamente solo con su vecino inmediato.
2. Detección del fallo del líder:
   - Cuando un nodo detecta que el líder actual ha fallado (por ejemplo, no responde a mensajes), inicia una elección.
3. Inicio de la elección:
   - El nodo que detecta el fallo del líder crea un mensaje de elección que contiene su propio identificador.
   - Este mensaje de elección se envía al siguiente nodo en el anillo.
4. Propagación del mensaje de elección:
   - Cada nodo que recibe un mensaje de elección compara el identificador en el mensaje con su propio identificador.
   - Si el identificador en el mensaje es mayor que su propio identificador, el nodo reenvía el mensaje al siguiente nodo en el anillo.
   - Si el identificador en el mensaje es menor que su propio identificador, el nodo reemplaza el identificador en el mensaje con su propio identificador y reenvía el mensaje al siguiente nodo.
   - Si el identificador en el mensaje es igual al identificador del nodo que inició la elección, el proceso de elección ha finalizado y el nodo con el identificador mayor ha sido elegido como líder.
5. Declaración del líder:
   - El nodo que detecta que su propio identificador ha regresado en el mensaje de elección reconoce que ha sido elegido como líder.
   - Este nodo ahora envía un mensaje de liderazgo alrededor del anillo para informar a todos los demás nodos de la identidad del nuevo líder.

Ventajas del algoritmo de Chang y Roberts

Simplicidad:

- El algoritmo es simple de implementar y entender, lo que facilita su adopción en sistemas distribuidos con topología de anillo.

Eficiencia en mensajes:

- El algoritmo minimiza el número de mensajes necesarios para realizar la elección, ya que cada mensaje de elección se reenvía un máximo de una vez por cada nodo en el anillo.

Determinismo:

- El algoritmo garantiza que el nodo con el identificador más alto siempre será elegido como líder, asegurando una elección determinista y justa.

Limitaciones del algoritmo de Chang y Roberts

Latencia:

- En anillos grandes, la latencia de la elección puede ser significativa, ya que el mensaje de elección debe recorrer potencialmente todo el anillo antes de que se elija un líder.

Fallos de nodos intermedios:

- Si un nodo intermedio falla durante el proceso de elección, el algoritmo puede necesitar reiniciarse, lo que puede introducir retrasos adicionales.

Topología fija:

- El algoritmo asume una topología de anillo fija y no es fácilmente adaptable a otras topologías de red.

Ejemplo de funcionamiento

Consideremos un sistema distribuido con cuatro nodos (A, B, C, D) con identificadores 1, 2, 3, y 4, respectivamente, organizados en un anillo:

```
A (1) -> B (2) -> C (3) -> D (4) -> A (1)
```

Supongamos que el nodo C detecta que el líder actual ha fallado:

1. Nodo C inicia la elección:
   - C envía un mensaje de elección con su identificador (3) a D.
2. Nodo D recibe el mensaje:
   - D compara su identificador (4) con el del mensaje (3).
   - D reemplaza el identificador en el mensaje con su propio identificador (4) y lo envía a A.
3. Nodo A recibe el mensaje:
   - A compara su identificador (1) con el del mensaje (4).
   - A reenvía el mensaje sin cambios a B.
4. Nodo B recibe el mensaje:

- B compara su identificador (2) con el del mensaje (4).
- B reenvía el mensaje sin cambios a C.

5. Nodo C recibe el mensaje:
   - C detecta que el identificador en el mensaje es mayor que el suyo, reconociendo que D (4) ha sido elegido como líder.
   - C envía un mensaje de liderazgo alrededor del anillo informando a todos los nodos que D es el nuevo líder.

Este proceso asegura que todos los nodos conocen al nuevo líder y que la elección se lleva a cabo de manera eficiente y correcta.

Implementación del algoritmo Chang y Roberts

```
import threading
import time
import random

class Node(threading.Thread):
    def __init__(self, node_id, next_node):
        super().__init__()
        self.node_id = node_id
        self.next_node = next_node
        self.leader = None
        self.initiator = False

    def run(self):
        if self.initiator:
            self.start_election()

    def start_election(self):
        print(f"Nodo {self.node_id} inicia la eleccion.")
        self.send_election_message(self.node_id)

    def send_election_message(self, election_id):
        print(f"Nodo {self.node_id} envia el mensaje de eleccion con
ID {election_id}.")
        self.next_node.receive_election_message(election_id)

    def receive_election_message(self, election_id):
        if election_id > self.node_id:
            self.send_election_message(election_id)
        elif election_id < self.node_id:
            self.send_election_message(self.node_id)
        elif election_id == self.node_id:
            print(f"Nodo {self.node_id} ha sido elegido como lider.")
            self.leader = self.node_id
            self.send_leader_message(self.node_id)

    def send_leader_message(self, leader_id):
```

```
        print(f"Nodo {self.node_id} envia el mensaje con lider con
lider ID {leader_id}.")
        self.next_node.receive_leader_message(leader_id)

    def receive_leader_message(self, leader_id):
        if self.leader is None:
            self.leader = leader_id
            self.send_leader_message(leader_id)
        print(f"Node {self.node_id} reconoce el lider {leader_id}.")

def create_ring(nodes):
    for i in range(len(nodes)):
        nodes[i].next_node = nodes[(i + 1) % len(nodes)]

if __name__ == "__main__":
    node_count = 5
    nodes = [Node(i, None) for i in range(node_count)]

    create_ring(nodes)

    # Randomly select a node to start the election
    initiator = random.choice(nodes)
    initiator.initiator = True

    # Start all nodes
    for node in nodes:
        node.start()

    # Wait for all nodes to complete
    for node in nodes:
        node.join()

    print("Eleccion completada.")
```

Tolerancia a fallos Bizantino

La tolerancia a fallos Bizantina se refiere a la capacidad de un sistema distribuido para resistir fallos arbitrarios, incluidos los comportamientos maliciosos. Este tipo de tolerancia a fallos es crucial en sistemas donde la confiabilidad y la seguridad son esenciales.

Un enfoque común para lograr la tolerancia a fallos bizantina es el uso de algoritmos de consenso bizantino, como el algoritmo de acuerdo Bizantino (Byzantine Agreement Algorithm), que permite que los nodos honestos en un sistema lleguen a un consenso incluso si algunos nodos se comportan de manera arbitraria.

Para alcanzar consenso en presencia de fallos bizantinos, se pueden usar varios algoritmos. A continuación, se describe uno de los más conocidos: el Algoritmo de Lamport, Shostak y Pease.

Algoritmo de Lamport, Shostak y Pease

Este algoritmo requiere que al menos dos tercios de los nodos sean leales (es decir, menos de un tercio de los nodos pueden ser bizantinos) para garantizar un consenso correcto. El algoritmo se desarrolla en varias rondas y cada nodo recopila información de otros nodos para determinar el valor final.

1. Ronda 0:
   - Cada nodo envía su valor inicial a todos los demás nodos.
2. Rondas posteriores:
   - Cada nodo recopila los valores recibidos de todos los demás nodos.
   - Cada nodo envía los valores recopilados en la ronda anterior a todos los demás nodos.
   - Este proceso se repite durante un número de rondas igual al número de nodos.
3. Determinación del valor final:
   - Después de un número suficiente de rondas, cada nodo utiliza un procedimiento de votación o consenso sobre los valores recibidos para decidir el valor final.

Ejemplo

Consideremos un sistema con cuatro nodos A,B,C,D, donde D es un nodo bizantino. El objetivo es que los nodos leales A,B,C alcancen un acuerdo común.

1. Ronda 0:
   - A envía su valor V_A a B,C, D.
   - B envía su valor V_B a A,C, D.
   - C envía su valor V_B a A,C, D
   - D envía valores (que pueden ser falsos) a A,B,C.
2. Rondas posteriores:
   - Cada nodo recopila los valores recibidos y envía esta información a todos los demás nodos.
   - Los nodos leales recopilan los valores y los envían en la siguiente ronda.
3. Determinación del valor final:
   - Después de suficientes rondas, los nodos leales determinan el valor final mediante un procedimiento de votación. Los nodos leales pueden identificar los valores inconsistentes enviados por el nodo bizantino y acordar un valor común basado en la mayoría de los valores leales.

Ventajas y desafíos

Ventajas:

1. Robustez:
   - El algoritmo puede tolerar nodos que se comportan de manera arbitraria o maliciosa.

2. Consenso garantizado:
    - Garantiza que los nodos leales alcanzarán un acuerdo común, siempre que menos de un tercio de los nodos sean bizantinos.

Desafíos:

1. Complejidad de comunicación:
    - Requiere un alto número de mensajes intercambiados entre los nodos, lo que puede ser costoso en términos de ancho de banda y latencia.
2. Limitación en el número de fallos:
    - Solo puede tolerar fallos bizantinos si el número de nodos bizantinos es menor que un tercio del total de nodos.

Implementación de un algoritmo de tolerancia a fallos Bizantinos

A continuación se muestra un ejemplo simplificado de un algoritmo de acuerdo Bizantino:

```
class ByzantineNode:
    def __init__(self, node_id, nodes, traitor=False):
        self.node_id = node_id
        self.nodes = nodes
        self.traitor = traitor

    def send_message(self, message, recipient):
        if self.traitor:
            message = "malicious"
        recipient.receive_message(message, self.node_id)

    def receive_message(self, message, sender_id):
        print(f"Nodo {self.node_id} recibe el mensaje '{message}'
desde el nodo {sender_id}.")

nodes = [ByzantineNode(i, []) for i in range(5)]
nodes[2].traitor = True  # Node 2 is a traitor
for node in nodes:
    node.nodes = [n for n in nodes if n != node]

nodes[0].send_message("hello", nodes[1])
```

Este ejemplo ilustra cómo los nodos pueden comunicarse entre sí, y cómo un nodo traidor puede alterar el mensaje.

Detección de terminación de procesos

La detección de terminación de procesos es un problema en sistemas distribuidos donde los nodos deben determinar cuándo todos los procesos en el sistema han terminado. Este problema es fundamental para coordinar tareas distribuidas y liberar recursos.

Uno de los algoritmos clásicos para la detección de terminación es el Algoritmo de Dijkstra-Scholten, que permite que un proceso inicie la detección de terminación y asegure que todos los procesos han terminado antes de declarar la terminación global.

Algoritmo de Dijkstra-Scholten

El algoritmo de Dijkstra-Scholten se utiliza para la detección de terminación en sistemas distribuidos. Funciona rastreando las dependencias de mensajes entre procesos y asegurando que todos los procesos han terminado antes de declarar la terminación global.

El algoritmo puede describirse de la siguiente manera:

1. Un proceso inicia la detección de terminación enviando mensajes a sus vecinos.
2. Cada proceso que recibe un mensaje marca al remitente como su "padre" en un árbol de dependencia.
3. Cuando un proceso no tiene mensajes pendientes ni hijos activos, envía un mensaje de terminación a su padre.
4. La raíz del árbol (el proceso iniciador) declara la terminación global cuando todos los procesos han confirmado la terminación.

Implementación del algoritmo de Dijkstra-Scholten

A continuación se muestra un ejemplo simplificado del Algoritmo de Dijkstra-Scholten en Python:

```
class Process:
    def __init__(self, process_id, neighbors):
        self.process_id = process_id
        self.neighbors = neighbors
        self.parent = None
        self.children = set()
        self.active = True

    def send_message(self, recipient):
        recipient.receive_message(self, self.process_id)

    def receive_message(self, sender, sender_id):
        if self.parent is None:
            self.parent = sender
        self.children.add(sender_id)
        self.process_task()

    def process_task(self):
        # Simulate task processing
        self.active = False
        self.check_termination()

    def check_termination(self):
        if not self.active and not self.children:
            if self.parent:
                self.parent.receive_termination(self.process_id)
```

```
    def receive_termination(self, child_id):
        self.children.remove(child_id)
        self.check_termination()

# Ejemplo de uso
processes = [Process(i, []) for i in range(5)]
for process in processes:
    process.neighbors = [p for p in processes if p != process]

# Iniciar la detección de terminación
initiator = processes[0]
initiator.send_message(processes[1])
```

Este ejemplo muestra cómo los procesos se comunican y determinan la terminación utilizando el Algoritmo de Dijkstra-Scholten.

Algoritmo de Huang

El algoritmo de Huang es un método para la detección de terminación en sistemas distribuidos. Este algoritmo utiliza un token para rastrear la actividad de los procesos y determinar cuándo todos los procesos han terminado.

El funcionamiento del Algoritmo de Huang es el siguiente:

1. Un proceso inicia la detección de terminación generando y enviando un token.
2. Cada proceso que recibe el token verifica su estado y el de sus vecinos.
3. Si un proceso no tiene actividad, marca el token como inactivo y lo pasa al siguiente proceso.
4. La detección de terminación se declara cuando el token regresa al proceso iniciador sin actividad pendiente.

Implementación del algoritmo de Huang

Aquí se presenta una implementación simplificada del Algoritmo de Huang en Python:

```
class HuangProcess:
    def __init__(self, process_id, neighbors):
        self.process_id = process_id
        self.neighbors = neighbors
        self.active = True

    def send_token(self, token, recipient):
        recipient.receive_token(token, self.process_id)

    def receive_token(self, token, sender_id):
        if not self.active:
            token["inactive"] += 1

        # Determine if all processes are inactive
```

```
        if token["inactive"] >= len(self.neighbors) + 1:  # Include
self in the count
            print(f"Terminación detectada por Proceso
{self.process_id}")
        else:
            sender_index = next(i for i, p in
enumerate(self.neighbors) if p.process_id == sender_id)
            next_index = (sender_index + 1) % len(self.neighbors)
            next_neighbor = self.neighbors[next_index]
            self.send_token(token, next_neighbor)

    def process_task(self):
        # Simulate task processing
        self.active = False

# Create processes and set neighbors
processes = [HuangProcess(i, []) for i in range(5)]
for process in processes:
    process.neighbors = [p for p in processes if p != process]

# Simulate task processing
for process in processes:
    process.process_task()

# Initiate termination detection
initiator = processes[0]
initiator.send_token({"inactive": 0}, processes[1])
```

Este ejemplo demuestra cómo los procesos utilizan un token para rastrear la actividad y determinar la terminación utilizando el Algoritmo de Huang.

Los algoritmos de elección de líder y detección de terminación son fundamentales para la coordinación y gestión de recursos en sistemas distribuidos. El algoritmo de Bully, la tolerancia a fallos Bizantino, el algoritmo de Dijkstra-Scholten y el algoritmo de Huang son métodos efectivos para abordar estos problemas.

Las implementaciones en Python proporcionan una base práctica para comprender y aplicar estos algoritmos en sistemas distribuidos reales, asegurando la consistencia y la confiabilidad en presencia de fallos y actividad concurrente.

Algoritmos de relojes lógicos

En sistemas distribuidos, la sincronización de eventos y la consistencia del tiempo son cruciales para el correcto funcionamiento y coordinación entre nodos. Los relojes lógicos y físicos son herramientas fundamentales para lograr esta sincronización.

Sincronización de relojes

La sincronización de relojes es el proceso de coordinar el tiempo entre los nodos de un sistema distribuido. Este proceso es esencial para asegurar que los eventos ocurran en el orden correcto y para mantener la consistencia temporal en todo el sistema.

Algoritmo de Berkeley

El algoritmo de Berkeley es un método centralizado para sincronizar relojes en un sistema distribuido. Se basa en un nodo maestro que coordina la sincronización del reloj entre todos los nodos del sistema. A continuación, se describen los pasos del algoritmo:

1. Selección del nodo maestro:
   - Un nodo es elegido como el maestro. Este nodo será responsable de coordinar la sincronización del reloj entre todos los nodos, incluyendo a sí mismo.
2. Recolección de tiempos:
   - El nodo maestro envía una solicitud a todos los demás nodos en el sistema, pidiéndoles que reporten sus tiempos actuales.
   - Cada nodo responde con su tiempo actual.
3. Cálculo del tiempo promedio:
   - El nodo maestro recopila todas las respuestas y calcula el tiempo promedio. Para esto, primero ajusta los tiempos recibidos teniendo en cuenta el retraso de la red (latencia) basado en el tiempo de ida y vuelta (round-trip time) de los mensajes.
   - El tiempo promedio se calcula como la media aritmética de los tiempos ajustados.
4. Determinación de los ajustes:
   - El nodo maestro determina los ajustes necesarios para cada nodo, calculando la diferencia entre el tiempo promedio y el tiempo reportado por cada nodo.
5. Distribución de los ajustes:
   - El nodo maestro envía a cada nodo la cantidad de ajuste que deben aplicar a sus relojes.
6. Ajuste de los relojes:
   - Cada nodo ajusta su reloj local según la instrucción recibida del nodo maestro.

Ejemplo

Supongamos que tenemos un sistema distribuido con cuatro nodos: A, B, C y D. El nodo $A4 es seleccionado como el nodo maestro.

1. Recolección de tiempos:
   - Nodo A solicita el tiempo a B, C y D.
   - Supongamos que los tiempos reportados son:
     - B: 10:00:10
     - C: 10:00:15
     - D: 10:00:20
2. Cálculo del tiempo promedio:
   - Nodo A recoge estos tiempos y calcula el tiempo promedio.
   - Supongamos que los tiempos ajustados después de considerar la latencia son:

        - B: 10:00:11
        - C: 10:00:14
        - D: 10:00:19
    - El tiempo promedio sería: $ = \frac{10:00:11+ 10:00:19 +10:00:12}{4} =10:00:14$
3. Determinación de los ajustes:
    - Nodo `A` calcula los ajustes necesarios:
        - `B` debe ajustar su reloj en +3 segundos.
        - `C` debe ajustar su reloj en 0 segundos.
        - `D` debe ajustar su reloj en -5 segundos.
4. Distribución de los ajustes:
    - Nodo A envía estos ajustes a `B`, `C` y `D`.
5. Ajuste de los relojes:
    - Cada nodo ajusta su reloj según las instrucciones recibidas:
        - B ajusta su reloj a 10:00:14.
        - C no necesita ajuste.
        - D ajusta su reloj a 10:00:14.

Ventajas del algoritmo de Berkeley

Descentralización:

- No requiere un servidor de tiempo externo, lo que es útil en entornos donde no hay acceso a un servidor de tiempo confiable.

Precisión:

- Ajusta los relojes de todos los nodos de acuerdo con un tiempo promedio, lo que puede ser más preciso en algunos entornos donde todos los nodos tienen tiempos locales similares.

Flexibilidad:

- El nodo maestro puede cambiar dinámicamente, lo que permite una mayor resiliencia en caso de fallos del nodo maestro.

Desafíos del algoritmo de Berkeley

Dependencia del nodo maestro:

- La precisión de la sincronización depende de la correcta operación del nodo maestro. Si el nodo maestro falla, la sincronización puede verse afectada hasta que se seleccione un nuevo maestro.

Latencia de la red:

- La precisión del ajuste depende de una correcta estimación de la latencia de la red. Fluctuaciones en la latencia pueden afectar la precisión del tiempo sincronizado.

Escalabilidad:

- En sistemas con un gran número de nodos, el nodo maestro puede convertirse en un cuello de botella debido a la gran cantidad de mensajes que debe procesar y enviar.

Implementación del algoritmo de Berkeley

```
class BerkeleyNode:
    def __init__(self, node_id, time):
        self.node_id = node_id
        self.time = time

    def adjust_time(self, offset):
        self.time += offset

class BerkeleyMaster:
    def __init__(self, nodes):
        self.nodes = nodes

    def synchronize_clocks(self):
        times = [node.time for node in self.nodes]
        average_time = sum(times) / len(times)
        for node in self.nodes:
            offset = average_time - node.time
            node.adjust_time(offset)
        return [(node.node_id, node.time) for node in self.nodes]

# Ejemplo de uso
nodes = [BerkeleyNode(i, time) for i, time in enumerate([10, 20, 30])]
master = BerkeleyMaster(nodes)
synchronized_times = master.synchronize_clocks()
print(synchronized_times)
```

Este código demuestra cómo un nodo maestro ajusta los tiempos de los nodos esclavos para sincronizarlos.

Algoritmo de Cristian

El Algoritmo de Cristian es un método de sincronización de relojes en el cual un nodo cliente solicita la hora a un servidor de tiempo. El servidor responde con su hora actual, y el cliente ajusta su reloj considerando el retraso de la red.

Implementación del algoritmo de Cristian

```
import time

class CristianServer:
    def __init__(self):
        self.time = time.time()

    def get_time(self):
        return self.time
```

```
class CristianClient:
    def __init__(self, server):
        self.server = server
        self.time = time.time()

    def synchronize_clock(self):
        request_time = time.time()
        server_time = self.server.get_time()
        response_time = time.time()
        round_trip_time = response_time - request_time
        self.time = server_time + round_trip_time / 2

# Ejemplo de uso
server = CristianServer()
client = CristianClient(server)
client.synchronize_clock()
print(client.time)
```

Este ejemplo muestra cómo un cliente sincroniza su reloj con el tiempo del servidor considerando el retraso de la red.

Algoritmo de intersección

El algoritmo de Intersección sincroniza relojes en un sistema distribuido calculando un intervalo de confianza en el que se supone que se encuentra el tiempo real. Los nodos intercambian estos intervalos y calculan la intersección de todos ellos para determinar el tiempo correcto.

Implementación del algoritmo de intersección

```
class IntersectionNode:
    def __init__(self, node_id, time, uncertainty):
        self.node_id = node_id
        self.time = time
        self.uncertainty = uncertainty

    def get_time_interval(self):
        return (self.time - self.uncertainty, self.time +
self.uncertainty)

def calculate_intersection(nodes):
    intervals = [node.get_time_interval() for node in nodes]
    max_start = max(interval[0] for interval in intervals)
    min_end = min(interval[1] for interval in intervals)
    if max_start <= min_end:
        return (max_start + min_end) / 2
    else:
        return None

# Ejemplo de uso
nodes = [IntersectionNode(i, time, uncertainty) for i, (time,
```

```
uncertainty) in enumerate([(10, 2), (12, 1), (14, 3)])]
intersection_time = calculate_intersection(nodes)
print(intersection_time)
```

Este código muestra cómo los nodos calculan la intersección de sus intervalos de tiempo para determinar el tiempo sincronizado.

Algoritmo de Marzullo

El algoritmo de Marzullo es una extensión del Algoritmo de Intersección que permite manejar múltiples fuentes de tiempo con diferentes precisiones y confiabilidades. Este algoritmo encuentra el intervalo más estrecho en el que se solapan la mayoría de las fuentes de tiempo.

Implementación del algoritmo de Marzullo

```
class MarzulloNode:
    def __init__(self, node_id, time, uncertainty):
        self.node_id = node_id
        self.time = time
        self.uncertainty = uncertainty

    def get_time_interval(self):
        return (self.time - self.uncertainty, self.time +
self.uncertainty)

def marzullo_algorithm(nodes):
    intervals = [node.get_time_interval() for node in nodes]
    events = []
    for start, end in intervals:
        events.append((start, +1))
        events.append((end, -1))
    events.sort()

    max_count = 0
    count = 0
    best_interval = None
    for event in events:
        count += event[1]
        if count > max_count:
            max_count = count
            best_interval = event[0]

    return best_interval

# Ejemplo de uso
nodes = [MarzulloNode(i, time, uncertainty) for i, (time, uncertainty)
in enumerate([(10, 2), (12, 1), (14, 3)])]
best_time = marzullo_algorithm(nodes)
print(best_time)
```

Este ejemplo muestra cómo se utiliza el Algoritmo de Marzullo para encontrar el intervalo de tiempo más confiable entre múltiples nodos.

Ordenamiento de Lamport

El ordenamiento de Lamport es un método para establecer un orden parcial de eventos en un sistema distribuido basado en la relación de "sucedió antes". Este método utiliza relojes lógicos para asignar marcas de tiempo a los eventos y asegurar que la causalidad se respete.

Implementación del ordenamiento de Lamport

```
class LamportClock:
    def __init__(self):
        self.time = 0

    def tick(self):
        self.time += 1
        return self.time

    def send_event(self):
        return self.tick()

    def receive_event(self, received_time):
        self.time = max(self.time, received_time) + 1
        return self.time

# Ejemplo de uso
clock1 = LamportClock()
clock2 = LamportClock()

# Evento en el nodo 1
t1 = clock1.send_event()
print(f"Reloj 1: {t1}")

# Evento en el nodo 2
t2 = clock2.send_event()
print(f"Reloj 2: {t2}")

# Nodo 1 envía un mensaje a nodo 2
t3 = clock2.receive_event(t1)
print(f"Reloj 2 despues de recibir : {t3}")
```

Este ejemplo ilustra cómo los relojes de Lamport garantizan un orden causal entre eventos en diferentes nodos.

Relojes vectoriales

Los relojes vectoriales son una extensión de los relojes lógicos que proporcionan un orden parcial de eventos en un sistema distribuido y pueden detectar violaciones de causalidad. Cada nodo mantiene un vector de tiempos lógicos, uno por cada nodo en el sistema.

Implementación de relojes vectoriales

```python
class VectorClock:
    def __init__(self, num_nodes, node_id):
        self.clock = [0] * num_nodes
        self.node_id = node_id

    def tick(self):
        self.clock[self.node_id] += 1

    def send_event(self):
        self.tick()
        return self.clock[:]

    def receive_event(self, received_vector):
        for i in range(len(self.clock)):
            self.clock[i] = max(self.clock[i], received_vector[i])
        self.clock[self.node_id] += 1

# Ejemplo de uso
num_nodes = 3
node1 = VectorClock(num_nodes, 0)
node2 = VectorClock(num_nodes, 1)
node3 = VectorClock(num_nodes, 2)

# Evento en el nodo 1
node1.tick()
print(f" Reloj Nodo 1 : {node1.clock}")

# Nodo 1 envía un mensaje a nodo 2
msg = node1.send_event()
node2.receive_event(msg)
print(f"Reloj Nodo 2 despues de recibir: {node2.clock}")

# Evento en el nodo 3
node3.tick()
print(f"Reloj Nodo 3 : {node3.clock}")
```

Este ejemplo muestra cómo los relojes vectoriales pueden detectar y mantener la causalidad entre eventos en un sistema distribuido.

Algoritmos de distribución de recursos

En sistemas distribuidos, la correcta distribución y gestión de recursos es fundamental para garantizar la eficiencia y consistencia del sistema. Uno de los desafíos principales es asegurar que los recursos compartidos no sean accedidos simultáneamente por múltiples procesos, lo que puede llevar a condiciones de carrera y otros problemas de concurrencia. Los algoritmos de exclusión mutua distribuidos están diseñados para manejar este problema, permitiendo que los procesos en un sistema distribuido accedan a recursos compartidos de manera segura y ordenada.

Exclusión mutua

La exclusión mutua es un concepto fundamental en sistemas concurrentes y distribuidos. Garantiza que solo un proceso pueda acceder a un recurso crítico en un momento dado, evitando conflictos y condiciones de carrera. Los algoritmos de exclusión mutua distribuidos extienden este concepto a sistemas donde los procesos pueden estar en diferentes nodos de una red.

Algoritmo de exclusión mutua distribuida de Lamport

El Algoritmo de Exclusión Mutua Distribuida de Lamport es uno de los algoritmos más conocidos y utilizados para manejar la exclusión mutua en sistemas distribuidos. La exclusión mutua es un mecanismo fundamental para evitar condiciones de carrera y asegurar que solo un nodo acceda a un recurso compartido en un momento dado. El algoritmo de Lamport utiliza relojes lógicos para mantener un orden total de los eventos y asegurar que los nodos accedan al recurso crítico de manera secuencial y sin conflictos.

Fundamentos del algoritmo de Lamport

1. Relojes lógicos:
   - El algoritmo utiliza relojes lógicos de Lamport para mantener un orden causal entre los eventos en un sistema distribuido. Cada nodo en el sistema tiene un reloj lógico que se incrementa con cada evento local y se ajusta en función de los mensajes recibidos de otros nodos.
2. Solicitud de recursos:
   - Cuando un nodo desea acceder a un recurso compartido, envía un mensaje de solicitud a todos los demás nodos en el sistema. Este mensaje incluye el identificador del nodo y su valor de reloj lógico actual.
3. Cola de solicitudes:
   - Cada nodo mantiene una cola de solicitudes ordenadas por el valor de los relojes lógicos y, en caso de empate, por el identificador del nodo. Esta cola se utiliza para determinar el orden en que los nodos deben acceder al recurso compartido.
4. Acuse de recibo:
   - Los nodos que reciben una solicitud de acceso al recurso envían un mensaje de acuse de recibo al nodo solicitante. Este mensaje confirma que la solicitud ha sido recibida y registrada en la cola de solicitudes del nodo receptor.
5. Acceso al recurso:
   - Un nodo puede acceder al recurso compartido solo cuando ha recibido mensajes de acuse de recibo de todos los demás nodos y su solicitud está en la parte superior de su propia cola de solicitudes.
6. Liberación del recurso:
   - Después de usar el recurso compartido, el nodo envía un mensaje de liberación a todos los demás nodos, indicando que ya no necesita el recurso. Los nodos que reciben este mensaje eliminan la solicitud del nodo liberador de sus colas de solicitudes.

Ejemplo de funcionamiento

Consideremos un sistema con tres nodos A,B y C que desean acceder a un recurso compartido. Supongamos que los relojes lógicos iniciales de los nodos son 1 para A, 2 para B y 3 para C.

1. Nodo A solicita el recurso:
   - Nodo A incrementa su reloj lógico a 2 y envía un mensaje de solicitud $(A, 2)$ a B y C.
2. Nodo B y C reciben la solicitud:
   - Nodo B y C reciben la solicitud $(A, 2)$, ajustan sus relojes lógicos a 3 (el máximo entre su reloj actual y el valor de la solicitud más 1) y envían un acuse de recibo a A.
3. Nodo B solicita el recurso:
   - Nodo B incrementa su reloj lógico a 4 y envía un mensaje de solicitud $(B, 4)$ a A y C.
4. Nodo A y C reciben la solicitud:
   - Nodo A y C reciben la solicitud $(B, 4)$, ajustan sus relojes lógicos a 5 y envían un acuse de recibo a B.
5. Nodo C solicita el recurso:
   - Nodo C incrementa su reloj lógico a 6 y envía un mensaje de solicitud $(C, 6)$ a A y B.
6. Nodo A y B reciben la solicitud:
   - Nodo A y B reciben la solicitud $(C, 6)$, ajustan sus relojes lógicos a 7 y envían un acuse de recibo a C.
7. Acceso al recurso:
   - Nodo A ha recibido acuses de recibo de todos los nodos y su solicitud $(A, 2)$ está en la parte superior de su cola, por lo que puede acceder al recurso compartido.
   - Después de usar el recurso, nodo A envía un mensaje de liberación a B y C.
8. Nodo B y C reciben la liberación:
   - Nodo B y C reciben el mensaje de liberación, eliminan la solicitud de A de sus colas y nodo B puede acceder al recurso porque su solicitud $(B, 4)$ ahora está en la parte superior de su cola.

Implementación del algoritmo de Lamport

```
import threading
import time
from queue import PriorityQueue

class LamportMutex:
    def __init__(self, node_id, num_nodes):
        self.node_id = node_id
```

```
        self.num_nodes = num_nodes
        self.clock = 0
        self.request_queue = PriorityQueue()
        self.replies_received = 0
        self.lock = threading.Lock()

    def send_request(self):
        self.clock += 1
        self.request_queue.put((self.clock, self.node_id))
        for i in range(self.num_nodes):
            if i != self.node_id:
                self.send_message(i, 'REQUEST', self.clock,
self.node_id)

    def send_message(self, target, msg_type, timestamp, sender_id):
        # Simulate sending a message over the network
        print(f"Nodo {self.node_id} enviando {msg_type} al Nodo
{target} con timestamp {timestamp}")
        time.sleep(0.1)
        # In a real implementation, this would send the message over
the network

    def receive_message(self, msg_type, timestamp, sender_id):
        with self.lock:
            self.clock = max(self.clock, timestamp) + 1
            if msg_type == 'REQUEST':
                self.request_queue.put((timestamp, sender_id))
                self.send_message(sender_id, 'REPLY', self.clock,
self.node_id)
            elif msg_type == 'REPLY':
                self.replies_received += 1

    def enter_critical_section(self):
        self.send_request()
        while self.replies_received < self.num_nodes - 1:
            time.sleep(0.1)
        print(f"Nodo {self.node_id} ingresando a la seccion crítica")

    def leave_critical_section(self):
        with self.lock:
            self.request_queue.get()
        for i in range(self.num_nodes):
            if i != self.node_id:
                self.send_message(i, 'RELEASE', self.clock,
self.node_id)
        print(f"Nodo {self.node_id} dejando la seccion critica")

# Ejemplo de uso
num_nodes = 3
nodes = [LamportMutex(i, num_nodes) for i in range(num_nodes)]
```

```
# Simulate node 0 entering critical section
nodes[0].enter_critical_section()
time.sleep(2)
nodes[0].leave_critical_section()
```

Este ejemplo demuestra cómo los nodos pueden coordinar el acceso a un recurso crítico utilizando el algoritmo de Lamport.

Algoritmo log(n) de Naimi-Trehel

El algoritmo log(n) de Naimi-Trehel es un algoritmo de exclusión mutua distribuida que optimiza el número de mensajes requeridos para alcanzar un acuerdo. Utiliza una estructura de árbol para distribuir las solicitudes de acceso, lo que reduce la complejidad del algoritmo.

Implementación del algoritmo de Naimi-Trehel

```
class NaimiTrehelMutex:
    def __init__(self, node_id, num_nodes):
        self.node_id = node_id
        self.num_nodes = num_nodes
        self.token_holder = False
        self.requesting = False
        self.parent = None
        self.children = set()

    def request_token(self):
        if self.token_holder:
            self.enter_critical_section()
        else:
            self.requesting = True
            if self.parent:
                self.send_request(self.parent)

    def send_request(self, target):
        # Simulate sending a request over the network
        print(f"Node {self.node_id} sending REQUEST to Node {target}")
        time.sleep(0.1)

    def receive_request(self, sender_id):
        if self.token_holder:
            self.token_holder = False
            self.send_token(sender_id)
        else:
            self.children.add(sender_id)
            if self.requesting:
                self.send_request(self.parent)

    def send_token(self, target):
        # Simulate sending the token over the network
```

```
        print(f"Nodo {self.node_id} enviando TOKEN al nodo {target}")
        time.sleep(0.1)
        self.token_holder = False
        if target in self.children:
            self.children.remove(target)
        self.parent = target

    def receive_token(self):
        self.token_holder = True
        if self.requesting:
            self.enter_critical_section()

    def enter_critical_section(self):
        print(f"Nodo {self.node_id} ingresando a la secciin critica")
        # Critical section code here
        self.leave_critical_section()

    def leave_critical_section(self):
        self.requesting = False
        if self.children:
            child = self.children.pop()
            self.send_token(child)
        print(f"Nodo {self.node_id} dejando la seccion critica")

# Ejemplo de uso
num_nodes = 3
nodes = [NaimiTrehelMutex(i, num_nodes) for i in range(num_nodes)]

# Simulate node 0 requesting the token
nodes[0].request_token()
time.sleep(2)
nodes[0].leave_critical_section()
```

Este ejemplo muestra cómo los nodos utilizan el algoritmo de Naimi-Trehel para coordinar el acceso a un recurso crítico.

Algoritmo de Maekawa

El algoritmo de Maekawa divide los nodos en grupos de votación, y cada nodo pertenece a varios grupos. Para acceder a un recurso crítico, un nodo debe obtener permisos de todos los nodos en sus grupos de votación.

Implementación del algoritmo de Maekawa

```
class MaekawaMutex:
    def __init__(self, node_id, quorum):
        self.node_id = node_id
        self.quorum = quorum
        self.voted = False
        self.votes_received = 0
```

```
    def request_access(self):
        self.votes_received = 0
        for node in self.quorum:
            node.receive_request(self)

    def receive_request(self, requester):
        if not self.voted:
            self.voted = True
            requester.receive_vote(self)

    def receive_vote(self, voter):
        self.votes_received += 1
        if self.votes_received == len(self.quorum):
            self.enter_critical_section()

    def enter_critical_section(self):
        print(f"Nodo {self.node_id} ingresando a la seccion critica")
        # Critical section code here
        self.leave_critical_section()

    def leave_critical_section(self):
        print(f"Nodo {self.node_id} dejando la seccion critica")
        for node in self.quorum:
            node.release_vote(self)

    def release_vote(self, requester):
        self.voted = False

# Ejemplo de uso
num_nodes = 3
quorum = [
    [0, 1],
    [1, 2],
    [0, 2]
]
nodes = [MaekawaMutex(i, [nodes[j] for j in q if j != i]) for i, q in
enumerate(quorum)]

# Simulate node 0 requesting access
nodes[0].request_access()
time.sleep(2)
nodes[0].leave_critical_section()
```

Este ejemplo ilustra cómo los nodos coordinan el acceso a recursos críticos utilizando el Algoritmo de Maekawa.

Algoritmo de Raymond

El algoritmo de Raymond utiliza una estructura de árbol en la que los nodos pasan un token de control a lo largo de las ramas del árbol. Solo el nodo que posee el token puede acceder al

recurso crítico. Cuando un nodo desea acceder al recurso, envía una solicitud a su padre en el árbol.

Implementación del algoritmo de Raymond

```
class RaymondMutex:
    def __init__(self, node_id, parent=None):
        self.node_id = node_id
        self.parent = parent
        self.token_holder = (parent is None)
        self.request_queue = []

    def request_access(self):
        if self.token_holder:
            self.enter_critical_section()
        else:
            self.request_queue.append(self.node_id)
            self.send_request_to_parent()

    def send_request_to_parent(self):
        if self.parent:
            self.parent.receive_request(self)

    def receive_request(self, requester):
        if not self.token_holder:
            self.request_queue.append(requester.node_id)
            self.send_request_to_parent()
        elif requester.node_id == self.node_id:
            self.enter_critical_section()
        else:
            self.send_token(requester)

    def send_token(self, requester):
        self.token_holder = False
        requester.receive_token(self)

    def receive_token(self, sender):
        self.token_holder = True
        if self.request_queue and self.request_queue[0] ==
self.node_id:
            self.enter_critical_section()
        else:
            self.send_token_to_next_in_queue()

    def send_token_to_next_in_queue(self):
        next_node_id = self.request_queue.pop(0)
        next_node = [node for node in nodes if node.node_id ==
next_node_id][0]
        self.send_token(next_node)
```

```
    def enter_critical_section(self):
        print(f"Nodo {self.node_id} ingresando a la seccion critica")
        # Critical section code here
        self.leave_critical_section()

    def leave_critical_section(self):
        print(f"Nodo {self.node_id} dejando critical section")
        if self.request_queue:
            self.send_token_to_next_in_queue()

# Ejemplo de uso
nodes = [RaymondMutex(i) for i in range(3)]
nodes[0].parent = nodes[1]
nodes[1].parent = nodes[2]

# Simulate node 0 requesting access
nodes[0].request_access()
time.sleep(2)
nodes[0].leave_critical_section()
```

Este ejemplo muestra cómo los nodos utilizan el algoritmo de Raymond para coordinar el acceso a un recurso crítico.

Algoritmo de Ricart-Agrawala

El algoritmo de Ricart-Agrawala es un algoritmo de exclusión mutua distribuida que utiliza un protocolo de dos fases. Los nodos envían mensajes de solicitud de acceso a todos los demás nodos y esperan respuestas antes de acceder al recurso crítico. Este algoritmo garantiza que no haya interbloqueos y es eficiente en términos de mensajes requeridos.

Implementación del algoritmo de Ricart-Agrawala

```
class RicartAgrawalaMutex:
    def __init__(self, node_id, num_nodes):
        self.node_id = node_id
        self.num_nodes = num_nodes
        self.clock = 0
        self.request_queue = []
        self.replies_received = 0

    def request_access(self):
        self.clock += 1
        self.request_queue.append((self.clock, self.node_id))
        for node in nodes:
            if node.node_id != self.node_id:
                node.receive_request(self.clock, self.node_id)

    def receive_request(self, timestamp, sender_id):
        self.clock = max(self.clock, timestamp) + 1
        self.request_queue.append((timestamp, sender_id))
```

```
        self.request_queue.sort()
        self.send_reply(sender_id)

    def send_reply(self, target_id):
        for node in nodes:
            if node.node_id == target_id:
                node.receive_reply(self.node_id)

    def receive_reply(self, sender_id):
        self.replies_received += 1
        if self.replies_received == self.num_nodes - 1:
            self.enter_critical_section()

    def enter_critical_section(self):
        print(f"Nodo {self.node_id} ingresando a la seccion critica")
        # Critical section code here
        self.leave_critical_section()

    def leave_critical_section(self):
        self.replies_received = 0
        self.request_queue = [(t, n) for t, n in self.request_queue if
n != self.node_id]
        for timestamp, node_id in self.request_queue:
            self.send_reply(node_id)
        print(f"Nodo {self.node_id} dejando la seccion critica)

# Ejemplo de uso
num_nodes = 3
nodes = [RicartAgrawalaMutex(i, num_nodes) for i in range(num_nodes)]

# Simulate node 0 requesting access
nodes[0].request_access()
time.sleep(2)
nodes[0].leave_critical_section()
```

Los algoritmos de exclusión mutua distribuidos son esenciales para garantizar el acceso seguro y ordenado a recursos críticos en sistemas distribuidos.

Algoritmos de toma de instantáneas

En sistemas distribuidos, registrar un estado global consistente es una tarea crucial para la recuperación de fallos, la detección de errores y la implementación de ciertas funcionalidades de depuración. Los algoritmos de toma de instantáneas permiten capturar el estado global de un sistema asíncrono, proporcionando una visión coherente de todos los procesos y canales de comunicación en un instante específico.

Algoritmo de toma de instantáneas (Snapshot)

El algoritmo de Toma de Instantáneas se utiliza para capturar el estado global consistente de un sistema distribuido. Este estado global incluye el estado de todos los procesos y el estado de los

canales de comunicación entre estos procesos. En un sistema asíncrono, los procesos no comparten un reloj común, lo que hace que la sincronización de estados sea un desafío.

Principios del algoritmo de toma de instantáneas

El algoritmo de Toma de Instantáneas funciona mediante la captura del estado de cada proceso y el estado de los mensajes en tránsito en los canales de comunicación. Para lograr un estado global consistente, el algoritmo sigue estos pasos:

1. Iniciación: Un proceso inicia la toma de instantáneas enviando un mensaje de marcador (marker) a todos los demás procesos.
2. Captura del estado del proceso: Al recibir el mensaje de marcador, cada proceso captura su propio estado local.
3. Captura del estado del canal: Los procesos registran los mensajes recibidos por cada canal después de recibir el mensaje de marcador y antes de capturar el estado del canal.

Estos pasos aseguran que el estado global capturado refleje un instante coherente en el tiempo del sistema distribuido.

Algoritmo de Chandy-Lamport

El algoritmo de Chandy-Lamport es uno de los métodos más conocidos para la toma de instantáneas en sistemas distribuidos. Este algoritmo garantiza que se capture un estado global consistente sin necesidad de sincronización de relojes entre los procesos.

Principios del algoritmo de Chandy-Lamport

El algoritmo de Chandy-Lamport se basa en el uso de mensajes de marcador para coordinar la captura del estado global. El algoritmo sigue estos pasos:

1. Iniciación: Un proceso inicia la toma de instantáneas registrando su propio estado y enviando mensajes de marcador a todos los demás procesos.
2. Recepción del marcador: Cuando un proceso recibe un mensaje de marcador por primera vez, registra su estado y envía mensajes de marcador a todos sus vecinos. Si un proceso recibe un mensaje de marcador por un canal del cual ya ha recibido un marcador anteriormente, simplemente registra el estado del canal.
3. Captura del estado del canal: Los procesos registran todos los mensajes que reciben después de haber registrado su estado y antes de recibir el primer marcador en cada canal de entrada.

Este proceso asegura que se capture un estado global consistente en el sistema distribuido.

Implementación del algoritmo de Chandy-Lamport

A continuación se presenta una implementación simplificada del Algoritmo de Chandy-Lamport en Python:

```
import threading
import time
from collections import defaultdict

class Process:
```

```
    def __init__(self, process_id):
        self.process_id = process_id
        self.state = None
        self.channels = defaultdict(list)
        self.neighbors = []
        self.marker_received = {}
        self.local_snapshot = None
        self.lock = threading.Lock()

    def set_neighbors(self, neighbors):
        self.neighbors = neighbors
        for neighbor in neighbors:
            self.marker_received[neighbor.process_id] = False

    def initiate_snapshot(self):
        with self.lock:
            self.local_snapshot = self.state
            print(f"Process {self.process_id} taking local snapshot:
{self.local_snapshot}")
            self.send_marker_messages()

    def send_marker_messages(self):
        for neighbor in self.neighbors:
            self.send_message(neighbor, 'MARKER')

    def send_message(self, neighbor, message_type, content=None):
        message = (message_type, self.process_id, content)
        neighbor.receive_message(message)

    def receive_message(self, message):
        message_type, sender_id, content = message
        with self.lock:
            if message_type == 'MARKER':
                if not self.marker_received[sender_id]:
                    self.marker_received[sender_id] = True
                    if self.local_snapshot is None:
                        self.local_snapshot = self.state
                        print(f"Process {self.process_id} taking local
snapshot: {self.local_snapshot}")
                        self.send_marker_messages()
                    else:
                        self.channels[sender_id].append(content)
                else:
                    self.channels[sender_id].append(content)
            else:
                if self.local_snapshot is not None:
                    self.channels[sender_id].append(content)
                else:
                    self.process_message(message)
```

```
    def process_message(self, message):
        # Simulate processing of a normal message
        print(f"Process {self.process_id} received message from
Process {message[1]}: {message[2]}")

    def update_state(self, new_state):
        self.state = new_state

# Ejemplo de uso
processes = [Process(i) for i in range(3)]
for i, process in enumerate(processes):
    neighbors = [p for j, p in enumerate(processes) if i != j]
    process.set_neighbors(neighbors)

# Simulate state updates
processes[0].update_state("State A")
processes[1].update_state("State B")
processes[2].update_state("State C")

# Initiate snapshot from process 0
processes[0].initiate_snapshot()

# Simulate message passing
time.sleep(1)
processes[1].send_message(processes[0], 'MESSAGE', "Message 1 from P1
to P0")
time.sleep(1)
processes[2].send_message(processes[0], 'MESSAGE', "Message 2 from P2
to P0")
time.sleep(1)
processes[2].send_message(processes[1], 'MESSAGE', "Message 3 from P2
to P1")
```

Este ejemplo muestra cómo los procesos pueden coordinarse para tomar una instantánea del estado global utilizando el Algoritmo de Chandy-Lamport. La implementación incluye la captura del estado local y el registro de mensajes en tránsito entre los procesos.

Comparación de algoritmos de toma de instantáneas

Además del algoritmo de Chandy-Lamport, existen otros algoritmos de toma de instantáneas utilizados en sistemas distribuidos, como el algoritmo de snapshot Distribuido y el algoritmo de Mattern. Estos algoritmos también buscan capturar un estado global consistente, pero pueden diferir en la forma en que manejan la sincronización de mensajes y la captura del estado.

- Algoritmo de snapshot distribuido: Utiliza mensajes de marcador similares a los de Chandy-Lamport, pero puede incluir optimizaciones para reducir el número de mensajes necesarios.
- Algoritmo de Mattern: Introduce el concepto de colores para marcar el estado de los procesos y los canales, utilizando una técnica de propagación de colores para determinar cuándo se ha capturado el estado global. Cada uno de estos algoritmos tiene sus propias

ventajas y desventajas, y la elección del algoritmo adecuado puede depender de las características específicas del sistema distribuido en el que se implementa.

Algoritmos de gestión de memoria

La gestión eficiente de la memoria es crucial en el diseño de sistemas operativos y en el desarrollo de aplicaciones para garantizar que los recursos de memoria sean utilizados de manera óptima. Existen varios algoritmos y técnicas para la asignación, liberación y recolección de memoria que abordan diversos problemas, como la fragmentación, la velocidad de asignación y la liberación, y la gestión de la memoria utilizada.

Algoritmos de asignación y liberación de memoria

Los algoritmos de asignación y liberación de memoria se encargan de gestionar la memoria dinámica, asegurando que los bloques de memoria sean asignados y liberados de manera eficiente.

Asignación de memoria Buddy

El algoritmo de asignación de memoria Buddy es un método que minimiza la fragmentación mediante la división de la memoria en bloques de tamaño potencia de dos. Cuando se solicita un bloque de memoria, el sistema busca el bloque libre más pequeño que puede satisfacer la solicitud. Si no se encuentra un bloque adecuado, el algoritmo divide un bloque más grande en dos "buddy" de igual tamaño hasta encontrar un bloque adecuado.

Implementación del algoritmo de asignación de memoria Buddy

```
class BuddyAllocator:
    def __init__(self, size):
        self.size = size
        self.free_list = {size: [0]}

    def allocate(self, size):
        size = self._next_power_of_two(size)
        for free_size in sorted(self.free_list.keys()):
            if free_size >= size:
                return self._split(free_size, size)
        return None

    def _split(self, free_size, size):
        if free_size == size:
            addr = self.free_list[free_size].pop(0)
            if not self.free_list[free_size]:
                del self.free_list[free_size]
            return addr
        addr = self.free_list[free_size].pop(0)
        buddy_size = free_size // 2
        if free_size not in self.free_list:
            self.free_list[buddy_size] = []
        if buddy_size not in self.free_list:
            self.free_list[buddy_size] = []
```

```
        self.free_list[buddy_size].append(addr + buddy_size)
        self.free_list[buddy_size].append(addr)
        return self._split(buddy_size, size)

    def free(self, addr, size):
        size = self._next_power_of_two(size)
        if size not in self.free_list:
            self.free_list[size] = []
        self.free_list[size].append(addr)
        self._coalesce(size)

    def _coalesce(self, size):
        while size < self.size:
            merged = False
            free_list = self.free_list[size]
            free_list.sort()  # Ensure the free_list is sorted
            for i in range(0, len(free_list), 2):
                if i + 1 < len(free_list) and free_list[i] + size ==
free_list[i + 1]:
                    addr = free_list[i]
                    free_list.pop(i)
                    free_list.pop(i)  # Since the list is shortened,
pop the same index again
                    size *= 2
                    if size not in self.free_list:
                        self.free_list[size] = []
                    self.free_list[size].append(addr)
                    merged = True
                    break
            if not merged:
                break

    def _next_power_of_two(self, x):
        return 1 << (x - 1).bit_length()

# Example usage
allocator = BuddyAllocator(1024)
addr1 = allocator.allocate(200)
print(f"Aignado en: {addr1}")
allocator.free(addr1, 200)
print("Memoria liberada")
```

Este código demuestra cómo el algoritmo Buddy asigna y libera memoria de manera eficiente para minimizar la fragmentación.

Recolectores de basura

Los recolectores de basura son algoritmos que se encargan de la gestión automática de la memoria, liberando la memoria que ya no es utilizada por el programa. Existen varios tipos de recolectores de basura, cada uno con sus propias ventajas y desventajas.

Algoritmo de Cheney

El algoritmo de Cheney es un recolector de basura que mejora el recolector de semiespacio. Divide la memoria en dos semiespacios y alterna entre ellos. Durante la recolección, copia los objetos vivos del semiespacio actual al otro semiespacio, compactando la memoria y eliminando la fragmentación.

Implementación del algoritmo de Cheney

```
class CheneyCollector:
    def __init__(self, size):
        self.size = size
        self.from_space = [None] * size
        self.to_space = [None] * size
        self.free_ptr = 0

    def allocate(self, obj):
        if self.free_ptr >= self.size:
            self.collect()
        addr = self.free_ptr
        self.from_space[addr] = obj
        self.free_ptr += 1
        return addr

    def collect(self):
        self.to_space = [None] * self.size
        self.free_ptr = 0
        for obj in self.from_space:
            if obj is not None:
                self.copy(obj)
        self.from_space, self.to_space = self.to_space,
self.from_space

    def copy(self, obj):
        addr = self.free_ptr
        self.to_space[addr] = obj
        self.free_ptr += 1
        return addr

# Ejemplo de uso
collector = CheneyCollector(10)
addr1 = collector.allocate("obj1")
print(f"Asignado obj1 en: {addr1}")
collector.collect()
print("Recoleccion de basura completa")
```

Esta implementación muestra cómo el algoritmo de Cheney gestiona la recolección de basura y la compactación de la memoria.

Recolector de basura generacional

El recolector de basura generacional es una técnica que mejora la eficiencia dividiendo los objetos en generaciones según su edad. La idea es que los objetos jóvenes tienen más probabilidades de ser recolectados pronto, mientras que los objetos viejos son más estables.

Implementación del recolector de basura generacional

```
class GenerationalCollector:
    def __init__(self, size):
        self.size = size
        self.young_gen = [None] * size
        self.old_gen = [None] * size
        self.young_ptr = 0
        self.old_ptr = 0

    def allocate(self, obj, old=False):
        if old:
            if self.old_ptr >= self.size:
                self.collect_old()
            addr = self.old_ptr
            self.old_gen[addr] = obj
            self.old_ptr += 1
        else:
            if self.young_ptr >= self.size:
                self.collect_young()
            addr = self.young_ptr
            self.young_gen[addr] = obj
            self.young_ptr += 1
        return addr

    def collect_young(self):
        self.old_gen = self.old_gen + [obj for obj in self.young_gen
if obj is not None]
        self.young_gen = [None] * self.size
        self.young_ptr = 0

    def collect_old(self):
        self.old_gen = [obj for obj in self.old_gen if obj is not
None]
        self.old_ptr = len(self.old_gen)
        self.old_gen += [None] * (self.size - self.old_ptr)

# Ejemplo de uso
collector = GenerationalCollector(10)
addr1 = collector.allocate("obj1")
print(f"Asignado en el obj1: {addr1}")
collector.collect_young()
print("Se completa la recoleccion de basura de la generacion joven")
```

Esta implementación ilustra cómo los recolectores de basura generacionales gestionan objetos de diferentes edades para optimizar la recolección.

Algoritmo de marcado-compacto

El algoritmo de marcado-compacto combina el algoritmo de marcado-barrido con el de copia de Cheney. Primero, marca todos los objetos vivos y luego compácta la memoria copiando los objetos vivos a un nuevo espacio contiguo.

Implementación del algoritmo de marcado-compacto

```
class MarkCompactCollector:
    def __init__(self, size):
        self.size = size
        self.memory = [None] * size
        self.marked = [False] * size

    def allocate(self, obj):
        for i in range(self.size):
            if self.memory[i] is None:
                self.memory[i] = obj
                return i
        self.collect()
        return self.allocate(obj)

    def mark(self, roots):
        for root in roots:
            self._mark(root)

    def _mark(self, obj):
        if obj is not None:
            addr = obj
            if not self.marked[addr]:
                self.marked[addr] = True

    def compact(self):
        new_memory = [None] * self.size
        j = 0
        for i in range(self.size):
            if self.marked[i]:
                new_memory[j] = self.memory[i]
                j += 1
        self.memory = new_memory

    def collect(self):
        self.mark(self.memory)
        self.compact()
        self.marked = [False] * self.size

# Ejemplo de uso
collector = MarkCompactCollector(10)
addr1 = collector.allocate("obj1")
print(f"Asignado el obj1 en: {addr1}")
```

```
collector.collect()
print("Recoleccion de basura completa")
```

Este ejemplo muestra cómo el algoritmo de marcado-compacto maneja la recolección de basura y la compactación de la memoria.

Algoritmo de marcado y barrido

El algoritmo de marcado y barrido es uno de los recolectores de basura más simples. Consiste en dos fases: primero, marca todos los objetos vivos y, segundo, barre la memoria para liberar los objetos no marcados.

Implementación del algoritmo de marcado y barrido

```
class MarkSweepCollector:
    def __init__(self, size):
        self.size = size
        self.memory = [None] * size
        self.marked = [False] * size
        self.object_to_address = {}

    def allocate(self, obj):
        for i in range(self.size):
            if self.memory[i] is None:
                self.memory[i] = obj
                self.object_to_address[obj] = i
                return i
        self.collect()
        return self.allocate(obj)

    def mark(self, roots):
        for root in roots:
            self._mark(root)

    def _mark(self, obj):
        if obj is not None and obj in self.object_to_address:
            addr = self.object_to_address[obj]
            if not self.marked[addr]:
                self.marked[addr] = True

    def sweep(self):
        for i in range(self.size):
            if not self.marked[i]:
                obj = self.memory[i]
                if obj is not None:
                    del self.object_to_address[obj]
                self.memory[i] = None
            else:
                self.marked[i] = False
```

```
    def collect(self):
        self.mark(self.memory)
        self.sweep()

# Example usage
collector = MarkSweepCollector(10)
addr1 = collector.allocate("obj1")
print(f"Asignado al obj1 em: {addr1}")
collector.collect()
print("Recoleccion de basura completa")
```

Recolector de semiespacio

El recolector de semiespacio es un recolector de basura de copia temprana que divide la memoria en dos semiespacios y alterna entre ellos. Los objetos vivos se copian de un semiespacio a otro durante la recolección.

Implementación del recolector de semiespacio

```
class SemispaceCollector:
    def __init__(self, size):
        self.size = size
        self.from_space = [None] * size
        self.to_space = [None] * size
        self.free_ptr = 0

    def allocate(self, obj):
        if self.free_ptr >= self.size:
            self.collect()
        addr = self.free_ptr
        self.from_space[addr] = obj
        self.free_ptr += 1
        return addr

    def collect(self):
        self.to_space = [None] * self.size
        self.free_ptr = 0
        for obj in self.from_space:
            if obj is not None:
                self.copy(obj)
        self.from_space, self.to_space = self.to_space,
self.from_space

    def copy(self, obj):
        addr = self.free_ptr
        self.to_space[addr] = obj
        self.free_ptr += 1
        return addr

# Ejemplo de uso
```

```
collector = SemispaceCollector(10)
addr1 = collector.allocate("obj1")
print(f"Asignado el obj1 en: {addr1}")
collector.collect()
print("Recoleccion de basura completa")
```

Ejercicios

Ejercicio 1: Sistema de consenso y elección de líder

Desarrolla una aplicación distribuida para gestionar una base de datos de inventario en un sistema distribuido. Implementa los siguientes requisitos:

- Usa el algoritmo de Paxos para garantizar el consenso en la actualización de registros de inventario.
- Implementa el Algoritmo de Bully para la elección dinámica de un coordinador que gestionará las solicitudes de los clientes.
- Utiliza relojes vectoriales para asegurar el ordenamiento parcial de los eventos y detectar violaciones de causalidad en la actualización de registros.
- Implementa un recolector de basura generacional para gestionar la memoria en el nodo coordinador.

Ejercicio 2: Sincronización y exclusión mutua

Diseña un sistema de reservas de salas de reuniones en una empresa utilizando los siguientes algoritmos:

- Implementa el algoritmo de Berkeley para sincronizar los relojes de todos los nodos que representan diferentes oficinas.
- Usa el algoritmo de Exclusión Mutua Distribuida de Lamport para asegurar que solo un nodo pueda reservar una sala de reuniones a la vez.
- Implementa el algoritmo de Marzullo para gestionar la precisión de la sincronización del tiempo entre nodos.
- Utiliza el algoritmo de marcado y barrido para recolectar la memoria de manera eficiente en cada nodo.

Ejercicio 3: Detección de terminación y gestión de recursos

Crea un sistema distribuido para la ejecución de tareas científicas en una red de computadoras utilizando:

- El algoritmo de Dijkstra-Scholten para detectar la terminación de procesos distribuidos.
- Implementa el algoritmo de Ricart-Agrawala para la exclusión mutua en el acceso a recursos compartidos como archivos de datos.
- Usa el algoritmo de intersección para sincronizar los relojes de los nodos antes de la ejecución de tareas críticas.
- Implementa el algoritmo de Cheney para gestionar la recolección de basura en los nodos de computación.

Ejercicio 4: Consistencia y toma de instantáneas

Desarrolla una aplicación de banca distribuida que garantice la consistencia de las transacciones bancarias utilizando:

- Implementa el algoritmo de Raft para el consenso en la replicación de las transacciones en diferentes nodos.
- Usa el algoritmo de Chandy-Lamport para tomar instantáneas del estado global del sistema, asegurando que las transacciones se reflejen de manera consistente.
- Implementa el algoritmo de Maekawa para la exclusión mutua en el acceso a cuentas bancarias durante las transacciones.
- Utiliza el recolector de semiespacio para gestionar la memoria en los nodos que almacenan las transacciones.

Ejercicio 5: Tolerancia a fallos y sincronización de eventos

Diseña un sistema de monitoreo de red que pueda tolerar fallos y asegurar la correcta sincronización de eventos utilizando:

- Implementa un protocolo de tolerancia a fallos Bizantina para asegurar que los datos de monitoreo sean precisos incluso en presencia de nodos maliciosos.
- Usa el algoritmo de Cristian para sincronizar los relojes de los nodos de monitoreo con un servidor de tiempo central.
- Implementa el algoritmo de Raymond para la exclusión mutua en el acceso a los datos de monitoreo.
- Utiliza el algoritmo de marcado-compacto para gestionar la recolección de basura y asegurar el uso eficiente de la memoria en los nodos de monitoreo.

```
## Tus respuestas
```

Otros ejercicios

Ejercicio 6: Sistema de archivos distribuidos con HDFS y GFS

Desarrolla una aplicación que almacene y gestione grandes volúmenes de datos utilizando sistemas de archivos distribuidos:

- Implementa la gestión de datos utilizando Hadoop Distributed File System (HDFS).
- Integra Google File System (GFS) para asegurar la replicación y distribución de archivos a través de múltiples nodos.
- Usa el algoritmo de Paxos para coordinar el acceso concurrente a los archivos en el sistema.
- Implementa el algoritmo de Berkeley para sincronizar los relojes de los nodos en el clúster.

Ejercicio 7: Bases de datos distribuidas y el Teorema CAP

Diseña un sistema de bases de datos distribuidas que cumpla con los principios del Teorema CAP:

- Implementa una base de datos NoSQL usando Cassandra y MongoDB, asegurando la consistencia eventual.

- Desarrolla una base de datos SQL distribuida utilizando Google Spanner y CockroachDB, garantizando la consistencia transaccional.
- Utiliza el algoritmo de Raft para el consenso en la replicación de datos.
- Implementa el algoritmo de Ricart-Agrawala para la exclusión mutua en operaciones de escritura.

Ejercicio 8: Plataforma de stream processing con Apache Kafka

Crea una plataforma de procesamiento de streams utilizando Apache Kafka:

- Implementa productores y consumidores de mensajes para procesar eventos en tiempo real.
- Usa Apache Zookeeper para la coordinación y gestión de clústeres de Kafka.
- Integra gRPC para la comunicación de alto rendimiento entre servicios distribuidos.
- Implementa un recolector de basura generacional para gestionar la memoria en los nodos de procesamiento.

Ejercicio 9: Microservicios y contenerización con Docker y Kubernetes

Desarrolla una arquitectura de microservicios utilizando Docker y Kubernetes:

- Implementa microservicios contenedorizados usando Docker para la gestión de servicios.
- Usa Kubernetes para orquestar y escalar los contenedores de manera automática.
- Integra un service Mesh como Istio o Linkerd para gestionar la comunicación y el tráfico entre microservicios.
- Implementa el algoritmo de marcado-compacto para la recolección de basura en los contenedores.

Ejercicio 10: Sistemas de computación en la nube con MapReduce y Hadoop

Diseña un sistema de procesamiento de datos en la nube utilizando MapReduce y Hadoop:

- Implementa trabajos de MapReduce para procesar grandes volúmenes de datos distribuidos en un clúster Hadoop.
- Usa Apache Spark para el procesamiento en memoria de datos distribuidos.
- Integra AWS Lambda o Google Cloud Functions para la computación sin servidor, ejecutando funciones en respuesta a eventos.
- Utiliza el algoritmo de Chandy-Lamport para tomar instantáneas del estado global del sistema durante el procesamiento de datos.

Ejercicio 11: Monitoreo y coordinación con Apache Zookeeper

Crea un sistema de monitoreo distribuido utilizando Apache Zookeeper:

- Implementa nodos de monitoreo que registren el estado de los servicios distribuidos.
- Usa Zookeeper para la coordinación y gestión de configuraciones distribuidas.
- Integra Apache Kafka para la recolección y procesamiento de logs de monitoreo en tiempo real.
- Implementa el algoritmo de Lamport para el ordenamiento de eventos basados en la relación de "sucedió antes".

Ejercicio 12: Plataforma Serverless con AWS Lambda y Google Cloud Functions

Desarrolla una plataforma de computación sin servidor utilizando AWS Lambda y Google Cloud Functions:

- Implementa funciones que se ejecuten en respuesta a eventos generados por los usuarios o por otros servicios.
- Usa Apache Kafka para orquestar y procesar los eventos que desencadenan las funciones serverless.
- Integra gRPC para la comunicación eficiente entre las funciones serverless y otros servicios distribuidos.
- Implementa el algoritmo de Cheney para la recolección de basura en las funciones serverless.

Ejercicio 13: Tolerancia a fallos Bizantina y coordinación de servicios

Desarrolla un sistema de e-commerce que garantice la tolerancia a fallos bizantina y la coordinación efectiva de servicios:

- Implementa un protocolo de Tolerancia a Fallos Bizantina para asegurar que las transacciones se procesen correctamente incluso en presencia de nodos maliciosos.
- Usa Apache Zookeeper para coordinar y gestionar la configuración de servicios distribuidos.
- Integra gRPC para la comunicación de alto rendimiento entre microservicios.
- Utiliza el Algoritmo de Maekawa para la exclusión mutua en la actualización del inventario.

Ejercicio 14: Sincronización de relojes y procesamiento de datos

Diseña un sistema de procesamiento de datos en tiempo real para una red de sensores:

- Implementa el algoritmo de Cristian para sincronizar los relojes de los nodos de sensores con un servidor de tiempo central.
- Usa Apache Kafka para la recolección y procesamiento en tiempo real de los datos de los sensores.
- Integra Apache Spark para el análisis en memoria de los datos recolectados.
- Implementa el algoritmo de marcado y barrido para la gestión de memoria en los nodos de procesamiento de datos.

Ejercicio 15: Coordinación de tareas y exclusión mutua

Crea un sistema de coordinación de tareas en una red de robots industriales:

- Usa el algoritmo de Chandy-Lamport para tomar instantáneas del estado global de los robots durante la ejecución de * tareas.
- Implementa el algoritmo de Raymond para la exclusión mutua en el acceso a recursos compartidos entre los robots.
- Utiliza relojes vectoriales para asegurar el ordenamiento parcial de los eventos y detectar violaciones de causalidad.

- Integra un recolector de basura generacional para la gestión eficiente de la memoria en los nodos de control de los robots.

```
## Tus respuestas
```