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

Índice general

Capítulo 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.

Capítulo 2

Solución propuesta

2.1. Flujo general de la solución

1. Inicialización del Sistema:

- **Entrada:** El programa recibe dos parámetros principales:
  - n: Número de nodos en la red.
  - f: Número de nodos bizantinos.
- **Configuración de la Red:**
  - *Modelo Watts y Strogatz:* Se configura la topología de la red con base en este modelo, asegurando que el coeficiente de agrupamiento sea mayor a 0.4 antes de comenzar el proceso. Este modelo garantiza que los nodos estén interconectados de manera eficiente para propagar la información.
  - **Creación de Nodos:** Se crean n nodos, de los cuales f son bizantinos, generando un total de $n - f$ nodos honestos, asegurando la desigualdad $n > 3f$ para mantener la integridad de la red.

2. Creación de Procesos y Asignación de Roles:

- **Spawning de Procesos:** Los nodos se representan como procesos concurrentes en Elixir, cada uno responsable de gestionar la lógica de su respectivo nodo en la blockchain.

  Vamos a crear nodos con diferentes roles, como se muestra a continuación:
- **Asignación de Roles:**
  - **Réplica:** Nodo que participa en el consenso de la red. Son todos los nodos excepto el líder.
  - **Líder (o primary):** Nodo que coordina el consenso de la red. Este rol se deberia asignar dinámicamente en cada ronda usando un criterio determinista. Pero para simplificar, inicialmente se elige un nodo honesto arbitrariamente.
  - **Bizantino:** Nodo que puede enviar mensajes maliciosos a la red. Este nodo también actúa como réplica, pero su comportamiento es defectuoso por diseño.

  Los estados de los nodos se gestionan de manera local en cada proceso. Cada nodo tendrá un estado inicial que incluye:

- **ID único:** Identificador del nodo el que le da elixir vamos a usar para no confundirmelo con el PID.
- **Rol:** Líder, réplica o bizantino.
- **Blockchain local:** Registro de bloques confirmados.
- **Mensajes recibidos:** Historial temporal de mensajes relacionados con la ronda actual.
- **Lider:** El ID del lider actual.

3. Ejecución del Consenso:

- Los nodos y el líder colaboran para ejecutar el consenso pBFT según lo descrito en la **Sección 2.2: Consenso**.
- El proceso incluye las fases de *pre-prepare*, *prepare* y *commit*, asegurando que todos los nodos honestos lleguen a un acuerdo sobre los bloques.
- Los nodos bizantinos intentarán interrumpir el consenso, enviando mensajes inconsistentes o maliciosos.

Resumen Visual del Flujo:

1. **Inicialización:** Parámetros (n, f) $\rightarrow$ Topología de Red (Modelo Watts y Strogatz) $\rightarrow$ Creación de Nodos (honestos y bizantinos).
2. **Asignación de Roles:** Líder creado, resto de nodos como réplicas (incluyendo bizantinos).
3. **Consenso:** Se ejecuta pBFT (**ver Sección 2.2**) para alcanzar consenso sobre nuevos bloques.
4. **Validación:** Cada nodo confirma el bloque acordado y lo agrega a su blockchain local.

2.2. Consenso : pBFT

Para el consenso, el equipo decidio utilizar el algoritmo ya creado, **pBFT**; para la información de esta seccion se utilizaron las referencias: [2] [1].

La primera es mas facil de digerir y tiene las bases del algoritmo, mientras que la segunda es mas fuerte, siendo el paper original del algoritmo, igualmente este segundo es mas por si se quiere profundizar en el tema.

Ahora si empecemos, vamos a caracterizar el problema primero para ver que es lo que se quiere resolver.

- **Nodos Bizantinos:** En nuestro sistema, como ya dijimos habra nodos que se comporten de manera maliciosa, enviando mensajes incorrectos a la red. Nuestro sistema debe ser capaz de tolerar estos nodos y llegar a un consenso a pesar de.

- **Errores en los mensajes:** Como es una red real, los mensajes pueden perderse, retrasarse o llegar en un orden incorrecto; se asume que siempre van a llegar en este modelo. Nuestro sistema debe ser capaz de tolerar estos errores y llegar a un consenso a pesar de.
- **Consistencia:** Todos los nodos que sean honestos deben llegar a un consenso sobre el orden de los bloques, y si un bloque es valido o no.
- **Validez:** El resultado consensuado no puede ser un bloque basura generado por un nodo malicioso.

Ahora, el algirtmo pBFT sigue una serie de 3 etapas para llegar a un consenso, estas son:

1. **Preparacion Inicial (Pre-Prepare):** El nodo que elegimos como lider, propone un mensaje para ser aceptado por la red, le llaman *request*. En esta etapa el lider envia el mensaje *pre-prepare* a todos los nodos, este mensaje contiene el numero de vista actual y el mensaje en si. Se utiliza generalmente un numero secuencial pero nosotros podemos usar el timestamp que llevan los bloques para esto. Junto con el hash son ID del mensaje que se envia.
2. **Preparacion (Prepare):** Una vez que un nodo recibe el mensaje *pre-prepare*, este nodo lo valida checando con la funcion del bloque, si es valido, con su ultimo bloque y el bloque si es valido también y checa tambien que sea del lider actual. Si todo esto es correcto, el nodo envia un mensaje *prepare* a todos los nodos, este mensaje contiene el hash del mensaje que se esta aceptando y el ID del nodo que lo envia. Los nodos van acumulando estos mensajes *prepare* de otros. Si un nodo recibe $2f + 1$ mensajes *prepare* validos (incluyendo el suyo), entonces este nodo puede pasar a la siguiente etapa. (f es la cantidad de nodos bizantinos, podemos ponerle que asuma el peor caso y que sea $n/3$).
3. **Compromiso (Commit):** Cuando llega a esta etapa el nodo envia un mensaje *commit* a todos los nodos, este mensaje contiene el hash del mensaje que se esta aceptando y el ID del nodo que lo envia. Los nodos van acumulando estos mensajes *commit* de otros. Si un nodo recibe $2f + 1$ mensajes *commit* validos (incluyendo el suyo), entonces este nodo puede aceptar el mensaje y agregarlo a su blockchain local. Ademas, el nodo envia un mensaje *reply* al nodo lider para que este sepa que el mensaje fue aceptado por la red.

Consideramos que este algoritmo es bueno a pesar de su complejidad en comunicacion $O(n^2)$ porque no vamos a crecer tanto la red, y ademas, es un algoritmo que es muy seguro y tolerante a fallos que no suena tan dificil de implementar pero si es posible que tengamos que hacer algunos ajustes para implementar el algoritmo en Elixir. *(Skill issue)*

2.3. Módulos y sus funciones

Capítulo 3

Resultados

3.1. Implementación

3.2. Pruebas

3.3. Notas de los resultados

Capítulo 4

Conclusiones

Bibliografía

[1] Miguel Castro y Barbara Liskov. «Practical Byzantine Fault Tolerance». En: *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*. New Orleans, LA, USA: USENIX Association, 1999. URL: https://pmg.csail.mit.edu/papers/osdi99.pdf.

[2] GeeksforGeeks. *Practical Byzantine Fault Tolerance (PBFT)*. https://www.geeksforgeeks.org/practical-byzantine-fault-tolerancepbft/. Último acceso: 22 de noviembre de 2024.