Universidad Nacional Autónoma de México

Facultad de Ciencias

Computación Distribuida - 7176

P R O Y E C T O

Equipo:

Ángeles Sánchez Aldo Javier - 320286144

Sánchez Victoria Leslie Paola - 320170513

Sosa Romo Juan Mario - 320051926

Fecha de entrega:
29 de Noviembre de 2024

Profesor:
Mat. Salvador López Mendoza

Ayudante:
Santiago Arroyo Lozano

Contents

Chapter 1

Planteamiento del problema

Este documento presenta la solución final al proyecto de la materia de Computación Distribuida. El objetivo de este trabajo es desarrollar una blockchain para un sistema de criptomonedas, implementado en el lenguaje de programación *Elixir*.

Durante la ejecución del proyecto, se simulará el sistema de criptomonedas gestionando los distintos procesos distribuidos que representan a los usuarios. Estos usuarios realizarán transacciones entre ellos, enviando mensajes para alcanzar un consenso y detectando y eliminando intentos de alteración de la cadena de bloques. La blockchain, aunque simula un entorno, debe operar de forma descentralizada y garantizar la integridad de la información, asegurando que los nodos honestos mantengan una copia consistente de las transacciones para que estas se validen e inserten correctamente.

Para modelar las conexiones de la red, se utiliza una implementación basada en el modelo *Watts y Strogatz*, que garantiza un coeficiente de agrupamiento alto y una estructura que favorece la propagación eficiente de la información. Se asegura que este coeficiente sea mayor a 0.4 para comenzar el proceso de consenso y transmisión de transacciones.

Una de las características importantes del proyecto es la introducción de **procesos bizantinos**, los cuales representan nodos maliciosos en la red. Estos procesos se encargan de generar bloques basura o incorrectos con la intención de alterar la blockchain y comprometer su integridad. Sin embargo, el sistema debe ser capaz de detectar y evitar que estos bloques maliciosos sean insertados en la cadena. Para garantizar la seguridad y la correcta operación de la red, se asume que, si hay f nodos bizantinos y n nodos en total, se cumple la condición $n > 3f$, lo que asegura que la mayoría de los nodos honestos puedan mantener la coherencia de la blockchain.

Consideraciones extra

Se pueden utilizar cuantas funciones auxiliares sean necesarias, siempre que se logre el objetivo deseado. No es necesario implementar la persistencia del estado o la validación de la cartera; una vez que el programa termina, la blockchain deja de existir, y no se almacenará nada de forma permanente. No se realizará ninguna verificación sobre la validez de las transacciones en cuanto a saldo disponible. Se proporcionará un módulo inicial llamado *Crypto* encargado de realizar los hasheos de los bloques. Finalmente, la red debe ser capaz de evitar la inserción de bloques generados por procesos bizantinos, manteniendo así la integridad del sistema.

Chapter 2

Solución propuesta

2.1 Flujo general de la solución

1. **Inicialización de procesos:**
   - La función `Main.run(10, 1)` inicializa:
     - Una red de 10 nodos, representados como procesos de Elixir.
     - La blockchain, que comienza con un bloque génesis común a todos los nodos.
2. **Construcción de la red:**
   - Los procesos se conectan entre sí utilizando el modelo Watts y Strogatz, garantizando un coeficiente de agrupación mayor a 0.4.
   - Cada nodo guarda una copia inicial de la blockchain, comenzando con el bloque génesis.
3. **Propuesta de un nuevo bloque:**
   - Un nodo selecciona o recibe la tarea de proponer un bloque nuevo.
   - El nodo genera el bloque con la siguiente información:
     - **data:** El contenido del bloque, por ahora un string.
     - **timestamp:** La marca de tiempo de la creación del bloque.
     - **previous_hash:** El hash del bloque anterior en la blockchain.
     - **hash:** El hash del bloque actual.
   - El bloque es transmitido a todos los nodos de la red, esto es, se envia el mensaje a los vecinos, valida el bloque y si es correcto lo propaga a sus vecinos. El mensaje se transmite incluyendo el identificador del nodo que lo propuso para que pueda recopilar los votos.
4. **Proceso de votación:**
   - Cada nodo recibe el bloque propuesto y realiza las siguientes verificaciones:
     - ¿El data del bloque es un string?
     - ¿El hash del bloque anterior coincide con su propia blockchain?
     - ¿El hash del bloque actual es válido?
     - ¿El timestamp del bloque es posterior al último bloque de la blockchain?
   - Con base en las verificaciones, el nodo responde con un voto:

- `ACEPTAR` si el bloque es válido.
- `RECHAZAR` si el bloque es inválido.

5. **Decisión de consenso:**

   - El nodo proponente recolecta los votos de todos los nodos; para esto, los nodos checan el identificador del nodo que propuso el bloque, si es vecino inmediato se envía el voto, si no se reenvía a los vecinos, estos recopilan los votos y los envían al nodo proponente, vamos a usar TTL para eviar que los mensajes se propaguen indefinidamente.
   - Si más del 50% de los votos son `ACEPTAR`, el bloque es considerado válido:
     - El bloque se agrega a la blockchain del nodo proponente.
     - El bloque es propagado a todos los nodos para que lo agreguen a sus respectivas blockchains, se usa un mensaje diferente al de proponer.
   - Si no se alcanza el consenso, el bloque es descartado.

6. **Interacción desde el evaluador:**

   - El evaluador puede utilizar la terminal interactiva de Elixir (`iex`) para:
     - Insertar bloques nuevos.
     - Verificar que las blockchains de todos los nodos son consistentes y reflejan los mismos bloques.-

2.2 Módulos y sus funciones

Archivo Blockchain.ex:

Módulo `Block`

`new/2`

`new(data, prev_hash)`

Crea un nuevo bloque con los datos proporcionados (`data`) y el hash del bloque previo (`prev_hash`).

$$\text{timestamp} = \text{DateTime.utc\_now}() \rightarrow \text{Fecha y hora actuales en formato UTC.}$$

Se genera un bloque inicial con un hash vacío y luego se calcula el hash del bloque usando el módulo `Crypto`:

$$\text{block.hash} = \text{Crypto.put\_hash(block)}$$

`valid?/1`

`valid?(%Block{})`

Valida que un bloque sea correcto verificando si el hash almacenado coincide con el hash calculado a partir de los datos del bloque.

$$\text{valid} = \text{block.hash} == \text{Crypto.hash(block)}$$

`valid?/2`

`valid?(%Block{}, %Block{})`

Valida que dos bloques consecutivos sean correctos. Comprueba lo siguiente:

- El `prev_hash` del segundo bloque coincide con el `hash` del primer bloque.
- El orden temporal entre los bloques es correcto: `block1.timestamp` $\leq$ `block2.timestamp`.
- Ambos bloques son válidos individualmente: `Block.valid?(block1)` y `Block.valid?(block2)`.

Módulo `Blockchain`

`new_genesis_block/0`

`new_genesis_block()`

Crea el bloque génesis, que es el primer bloque de la blockchain. Este bloque contiene:

- `data`: "Genesis Block".
- `timestamp`: "2024-01-01 00:00:00Z".
- `prev_hash`: "0".

El hash se calcula con `Crypto.put_hash(block)`.

`new/0`

`new()`

Inicializa una nueva blockchain con el bloque génesis. Retorna una estructura de tipo `Blockchain`:

$$\text{blockchain} = \%Blockchain\{\text{chain} : [\text{genesis\_block}]\}$$

`valid?/1`

`valid?(%Blockchain{})`

Valida toda la blockchain asegurándose de que:

- El hash de cada bloque coincide con el calculado para ese bloque.
- El `prev_hash` de cada bloque coincide con el `hash` del bloque anterior.
- Los bloques están ordenados cronológicamente.

Utiliza `Enum.chunk_every(2, 1, :discard)` para generar pares consecutivos de bloques, y luego verifica la validez de cada par con `Block.valid?/2`.

`insert/2`

`insert(%Block{}, %Blockchain{})`

Inserta un nuevo bloque en la blockchain. Verifica que:

- Existe un bloque previo (`List.last(chain)`).
- El nuevo bloque es válido en relación con el último bloque: `Block.valid?(last_block, new_block)`.

Si todo está en orden, entonces el nuevo bloque se añade a la cadena:

$$\text{updated\_chain} = \text{chain} + +[\text{new\_block}]$$

Si no está en orden, retorna un error indicando que el bloque no es válido.

Archivo Grafica.ex:

Módulo `Grafica`

`inicia/1`

`inicia(estado_inicial \\ %{})`

Inicializa un nodo con un estado inicial, que incluye:

- `vecinos`: lista de otros nodos con los que se comunica.
- `blockchain`: blockchain actual del nodo (inicializada con `Blockchain.new()`).
- `bizantino`: indica si el nodo es malicioso (`true`) o no (`false`).
- `mensajes`: estructura para almacenar mensajes de tipo `prepare` y `commit`.

El nodo llama a `recibe_mensaje/1` para comenzar a procesar mensajes.

`recibe_mensaje/1`

`recibe_mensaje(estado)`

Espera y procesa mensajes recibidos de otros nodos. Llama a `procesa_mensaje/2` para manejar cada mensaje y actualiza el estado del nodo.

`procesa_mensaje/2`

`procesa_mensaje(mensaje, estado)`

Procesa diferentes tipos de mensajes. Los cuales son:

`Estado` Muestra el estado actual del nodo y reinicia los mensajes.

$$\text{estado} = \text{reiniciar\_mensajes\_y\_vistos}(\text{estado})$$

`Vecinos` Asigna una lista de vecinos al nodo y actualiza su estado.

$$\text{estado} = \text{Map.put(estado, :vecinos, vecinos)}$$

`Bloque` Envía un bloque a los vecinos. Si el nodo es bizantino, envía un bloque modificado con datos corruptos.

`Preprepare` Procesa un mensaje `preprepare`. Si el bloque es válido, envía un mensaje `prepare` a los vecinos.

`Prepare` Procesa un mensaje `prepare`. Si el nodo no ha visto el mensaje antes, lo agrega a sus mensajes y verifica si se alcanza un cuórum. Si el cuórum es suficiente, envía un mensaje `commit`.

`Commit` Procesa un mensaje `commit`. Si el nodo no ha visto el mensaje antes, lo agrega a la blockchain una vez alcanzado el cuórum.

`reiniciar_mensajes_y_vistos/1`

`reiniciar_mensajes_y_vistos(estado)`

Reinicia los mensajes (`prepare` y `commit`) y limpia la lista de mensajes vistos.

$$\text{estado} = \text{Map.put(estado, :mensajes, \%:prepare => [], :commit => [])}$$

`actualizar_mensajes/3`

`actualizar_mensajes(estado, tipo, mensaje)`

Agrega un mensaje a la lista correspondiente (`prepare` o `commit`).

$$\text{mensajes\_actualizados} = \text{Map.update(estado[:mensajes], tipo, [mensaje], [mensaje | \&1])}$$

`suficiente_cuorum?/2`

`suficiente_cuorum?(estado, tipo)`

Verifica si se alcanzó un cuórum para un tipo de mensaje (`prepare` o `commit`). Calcula un umbral basado en la cantidad de nodos:

$$\text{threshold} = \left\lfloor \frac{2 \cdot (\text{total\_nodos})}{3} \right\rfloor + 1$$

Retorna `true` si el número de mensajes únicos es suficiente para alcanzar el cuórum.

`agregar_bloque/2`

`agregar_bloque(bloque, estado)`

Agrega un bloque a la blockchain del nodo si es válido. Si ocurre un error, mantiene el estado actual.

`mensaje_visto?/2`

`mensaje_visto?(estado, mensaje)`

Verifica si un mensaje ya ha sido visto por el nodo. Utiliza un conjunto (`MapSet`) para comprobar duplicados.

`marcar_mensaje_visto/2`

`marcar_mensaje_visto(estado, mensaje)`

Marca un mensaje como visto y lo añade al conjunto `mensajes_vistos`.

Archivo Crypto.ex:

Módulo `Crypto`

El módulo `Crypto` asegura la integridad de los bloques en una blockchain.

Atributo de Módulo

`@block_fields [:data, :timestamp, :prev_hash]`

Define los campos del bloque (`data`, `timestamp` y `prev_hash`) que serán utilizados para calcular su hash.

`hash/1`

`hash(%{} = block)`

Calcula el hash criptográfico de un bloque.

Parámetros

- `block`: Un mapa que representa un bloque, utilizando `@block_fields`.

Retorno

- Una cadena que representa el hash calculado para el bloque.

`put_hash/1`

`put_hash(%{} = block)`

Calcula e inserta el hash en un bloque.

Parámetros

- `block`: Un mapa que representa un bloque.

Retorno

- El bloque actualizado con un nuevo campo `:hash`, que contiene el hash calculado.

Funcionamiento

1. Llama a la función `hash/1` para calcular el hash del bloque.
2. Devuelve el bloque con el campo `:hash` actualizado.

`encode_to_binary/1`

`encode_to_binary(map)`

Convierte un mapa en una representación binaria.

Parámetros

- `map`: Un mapa cuyos valores serán convertidos a binario.

Retorno

- Una lista de caracteres binarios que representan el mapa.

Funcionamiento

1. Convierte cada valor del mapa a binario mediante la función `to_binary/1`.
2. Une los valores binarios en una cadena.
3. Convierte la cadena a una lista binaria con `:erlang.binary_to_list/1`.

`to_binary/1`

`to_binary(value)`

Convierte un valor individual en su representación binaria.

Parámetros

- `value`: Un valor de cualquier tipo.

Retorno

- Una representación binaria del valor. Si ya es binario, se retorna tal cual; de lo contrario, se convierte usando `inspect/1`.

`simple_hash/1`

`simple_hash(binary)`

Calcula un hash sencillo a partir de una cadena binaria.

Parámetros

- `binary`: Una cadena binaria de entrada.

Retorno

- Una cadena hexadecimal que representa el hash calculado.

Funcionamiento

1. Utiliza la función `:erlang.phash2/1` para calcular el hash de la cadena.
2. Convierte el valor resultante a una cadena hexadecimal con `Integer.to_string/2`.

Archivo Main.ex

Módulo `Main`

Este módulo controla la creación de una red de nodos utilizando el modelo de Watts y Strogatz. Simula una blockchain distribuida donde los nodos se comunican y llegan a consenso mediante el algoritmo PBFT.

`run/2`

`run(n, f)`

Inicializa una red de nodos, simula el consenso enviando bloques válidos e inválidos y verifica los estados finales.

Parámetros:

- `n`: Número total de nodos en la red.
- `f`: Número de nodos bizantinos en la red.

Retorno:

- Lista de procesos de nodos creados.

Funcionamiento:

1. Crea nodos (honestos y bizantinos) mediante `crea_nodos/2`.
2. Asigna vecinos a los nodos utilizando el modelo de Watts y Strogatz en `asigna_vecinos/1`.
3. Simula el envío y la propagación de bloques válidos e inválidos.
4. Verifica el estado final de los nodos.

`crea_nodos/2`

`crea_nodos(n, f)`

Crea una lista de nodos que incluye:

- `f` nodos bizantinos creados con `NodoBizantino.inicia/0`.
- `n-f` nodos honestos creados con `NodoHonesto.inicia/0`.

Retorno:

- Lista de procesos de nodos creados.

`asigna_vecinos/1`

`asigna_vecinos(procesos)`

Asigna vecinos a los nodos de la red, utilizando un modelo en anillo con probabilidades de reconexión aleatoria.

Funcionamiento:

1. Asigna a cada nodo `k` vecinos iniciales en un anillo.
2. Introduce conexiones aleatorias con probabilidad de reenlace `0.1`.
3. Verifica que el coeficiente de agrupamiento sea mayor a `0.4`.

Retorno: Ninguno. Envía mensajes con los vecinos asignados a cada nodo.

`clustering_coefficient/2`

`clustering_coefficient(vecinos, procesos)`

Calcula el coeficiente de agrupamiento de la red.

Parámetros:

- `vecinos`: Lista de listas que representa los vecinos de cada nodo.
- `procesos`: Lista de procesos de los nodos.

Retorno:

- El coeficiente de agrupamiento (promedio del número de triángulos formados por los nodos).

—

Módulo `NodoHonesto`

Simula un nodo honesto que participa de manera no maliciosa en la red.

`inicia/0`

`inicia()`

Inicia un nodo honesto con un estado inicial:

- `vecinos`: Lista de vecinos asignados.
- `blockchain`: Blockchain inicializada.

- `bizantino`: `false`.
- `mensajes`: Diccionario vacío para almacenar mensajes `prepare` y `commit`.

Llama a `Grafica.inicia/1` para iniciar el nodo.
—

Módulo `NodoBizantino`

Simula un nodo bizantino que actúa de manera maliciosa en la red.

`inicia/0`

`inicia()`

Inicia un nodo bizantino con un estado inicial:

- `vecinos`: Lista de vecinos asignados.
- `blockchain`: Blockchain inicializada.
- `bizantino`: `true`.
- `mensajes`: Diccionario vacío para almacenar mensajes `prepare` y `commit`.

Llama a `Grafica.inicia/1` para iniciar el nodo.

2.3 Consenso

Para el consenso, el equipo decidió utilizar el algoritmo ya creado, **pBFT**; para la información de esta sección se utilizaron las referencias: [**geeksforgeeks_pbft**] [**castro_liskov_1999**].

La primera es más accesible y proporciona las bases del algoritmo, mientras que la segunda es más técnica, siendo el paper original. Este segundo recurso es útil para profundizar en el tema.

Antes de explicar el consenso, los problemas que el algoritmo busca resolver son:

- **Nodos Bizantinos:** Habrá nodos maliciosos que envíen mensajes incorrectos a la red. Nuestro sistema debe tolerarlos y alcanzar un consenso a pesar de su presencia.
- **Errores en los mensajes:** En una red real, los mensajes pueden perderse, retrasarse o llegar desordenados; asumimos que siempre llegan eventualmente. El sistema debe tolerar estos problemas y alcanzar el consenso.
- **Consistencia:** Todos los nodos honestos deben consensuar el orden de los bloques y validar su legitimidad.
- **Validez:** El consenso alcanzado no debe incluir bloques corruptos generados por nodos maliciosos.

El Algoritmo pBFT

El algoritmo **pBFT** sigue tres etapas principales para llegar al consenso:

1. **Preparación Inicial (Pre-Prepare):** Un nodo propone un mensaje para ser aceptado por la red. Este mensaje, llamado *preprepare*, contiene un identificador único (hash o timestamp) y el bloque propuesto.
2. **Preparación (Prepare):** Los nodos que reciben un mensaje *preprepare* validan el bloque. Si es válido, envían un mensaje *prepare* a sus vecinos. Cada nodo acumula mensajes *prepare*. Si recibe al menos $2f + 1$ mensajes válidos (incluido el suyo), pasa a la siguiente etapa.
3. **Compromiso (Commit):** En esta fase, el nodo envía un mensaje *commit* a todos sus vecinos. Si un nodo acumula $2f + 1$ mensajes *commit* válidos, acepta el bloque y lo agrega a su `blockchain` local.

Aunque el algoritmo tiene una complejidad de comunicación $O(n^2)$, consideramos que esto es manejable para redes pequeñas o medianas.

Chapter 3

Conclusiones

Durante la implementación de nuestra variante del algoritmo `Practical Byzantine Fault Tolerance (PBFT)`, llegamos a las siguientes conclusiones:

- **Eliminación del líder:**
Permitimos que cualquier nodo propusiera bloques, eliminando la dependencia del líder. Esto simplificó la lógica aunque incrementó el volumen de mensajes.

- **Topología de red:**
Usar el modelo de `Watts y Strogatz` permitió una rápida propagación de mensajes gracias a un alto coeficiente de agrupamiento.

- **Tolerancia a nodos bizantinos:**
Suponiendo que hay f nodos bizantinos, el algoritmo funciona correctamente en configuraciones de $3f + 1$, logrando consenso a pesar de las fallas.

En conclusión, implementamos una solución descentralizada basada en `PBFT` adaptada a redes distribuidas. Es posible escalar el proyecto y asegurarse que funcione correctamente en más escenarios, como varios procesos intentando mandar mensaje al mismo tiempo.