# Sincronización y consistencia en computación paralela y distribuida

La computación paralela y distribuida es esencial para maximizar la eficiencia y rendimiento de sistemas informáticos modernos. Dentro de esta área, la sincronización y consistencia son aspectos críticos para asegurar que las operaciones concurrentes se ejecuten correctamente y que los datos se mantengan coherentes.

## Bloqueos (Locks) y técnicas sin bloqueo (Lock-free)

### Bloqueos (locks)

Los bloqueos son una técnica fundamental para manejar la sincronización en sistemas paralelos y distribuidos. Permiten a los procesos acceder a recursos compartidos de manera controlada, previniendo condiciones de carrera y garantizando la coherencia de los datos. Sin embargo, los bloqueos pueden introducir problemas de rendimiento, como la contención de recursos y los deadlocks.

**Tipos de bloqueos:**

1. **Mutex (Mutual Exclusion):** Es el tipo más básico de bloqueo que asegura que solo un hilo pueda acceder a una sección crítica del código a la vez.
2. **Spinlocks:** Un tipo de mutex que emplea un bucle de espera activa para adquirir el bloqueo. Son eficientes en escenarios donde el tiempo de espera es corto.
3. **Readers-Writers Locks:** Permiten múltiples accesos de lectura concurrente pero restringen el acceso de escritura para evitar inconsistencias.

**Problemas de bloqueos:**

- **Contención:** Ocurre cuando múltiples hilos compiten por el mismo bloqueo, lo que puede reducir significativamente el rendimiento.
- **Deadlocks:** Situación en la que dos o más hilos se bloquean mutuamente al esperar indefinidamente por recursos.

### Técnicas sin Bloqueo (Lock-free)

Las técnicas sin bloqueo (lock-free) representan un enfoque avanzado en la computación paralela, orientado a mejorar la escalabilidad y el rendimiento de los sistemas concurrentes. Este enfoque es crucial para aplicaciones donde los bloqueos tradicionales pueden causar cuellos de botella significativos, especialmente en sistemas con alta concurrencia.

**Tipos de técnicas sin bloqueo**

Lock-Free:

- Descripción: Las estructuras de datos lock-free aseguran que al menos un hilo complete su operación en un número finito de pasos, incluso si otros hilos son suspendidos. Esta propiedad evita el problema de la inanición de hilos.

- Implementación: Utiliza primitivas atómicas como Compare-And-Swap (CAS) o Load-Link/Store-Conditional (LL/SC) para asegurar el progreso de al menos un hilo.

Wait-Free:

- Descripción: A diferencia de las técnicas lock-free, las estructuras de datos wait-free garantizan que todos los hilos completarán sus operaciones en un número finito de pasos, proporcionando la mayor garantía de progreso.
- Implementación: Generalmente más complejas que las lock-free, estas técnicas pueden utilizar algoritmos de consenso y estructuras avanzadas de gestión de concurrencia.

Obstruction-Free:

- Descripción: Estas técnicas garantizan el progreso de los hilos solo en ausencia de contención. Es decir, un hilo puede completar su operación sin ser interrumpido por otros hilos.
- Implementación: Aunque es menos robusta que lock-free y wait-free, la obstrucción-free es más fácil de implementar y puede ser útil en ciertos escenarios.

**Implementación de estructuras de datos sin bloqueo**

Compare-And-Swap (CAS):

- Descripción: CAS es una operación atómica que permite comparar el valor actual de una variable con un valor esperado y, si coinciden, cambiar el valor a uno nuevo. Es fundamental en la implementación de estructuras de datos sin bloqueo.
- Ejemplo: Utilizado en pilas sin bloqueo, colas, y listas enlazadas. Un nodo puede ser añadido o removido de una lista enlazada mediante la operación CAS para asegurar que la modificación no se vea interferida por otros hilos.

Load-Link/Store-Conditional (LL/SC):

- Descripción: LL/SC es una pareja de operaciones atómicas que trabajan juntas para evitar algunos problemas de CAS, como el ABA problem. LL lee un valor y SC escribe un nuevo valor solo si ningún otro hilo ha modificado la variable en el ínterin.
- Ejemplo: Utilizado en sistemas que requieren mayor garantía de consistencia y donde CAS puede fallar debido a cambios concurrentes en el valor de la variable.

**Ejemplos de estructuras de datos sin bloqueo

Colas sin bloqueo:

- Descripción: Las colas sin bloqueo permiten múltiples hilos encolar y desencolar elementos simultáneamente sin necesidad de bloqueos.
- Implementación: Utilizan CAS para modificar punteros de cabeza y cola de la cola de manera atómica.

Listas enlazadas sin bloqueo:

- Descripción: Permiten inserciones y eliminaciones concurrentes en una lista enlazada.
- Implementación: Utilizan CAS para modificar los punteros de los nodos.

Pilas sin bloqueo:

- Descripción: Permiten operaciones push y pop concurrentes en una pila.
- Implementación: CAS se usa para asegurar que la parte superior de la pila se modifique de manera segura.

## Ejercicios

1. Describe las diferencias entre técnicas lock-free, wait-free y obstruction-free. Proporciona ejemplos de situaciones en las que cada una sería más adecuada.
2. Discute las ventajas y desventajas de utilizar técnicas sin bloqueo en sistemas concurrentes.
3. Explica el problema ABA en el contexto de las técnicas sin bloqueo. ¿Cómo afecta a la operación CAS?
4. Investiga cómo las operaciones LL/SC abordan el problema ABA.
5. Define y explica la operación Compare-And-Swap (CAS). ¿Por qué es crucial en la implementación de estructuras de datos sin bloqueo?
6. Describe las operaciones Load-Link (LL) y Store-Conditional (SC). ¿Cómo funcionan juntas para asegurar la consistencia?

7 . Implementa una pila sin bloqueo usando la operación CAS. Utiliza el módulo multiprocessing para simular hilos concurrentes.

```
import threading
import time
from typing import Optional

class Node:
    def __init__(self, value: int):
        self.value = value
        self.next = None

class LockFreeStack:
    def __init__(self):
        self.top: Optional[Node] = None
        self.lock = threading.Lock()

    def cas(self, old, new):
        with self.lock:
            if self.top == old:
                self.top = new
                return True
            return False

    def push(self, value: int):
        new_node = Node(value)
        while True:
            old_top = self.top
            new_node.next = old_top
            if self.cas(old_top, new_node):
                return
```

```
    def pop(self) -> Optional[int]:
        while True:
            old_top = self.top
            if old_top is None:
                return None
            new_top = old_top.next
            if self.cas(old_top, new_top):
                return old_top.value

stack = LockFreeStack()

def worker_push():
    for i in range(100):
        stack.push(i)
        time.sleep(0.01)

def worker_pop():
    for _ in range(100):
        print(stack.pop())
        time.sleep(0.01)

threads = []
for _ in range(2):
    t1 = threading.Thread(target=worker_push)
    t2 = threading.Thread(target=worker_pop)
    threads.append(t1)
    threads.append(t2)
    t1.start()
    t2.start()

for t in threads:
    t.join()
```

8 . Implementa una cola sin bloqueo usando CAS para modificar los punteros de cabeza y cola. Simula la concurrencia utilizando threading.

```
import threading
import time
from typing import Optional

class Node:
    def __init__(self, value: int):
        self.value = value
        self.next = None

class LockFreeQueue:
    def __init__(self):
        self.head: Optional[Node] = Node(None)
        self.tail: Optional[Node] = self.head
        self.lock = threading.Lock()
```

```
    def cas(self, attr, old, new):
        with self.lock:
            if getattr(self, attr) == old:
                setattr(self, attr, new)
                return True
            return False

    def enqueue(self, value: int):
        new_node = Node(value)
        while True:
            old_tail = self.tail
            next_tail = old_tail.next
            if old_tail == self.tail:
                if next_tail is None:
                    if self.cas('tail.next', next_tail, new_node):
                        self.cas('tail', old_tail, new_node)
                        return
                else:
                    self.cas('tail', old_tail, next_tail)

    def dequeue(self) -> Optional[int]:
        while True:
            old_head = self.head
            old_tail = self.tail
            next_head = old_head.next
            if old_head == self.head:
                if old_head == old_tail:
                    if next_head is None:
                        return None
                    self.cas('tail', old_tail, next_head)
                else:
                    value = next_head.value
                    if self.cas('head', old_head, next_head):
                        return value

queue = LockFreeQueue()

def worker_enqueue():
    for i in range(100):
        queue.enqueue(i)
        time.sleep(0.01)

def worker_dequeue():
    for _ in range(100):
        print(queue.dequeue())
        time.sleep(0.01)

threads = []
for _ in range(2):
```

```
    t1 = threading.Thread(target=worker_enqueue)
    t2 = threading.Thread(target=worker_dequeue)
    threads.append(t1)
    threads.append(t2)
    t1.start()
    t2.start()

for t in threads:
    t.join()
```

9 . Implementa una simulación de las operaciones Load-Link (LL) y Store-Conditional (SC) en Python para una lista enlazada sin bloqueo.

```
import threading
import time

class Node:
    def __init__(self, value: int):
        self.value = value
        self.next = None

class LockFreeLinkedList:
    def __init__(self):
        self.head: Optional[Node] = None
        self.lock = threading.Lock()

    def load_link(self, node):
        return node

    def store_conditional(self, old_node, new_node):
        with self.lock:
            if self.head == old_node:
                self.head = new_node
                return True
            return False

    def insert(self, value: int):
        new_node = Node(value)
        while True:
            old_head = self.load_link(self.head)
            new_node.next = old_head
            if self.store_conditional(old_head, new_node):
                return

    def remove(self) -> Optional[int]:
        while True:
            old_head = self.load_link(self.head)
            if old_head is None:
                return None
            new_head = old_head.next
```

```
            if self.store_conditional(old_head, new_head):
                return old_head.value

linked_list = LockFreeLinkedList()

def worker_insert():
    for i in range(100):
        linked_list.insert(i)
        time.sleep(0.01)

def worker_remove():
    for _ in range(100):
        print(linked_list.remove())
        time.sleep(0.01)

threads = []
for _ in range(2):
    t1 = threading.Thread(target=worker_insert)
    t2 = threading.Thread(target=worker_remove)
    threads.append(t1)
    threads.append(t2)
    t1.start()
    t2.start()

for t in threads:
    t.join()
```

1. Investiga y describe los principales desafíos en la implementación de estructuras de datos lock-free. ¿Cómo afectan estos desafíos a la complejidad del código y al rendimiento del sistema?

2. Propón un marco teórico para evaluar el rendimiento de una estructura de datos lock-free frente a una estructura de datos tradicional con bloqueo. ¿Qué métricas usarías y por qué?

3. Analiza cómo las técnicas sin bloqueo pueden mejorar la escalabilidad de una aplicación. Proporciona ejemplos concretos de aplicaciones o sistemas que se benefician de estas técnicas.

13 . Implementa un contador lock-free utilizando CAS para asegurar la atomicidad de las operaciones de incremento y decremento.

```
## Tu respuesta
```

1. Implementa una lista enlazada wait-free utilizando una técnica como el algoritmo de consenso para garantizar que todas las operaciones se completen en un número finito de pasos.

```
## Tu respuesta
```

1. Implementa una cola de prioridad sin bloqueo utilizando CAS y otras primitivas atómicas. La cola debe permitir la inserción y extracción de elementos basados en su prioridad.

```
## Tu respuesta
```

# Consistencia eventual vs. consistencia fuerte

## Consistencia eventual

La consistencia eventual es un modelo de consistencia en sistemas distribuidos donde, en ausencia de nuevas actualizaciones, todos los nodos finalmente alcanzarán un estado consistente. Este modelo es más flexible y escalable, permitiendo una mayor disponibilidad y particionamiento en el sistema, a expensas de una coherencia inmediata.

**Ventajas de la consistencia eventual:**

- **Alta disponibilidad:** Permite que los nodos respondan a las operaciones de lectura/escritura incluso si están temporalmente desincronizados.
- **Escalabilidad:** Reduce la necesidad de comunicación constante entre nodos, lo que mejora la escalabilidad del sistema.

**Desventajas:**

- **Complejidad en la gestión de concurrencia:** Requiere mecanismos adicionales para resolver conflictos de datos.
- **Incertidumbre temporal:** No garantiza cuándo exactamente los datos se volverán consistentes.

## Consistencia fuerte

La consistencia fuerte garantiza que todas las operaciones de lectura de datos devuelvan el resultado más reciente de una operación de escritura, proporcionando una visión inmediata y consistente del sistema.

**Ventajas de la consistencia fuerte:**

- **Simplicidad para los desarrolladores:** Facilita la lógica de programación al asegurar que los datos leídos son siempre los más recientes.
- **Coherencia inmediata:** Ideal para aplicaciones que requieren datos altamente consistentes en tiempo real.

**Desventajas:**

- **Rendimiento y Latencia:** Puede introducir una latencia significativa debido a la necesidad de coordinación entre nodos.
- **Escalabilidad Limitada:** La necesidad de sincronización constante entre nodos puede afectar la escalabilidad del sistema.

# Ejercicios

1. Describe las diferencias clave entre consistencia eventual y consistencia fuerte. Proporciona ejemplos de aplicaciones donde cada modelo sería más adecuado.

2. Discute cómo la CAP theorem (Teorema de CAP) influye en la elección entre consistencia eventual y consistencia fuerte.
3. Explica los desafíos técnicos en la implementación de consistencia eventual en un sistema distribuido. ¿Cómo se pueden resolver los conflictos de datos?
4. Describe los problemas de latencia y rendimiento asociados con la consistencia fuerte. ¿Qué técnicas se pueden utilizar para mitigar estos problemas?
5. Proporciona un análisis detallado de un caso de uso real donde se utilice consistencia eventual y otro donde se utilice consistencia fuerte. ¿Por qué se eligió cada modelo en estos casos?

1. Implementa una simulación simple de consistencia eventual en un sistema distribuido. Usa múltiples nodos que eventualmente alcanzan un estado consistente.

```
import threading
import time
import random

class Node:
    def __init__(self, name):
        self.name = name
        self.data = {}
        self.lock = threading.Lock()

    def update(self, key, value):
        with self.lock:
            self.data[key] = value

    def get(self, key):
        with self.lock:
            return self.data.get(key)

    def synchronize(self, other_node):
        with self.lock, other_node.lock:
            for key, value in other_node.data.items():
                self.data[key] = value

def simulate_consistency_eventual(nodes, key, value):
    chosen_node = random.choice(nodes)
    chosen_node.update(key, value)
    print(f"Nodo {chosen_node.name} actualizado {key} a {value}")

    for _ in range(10):  # Simulate eventual consistency
        time.sleep(random.uniform(0.1, 0.5))
        node1, node2 = random.sample(nodes, 2)
        node1.synchronize(node2)
        print(f"Nodo {node1.name} y {node2.name} sincronizado")

nodes = [Node(f"Nodo{i}") for i in range(5)]

threads = [threading.Thread(target=simulate_consistency_eventual,
args=(nodes, 'key1', i)) for i in range(3)]
```

```
for t in threads:
    t.start()

for t in threads:
    t.join()

for node in nodes:
    print(f"{node.name} data: {node.data}")
```

7 . Implementa una simulación simple de consistencia fuerte en un sistema distribuido. Usa múltiples nodos y asegura que las lecturas siempre reflejen la escritura más reciente.

```
import threading

class StrongConsistentNode:
    def __init__(self, name, central_storage):
        self.name = name
        self.central_storage = central_storage

    def update(self, key, value):
        self.central_storage.update(key, value)
        print(f"Nodo {self.name} actualizado {key} a {value}")

    def get(self, key):
        value = self.central_storage.get(key)
        print(f"Nodo {self.name} se lee{key} como {value}")
        return value

class CentralStorage:
    def __init__(self):
        self.data = {}
        self.lock = threading.Lock()

    def update(self, key, value):
        with self.lock:
            self.data[key] = value

    def get(self, key):
        with self.lock:
            return self.data.get(key)

central_storage = CentralStorage()
nodes = [StrongConsistentNode(f"Nodo{i}", central_storage) for i in
range(5)]

def simulate_consistency_strong(node, key, value):
    node.update(key, value)
    node.get(key)
```

```
threads = [threading.Thread(target=simulate_consistency_strong,
args=(nodes[i], 'key1', i)) for i in range(3)]

for t in threads:
    t.start()

for t in threads:
    t.join()

for node in nodes:
    print(f"{node.name} data: {node.central_storage.data}")
```

8 . Implementa un script que prueba la escalabilidad de un sistema con consistencia eventual versus consistencia fuerte, midiendo el tiempo de respuesta a medida que aumentan los nodos.

```
## Tu respuesta
```

## Algoritmos de consenso

Los algoritmos de consenso son esenciales para lograr consistencia en sistemas distribuidos. Garantizan que un grupo de nodos acuerden un valor común, a pesar de fallos y latencias en la red. Dos de los algoritmos más conocidos son Paxos y Raft.

### Paxos

Paxos es un algoritmo de consenso altamente robusto diseñado por Leslie Lamport. Es conocido por su capacidad para tolerar fallos y su formalismo riguroso.

**Componentes principales de Paxos:**

1. **Proposers (Proponentes):** Proponen valores a ser aceptados.
2. **Acceptors (Aceptantes):** Votan por los valores propuestos.
3. **Learners (Aprendices):** Aprenden el valor acordado.

**Fases del algoritmo Paxos:**

1. Fase de preparación: Un proposer elige un número de propuesta y solicita a los acceptors prometer no aceptar propuestas anteriores a ese número.
2. Fase de aceptación: Si los acceptors responden con promesas, el proposer envía una propuesta con el número elegido. Los acceptors entonces aceptan la propuesta si no han prometido aceptar otra mayor.
3. Fase de aprendizaje: Los learners reciben el valor acordado y lo consideran definitivo una vez que una mayoría de acceptors lo ha aceptado.

**Desventajas de Paxos:**

- **Complejidad:** El protocolo es complejo y difícil de implementar correctamente.
- **Latencia:** Puede ser ineficiente debido a la necesidad de múltiples rondas de comunicación.

- **No garantiza progresión:** En ciertas condiciones de red, es posible que no se logre un consenso.

## Raft

Raft es un algoritmo de consenso diseñado para ser más comprensible y fácil de implementar que Paxos. Divide el problema de consenso en componentes más manejables y sigue un enfoque de líder único.

**Componentes principales de Raft:**

1. **Líder:** Responsable de gestionar las entradas de log y replicarlas a los seguidores.
2. **Seguidores:** Replican el log del líder y responden a sus solicitudes.
3. **Candidatos:** Nodos que intentan convertirse en líderes a través de elecciones.

**Fases del algoritmo raft:**

1. **Elección de líder:** Los nodos se convierten en candidatos y solicitan votos de otros nodos. El candidato que recibe la mayoría de votos se convierte en el líder.
2. **Replicación de log:** El líder recibe entradas de log y las envía a los seguidores. Una entrada se considera comprometida cuando la mayoría de los seguidores la han replicado.
3. **Aplicación de entradas:** Una vez comprometidas, las entradas de log se aplican a la máquina de estado.

**Ventajas de Raft:**

- **Simplicidad y comprensibilidad:** Su diseño modular y claro lo hace más fácil de entender e implementar.
- **Eficiencia en elección de líder:** Reduce la latencia en el proceso de elección de líder.

**Desventajas:**

- **Limitaciones de escalabilidad:** Puede no ser tan eficiente en sistemas extremadamente grandes debido a su enfoque centralizado en el líder.
- **Single Point of Failure:** La caída del líder puede causar demoras hasta que se elija un nuevo líder.

## Algoritmo de consenso Bizantino (BFT)

El algoritmo de tolerancia a fallos Bizantinos (BFT) se utiliza para lograr consenso en sistemas donde los nodos pueden fallar de manera arbitraria o maliciosa. Es común en sistemas críticos de alta seguridad.

**Componentes principales del algoritmo BFT:**

- **Nodos participantes:** Todos los nodos deben comunicarse entre sí para garantizar la integridad del consenso.
- ** Rondas de consenso:** Se realizan múltiples rondas de votación para acordar un valor común.

**Fases del algoritmo BFT:**

- **Fase de propuesta:** Un nodo líder propone un valor.
- **Fase de preparación:** Todos los nodos envían y reciben mensajes de preparación para validar la propuesta.
- **Fase de compromiso:** Si un nodo recibe suficientes respuestas de preparación, envía un mensaje de compromiso.
- **Fase de decisión:** Una vez que se alcanza un consenso, todos los nodos acuerdan el valor final.

**Ventajas de BFT:**

- **Alta resiliencia:** Tolera fallos bizantinos y ataques maliciosos.
- **Fiabilidad:** Garantiza que se alcance un consenso correcto incluso en presencia de nodos defectuosos.

**Desventajas:**

- **Complejidad y coste computacional:** Requiere múltiples rondas de comunicación y puede ser costoso en términos de tiempo y recursos.
- **Escalabilidad:** La sobrecarga de comunicación puede limitar su escalabilidad en grandes redes.

## Algoritmo de consenso de Nakamoto (Blockchain)

El algoritmo de consenso de Nakamoto, utilizado en blockchain, es conocido por su aplicación en criptomonedas como Bitcoin. Utiliza un enfoque de prueba de trabajo (Proof of Work).

**Componentes principales del algoritmo de Nakamoto:**

- **Mineros:** Compiten para resolver problemas criptográficos complejos.
- **Bloques:** Los mineros crean bloques de transacciones.
- **Cadena de bloques (Blockchain):** Cada bloque está vinculado al anterior, formando una cadena segura.

**Fases del algoritmo de Nakamoto:**

- **Propuesta de bloque:** Los mineros recopilan transacciones y proponen nuevos bloques.
- **Prueba de trabajo:** Los mineros resuelven un problema criptográfico para validar el bloque.
- **Verificación y adición:** El bloque validado se añade a la cadena y se propaga a todos los nodos de la red.

**Ventajas del algoritmo de Nakamoto:**

- Descentralización: No requiere una autoridad central, lo que aumenta la resistencia a la censura.
- Seguridad: La Prueba de Trabajo proporciona una fuerte protección contra ataques.

**Desventajas:**

- Consumo de energía: La Prueba de Trabajo requiere una cantidad significativa de energía.

- Latencia: La validación de bloques puede introducir retrasos en la confirmación de transacciones.

## Ejercicios

1. Explica el proceso de consenso de Paxos. Describe las fases de preparación y aceptación, y cómo los proposers, acceptors y learners interactúan para alcanzar un consenso.
2. Analiza las ventajas y desventajas de Paxos en un entorno distribuido con alta latencia de red. ¿Cómo afecta la latencia a la eficiencia del algoritmo?
3. Implementa una versión simplificada del algoritmo Paxos en Python. Crea un simulador con tres proposers, cinco acceptors y dos learners. Simula un escenario en el que los proposers intentan alcanzar un consenso sobre un valor.
4. Modifica la implementación anterior para incluir fallos de nodos (por ejemplo, proposers y acceptors que no responden) y observa cómo afecta al proceso de consenso. ¿Cómo maneja Paxos estos fallos?
5. Describe las fases del algoritmo Raft: elección de líder, replicación de log y aplicación de entradas. ¿Cómo garantiza Raft la consistencia del log replicado entre los seguidores?
6. Compara y contrasta Paxos y Raft en términos de simplicidad, eficiencia y capacidad de recuperación ante fallos. ¿Por qué podría ser preferible usar Raft en ciertas aplicaciones?
7. Implementa un simulador de Raft en Python. Incluye la lógica para la elección de líder, replicación de log y aplicación de entradas en una máquina de estado. Simula un cluster de cinco nodos.
8. Añade soporte para el manejo de particiones de red en tu simulador de Raft. Implementa un escenario donde la red se divide en dos particiones, y observa cómo Raft maneja la elección de líder y la replicación de log durante y después de la partición.
9. Explica cómo el algoritmo de tolerancia a Ffallos Bizantinos (BFT) logra consenso en presencia de nodos maliciosos. ¿Qué garantías proporciona en términos de seguridad y fiabilidad?
10. Discute las implicaciones de usar un Algoritmo BFT en un entorno distribuido con más de un tercio de los nodos potencialmente fallando de manera arbitraria. ¿Cómo afecta esto a la robustez del algoritmo?
11. Implementa un algoritmo BFT simplificado en Python. Crea un sistema con siete nodos donde hasta dos pueden fallar de manera arbitraria. Simula un proceso de consenso en presencia de estos fallos.
12. Modifica la implementación anterior para incluir ataques maliciosos, como la inyección de datos incorrectos y la falsificación de respuestas. Analiza cómo el algoritmo maneja y mitiga estos ataques.
13. Describe el proceso de consenso utilizado en blockchain (algoritmo de Nakamoto) con énfasis en la prueba de trabajo (Proof of Work). ¿Cómo garantiza la seguridad y la integridad de la cadena de bloques?
14. Analiza las desventajas del Algoritmo de Nakamoto en términos de consumo de energía y latencia de transacciones. ¿Qué alternativas existen para mitigar estos problemas?
15. Implementa una versión simplificada de un blockchain en Python, incluyendo la lógica para la creación de bloques, la prueba de trabajo y la validación de transacciones. Simula la minería de bloques en una red distribuida.

16. Extiende la implementación anterior para incluir múltiples nodos que compiten por minar bloques. Implementa un mecanismo para la resolución de bifurcaciones en la cadena de bloques y observa cómo la red alcanza un consenso.
17. [Opcional]. Investiga y explica el algoritmo de consenso de Practical Byzantine Fault Tolerance (PBFT). ¿En qué se diferencia de los algoritmos tradicionales de BFT y en qué escenarios es más adecuado?
18. [Opcional]Describe el algoritmo de consenso Delegated Proof of Stake (DPoS). ¿Cómo maneja la elección de nodos validadores y qué ventajas ofrece sobre la Prueba de Trabajo (PoW)?
19. [Opcional] Implementa un simulador de PBFT en Python. Crea un sistema con cuatro nodos donde uno puede fallar de manera arbitraria. Simula un proceso de consenso y analiza el rendimiento del algoritmo.
20. [Opcional] Implementa un sistema basado en DPoS en Python. Incluye la lógica para la elección de delegados, la validación de transacciones y la creación de bloques. Simula un escenario donde los delegados son seleccionados y reemplazados dinámicamente.

```
## Tus respuestas
```