Universidad Nacional Autónoma de México

Facultad de Ciencias

Computación Distribuida

Proyecto

Salvador López Mendoza
Santiago Arroyo Lozano

Autores:

Hermes Alberto Delgado Día
319258613
José Eduardo Cruz Campos
319312087

22 de noviembre del 2024

Índice

1. Módulos

1.1. Crypto

Módulo proporcionado en clase.

- Maneja las operaciones de hashing para los bloques.
- Implementa funciones básicas para calcular y verificar hashes.

1.2. Network

- Implementa el modelo de red `Watts-Strogatz` que es usado para crear una red de nodos con propiedades de "mundo pequeño".
- El modelo funciona así:
  1. Crea un anillo inicial donde cada nodo se conecta con sus **k** vecinos más cercanos.
  2. Para cada conexión, con probabilidad β, reconecta ese enlace a un nodo aleatorio.
  3. Esto crea una red que tiene tanto agrupamiento local como "atajos" globales.

1.3. Block

- Define la estructura básica de un bloque:
  - `data`: contenido del bloque.
  - `timestamp`: marca temporal.
  - `prev_hash`: hash del bloque anterior.
  - `hash`: hash del bloque inicial.
- **new/2**: Crea un nuevo bloque con datos y hash previo.
- **valid?/1**: Valida un bloque individual.
  - Verifica que el hash sea correcto.
  - Rechaza datos maliciosos.
  - Comprueba que el timestamp no esté en el futuro.
- **valid?/2**: Verifica la validez de dos bloques consecutivos.
- **to_string/1**: Convierte un bloque a una representación de cadena legible.

1.4. Blockchain

Gestiona la cadena de bloques completa:

- **insert/2**: Añade un nuevo bloque a la cadena.
- **validate_block_for_chain/2**: Valida si un bloque puede ser añadido a la cadena.
- **valid?/1**: Verifica la integridad de toda la cadena de bloques.
- **to_string/1**: Convierte toda la cadena a una representación de cadena.

1.5. BlockchainNode

Simula un nodo en la red blockchain con comportamiento bizantino:

- **start/3**: Inicia un nodo con un ID, vecinos y probabilidad de comportamiento bizantino.
- Implementa diferentes estrategias de nodo bizantino:
  - Modificación sutil de bloques.
  - Generación de bloques falsos.
  - Ignorar mensajes.
- Maneja la validación y propagación de bloques.
- Soporta tanto comportamiento normal como bizantino.

1.6. Main

- Módulo de control principal.
- Configura y ejecuta la red.
- Permite proponer transacciones y consultar el estado.

2. Modelo de Watts-Strogatz

2.1. Creación del Anillo Inicial

```elixir
defp ring_neighbors(node, n, k) do
  for i <- 1..div(k, 2),
      neighbor <- [rem(node + i + n, n), rem(node - i + n, n)],
      do: neighbor
end
```

- Comienza creando un anillo donde cada nodo está conectado a sus **k/2** vecinos más cercanos en ambas direcciones.
- Usa aritmética modular (**rem**) para manejar la circularidad del anillo.
- Si **k** es 4, cada nodo se conecta a 2 veciones a la derecha y 2 a la izquierda.

2.2. Reconexión Probabilística

```elixir
defp rewire_neighbors(node, neighbors, n, beta, network) do
  Enum.map(neighbors, fn neighbor ->
    if :rand.uniform() < beta do
      new_neighbor = get_random_node(n, [node | neighbors])
      new_neighbor
    else
      neighbor
    end
  end)
end
```

- Con probabilidad **beta**, cada conexión se reemplaza por una conexión a un nodo aleatorio.
- Evita conexiones a nodos ya conectados o al propio nodo.
- Esto introduce aleatoriedad manteniendo algunas propiedades de la red original.

2.3. Características del Modelo

- Preserva el agrupamiento de la red original.
- Introduce atajos aleatorios.
- Simula redes del mundo real con clustering local y conexiones de largo alcance.

3. Algoritmo de Consenso

El algoritmo de consenso busca:

- Validar bloques.
- Prevenir inserciones maliciosas.
- Mantener la integridad de la cadena.
- Tolerar la presencia de nodos bizantinos(no confiables).

3.1. Componentes Principales del Consenso

3.1.1. Validación de Bloques Individual

```elixir
def valid?(block) do
  cond do
    # Verificación de hash
    block.hash != Crypto.hash(block) ->
      {:error, "Hash inválido"}

    # Detección de datos maliciosos
    is_binary(block.data) && String.starts_with?(block.data, "
      INVALID_DATA_") ->
      {:error, "Datos maliciosos detectados"}

    # Validación de timestamp
    DateTime.compare(block.timestamp, DateTime.utc_now()) == :gt ->
      {:error, "Timestamp inválido"}

    true ->
      {:ok, block}
  end
end
```

Criterios de Validación:

- Integridad del hash.
- Prevención de datos maliciosos.
- Coherencia temporal.

3.1.2. Validación de Cadena Consecutiva

```elixir
def valid?(chain) do
  chain
  |> Enum.chunk_every(2, 1, :discard)
  |> Enum.reduce_while({:ok, []}, fn [block1, block2], {:ok, _} ->
    case Block.valid?(block1, block2) do
      {:ok, _} -> {:cont, {:ok, [block1, block2]}}
      {:error, reason} -> {:halt, {:error, reason}}
    end
  end)
end
```

Objetivo:

- Verificar la continuidad entre bloques.
- Asegurar que cada bloque apunte correctamente al anterior.

3.2. Mecanismo de Consenso Distribuido

3.2.1. Probabilidad Bizantina

```elixir
defp is_byzantine?(byzantine_probability) do
  :rand.uniform() < byzantine_probability
end
```

- Introduce aleatoriedad en el comportamiento de nodos.
- Simula escenarios de red no confiable.

3.2.2. Estrategias de Nodos

Nodos Normales:

- Validan bloques estrictamente.
- Propagan bloques válidos.
- Mantienen la integridad de la cadena.

Nodos Bizantinos:
Posibles acciones:

- Modificar bloques sutilmente.
- Crear bloques falsos.
- Ignorar mensajes.

- Rechazar bloques arbitrariamente.

Ciclos de Validación:

```elixir
{:validate_block, block} ->
  cond do
    # Bloque ya procesado
    MapSet.member?(state.processed_blocks, block.hash) ->
      loop(state)

    # Comportamiento bizantino
    is_byzantine?(state.byzantine_probability) ->
      byzantine_validation_action = [:accept, :reject, :ignore]
      |> Enum.random()

      case byzantine_validation_action do
        :accept ->
          # Aceptar sin validación completa
          new_chain = state.blockchain ++ [block]
          broadcast_to_neighbors(state.neighbors, {:validate_block,
              block})

        :reject ->
          # Rechazar arbitrariamente

        :ignore ->
          # Ignorar completamente
      end

    # Validación normal
    true ->
      case Blockchain.validate_block_for_chain(block, state.
          blockchain) do
        {:ok, _} ->
          # Agregar bloque válido
          new_chain = state.blockchain ++ [block]
          broadcast_to_neighbors(state.neighbors, {:validate_block,
              block})

        {:error, reason} ->
          # Rechazar bloque inválido
      end
  end
```

- Cada nodo valida y propaga bloques a sus vecinos.
- Construye consenso mediante validación distribuida.

3.3. Mecanismos de Propagación

```elixir
defp broadcast_to_neighbors(neighbors, message) do
  Enum.each(neighbors, fn neighbor ->
    send(:"node_#{neighbor}", message)
  end)
end
```

Funcionalidad:

- Enviar bloques a todos los nodos vecinos.
- Propagación rápida de información.
- Basado en la topología de red Watts-Strogatz.

3.4. Proceso Completo del Consenso

1. Nodo crea bloque.
2. Valida bloque localmente.
3. Propaga a vecinos.
4. Cada vecino:
   - Valida individualmente.
   - Decide aceptar/rechazar.
   - Propaga si es válido.
5. Red tiende a convergir.

3.5. Tolerancia Bizantina

- Probabilidad configurable de nodos maliciosos.
- Estrategias aleatorias de manipulación.
- Simulación de escenarios de red real.

4. Comando de ejecución

Para ejecutar el código dentro de la carpeta src:

```
?- elixir crypto.ex
```

5. Ejemplo de ejecución

Se crea una red de 10 nodos con 20 % de probabilidad de comportamiento bizantino.

```
network = Main.run(10, 0.2)
```

Se muestra el estado de la red.

```
Main.print_network_status(network)
```

Se realizan tres transferencias.

```
Main.propuse_transaction(0,"Transferencia: Alice ->Bob: $100" )
Main.propuse_transaction(1,'Transferencia: Bob ->Charlie: $20" )
Main.propuse_transaction(2,"Transferencia: Charlie ->Alice: $30" )
```

Se obtiene la cadena de bloques del nodo 0 y se imprime en consola.

```
Main.get_blockchain(0)
```