Practica Sensor de Temperatura

Para este sistema, utilizaremos un **Patrón de Arquitectura por Capas** y el concepto de **Inmutabilidad** para el bloque.

Arquitectura del Sistema

1. **Modelo (Block):** El "átomo" de nuestra cadena.
2. **Servicio de Blockchain:** Encargado de validar y añadir bloques.
3. **Servidor (Controlador):** Escucha al cliente, verifica la temperatura y decide el "apagado".
4. **Cliente (Sensor):** Captura datos, hashea y envía.

1. El Eslabón: Clase Block

Un bloque moderno debe ser inmutable. Usaremos SHA-256 para el hash. La fórmula matemática del hash de cada bloque es:

Hash = SHA256(index + timestamp + data + previousHash)

2. El Servidor: "The Guard" (`Server.java`)

El servidor no solo recibe datos; actúa como un **Monitor de Seguridad**. Usaremos un umbral (por ejemplo, 50°C).

Implementa un método `isChainValid()` en el servidor que recorra la lista y verifique que `currentBlock.previousHash` sea igual a `previousBlock.hash`.

2.1. Implementación de la Validación de Integridad

Añadiremos este método dentro de la clase que gestione la lista (en nuestro caso, dentro del `Servidor` o una clase `BlockchainManager`).

2.1.1. El Método `isChainValid()`

Este método recorre la cadena bloque a bloque realizando dos comprobaciones críticas:

1. **Integridad del Bloque:** ¿El hash guardado coincide con el contenido actual?
2. **Continuidad de la Cadena:** ¿El bloque apunta correctamente al hash del anterior?

2.1.2. Integración en el Flujo del Servidor

Lo ideal es que el servidor ejecute esta validación **cada vez que se intenta añadir un nuevo bloque**. Si la cadena no es válida antes de empezar, el servidor debería rechazar nuevas conexiones.

3. El Cliente: "The Sensor" (`SensorClient.java`)

El cliente es el responsable de la "prueba de trabajo" (en este caso, generar el registro y enviarlo).

Patrones y Mejores Prácticas Aplicadas

1. **Separación de Preocupaciones (SoC):** La lógica de cifrado está en el modelo (`Block`), la lógica de negocio en el `Server` y la captura de datos en el `Client`.
2. **Protocolo de Mensajería:** Hemos definido un formato simple `COMANDO:VALOR`. En aplicaciones reales, usarías **JSON**.
3. **Fail-Fast (Apagado Crítico):** El servidor rompe el bucle de escucha (`break`) inmediatamente al detectar el peligro, simulando un apagado físico.
4. **Inmutabilidad del Ledger:** Al usar la lista `blockchain` vinculada por hashes, si alguien cambiara un dato antiguo en la memoria, el hash del bloque siguiente ya no coincidiría.

¿Por qué esto es un patrón de programación correcto?

Al aplicar este método, estás implementando el principio de **"Confianza Cero" (Zero Trust)**:

- **Inmutabilidad Verificable:** No asumes que los datos en memoria están bien solo porque tú los escribiste. Los verificas matemáticamente.
- **Efecto Dominó:** Debido a que el hash del Bloque 3 depende del hash del Bloque 2, cualquier cambio mínimo en el Bloque 1 invalidará **toda** la cadena posterior.
- **Auditoría en Tiempo Real:** En programación de servicios y procesos, es vital que el servidor se autoevalúe constantemente para evitar propagar errores o datos corruptos por la red.

Resumen de la práctica completa

1. **Cliente (Sensor):** Envía `TEMP:55`.
2. **Servidor:** * Verifica `isChainValid()`.
    - Comprueba si `55 > Limite`.
    - Si es así, emite orden de `APAGADO`.
    - Si no, genera el `Block`, calcula el SHA-256 y lo guarda.

Debemos seguir un esquema de **almacenamiento híbrido**.

En sistemas reales de Blockchain e IoT, no se guarda toda la información pesada "on-chain" (dentro de la cadena) porque es ineficiente y caro. En su lugar, se usa **Off-chain Storage** para el detalle y la **Blockchain** para la integridad.

Aquí tienes la estructura recomendada para ambas bases de datos:

4. Almacenamiento de datos

4.1. Base de Datos Remota (Relational/SQL)

Propósito: Almacenar el historial detallado, realizar estadísticas y consultas rápidas de temperatura. Podrías usar **MySQL** o **PostgreSQL**.

Tabla: historico_temperaturas

Esta tabla guarda el "qué" y el "cuándo".

Campo	Tipo	Descripción
id	INT (PK)	Identificador único autoincremental.
sensor_id	VARCHAR	Identificador del cliente/terminal que envía el dato.
temperatura	DECIMAL	El valor capturado (ej: 42.5).
fecha_hora	DATETIME	Timestamp exacto de la captura.
blockchain_ref	VARCHAR	**El Hash del bloque** generado. Sirve de enlace entre ambas BD.

4.2. Base de Datos Blockchain (Ledger)

Propósito: Garantizar que los datos no han sido borrados ni modificados. En Java, esto suele representarse como un archivo plano (JSON/XML) o una BD de clave-valor como **LevelDB**.

Estructura del Bloque (Token)

Cada registro en esta base de datos es un bloque inmutable.

Campo	Descripción
index	La posición del bloque en la cadena.
timestamp	Fecha de creación del bloque.
previous_hash	El hash del bloque anterior (garantiza la cadena).
data_hash	**El Hash del registro de la BD remota**. Si alguien cambia la temperatura en la SQL, este hash ya no coincidirá.
hash	El identificador único del bloque actual (SHA-256).

4.3 El Flujo Correcto

Para que el sistema sea robusto, el proceso debe seguir este orden exacto:

1. **Captura:** El Cliente lee la temperatura.
2. **Inserción SQL:** El Servidor guarda los datos en la BD `historico_temperaturas`.
3. **Hasheo:** El Servidor genera un hash de esa fila (puedes hashear el `id + temperatura + fecha_hora`).
4. **Tokenización:** Ese hash resultante se mete como `data_hash` en un nuevo bloque de la **Blockchain**.
5. **Cierre del Círculo:** El `hash` del bloque generado se actualiza en la columna `blockchain_ref` de la BD SQL.

¿Por qué así?

Si un hacker entra en tu MySQL y cambia una temperatura de 80°C a 20°C para que no parezca que hubo un fallo, tú puedes correr un script de auditoría que:

1. Recalcule el hash del dato en MySQL.
2. Lo compare con el `data_hash` guardado en la Blockchain.
3. Como el hash no coincide, detectas el fraude inmediatamente aunque los datos "parezcan" correctos.

4.4 Estructura SQL (Base de Datos Remota)

Este script crea la tabla donde guardaremos el histórico. El campo `blockchain_hash` es vital, ya que es el "puntero" que nos dice en qué bloque de la cadena se certificó ese dato.

SQL

```
CREATE DATABASE sistema_monitoreo;
USE sistema_monitoreo;

CREATE TABLE lecturas_temperatura (
    id INT AUTO_INCREMENT PRIMARY KEY,
    sensor_id VARCHAR(50) NOT NULL,
    valor_temp DECIMAL(5,2) NOT NULL,
    fecha_registro TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    -- Este campo vincula la fila con un bloque específico
    blockchain_hash VARCHAR(64) DEFAULT NULL
);
```

4.5 Generación del "Data Hash" en Java

Para que la Blockchain proteja el registro SQL, no guardamos la temperatura directamente, sino un **Hash de Integridad** del registro. Si alguien cambia el `valor_temp` en el SQL, el hash dejará de coincidir.

Añadiremos este método en tu servidor para procesar el dato antes de meterlo en el bloque.

4.6. El Flujo de Trabajo "Correcto" (Pattern)

Para implementar esto en tu `ManejadorCliente` (el hilo del servidor), el orden de ejecución debe ser este:

1. **Recibir:** El socket recibe la temperatura del cliente.
2. **Persistir SQL:** Insertar en la tabla `lecturas_temperatura` (el campo `blockchain_hash` se queda vacío de momento).
3. **Certificar**:
    - Recuperas el ID y la fecha que acaba de asignar la base de datos.
    - Generas el `DataHash` con esos valores.
    - Creas un nuevo `Block` en la Blockchain cuyo contenido (`data`) sea ese `DataHash`.
4. **Actualizar SQL:** Haces un `UPDATE` en la fila de la base de datos para guardar el `hash` del bloque recién creado en la columna `blockchain_hash`.

4.7 ¿Cómo se audita esto?

Si dentro de un mes quieres saber si los datos son reales, el proceso de auditoría sería:

1. Tomas una fila de la SQL.
2. Recalculas su hash usando `generarDataHash`.
3. Buscas en la Blockchain el bloque que tenga el hash que indica la columna `blockchain_hash`.
4. Compruebas si el `data` de ese bloque es **idéntico** al hash que acabas de recalcular.

Si coinciden: El dato es íntegro. **Si no coinciden:** Alguien ha manipulado la base de datos SQL y la "prueba matemática" de la Blockchain lo ha detectado.

5. Conexion con la base de datos:

5.1. Preparación: El Driver

Asegúrate de tener el conector de MySQL (el archivo `.jar`) en el *classpath* de tu proyecto.

5.2. Clase de Conexión: `DatabaseService.java`

Es mejor separar la lógica de base de datos en una clase dedicada (Patrón DAO/Service).

5.3. Integración en el Servidor.

Ahora, en el hilo que maneja al cliente, debes llamar a estos métodos en el orden correcto.

Java

```
// Dentro del método que procesa la temperatura en el Servidor:

// 1. Guardar en la base de datos SQL remota (IP: 192.168.20.118)
int idGenerado = DatabaseService.guardarLectura(sensorId, temp);

if (idGenerado != -1) {
    // 2. Crear el bloque en la Blockchain
    // Usamos el hash previo de nuestra lista en memoria
    String prevHash = blockchain.get(blockchain.size() - 1).hash;

    // El 'data' del bloque será el hash de integridad de la BD SQL
    String dataHash = generarDataHash(sensorId, temp,
String.valueOf(idGenerado));
    Block nuevoBloque = new Block(dataHash, prevHash);
    blockchain.add(nuevoBloque);

    // 3. Volver a la SQL para guardar la referencia del bloque (el sello final)
    DatabaseService.vincularConBlockchain(idGenerado, nuevoBloque.hash);

    System.out.println("Sincronización completa: SQL (ID " + idGenerado + ") <->
Blockchain (Hash " + nuevoBloque.hash.substring(0,8) + "...)");
}
```

1. **Conectividad Remota:** Java se conecta a la IP `192.168.20.118` usando el puerto estándar de MySQL (3306).
2. **Persistencia Dual:** El dato vive en una tabla SQL para ser consultado fácilmente, pero su "huella digital" vive en la Blockchain para ser inmutable.
3. **Integridad:** Gracias a que recuperamos el `idGenerado`, nos aseguramos de que el bloque de la Blockchain certifique exactamente la fila que acabamos de crear en la base de datos remota.

Un detalle importante: Asegúrate de que en el servidor con la IP `192.168.20.118`, el usuario de la base de datos tenga permisos para recibir conexiones externas (no solo desde `localhost`), de lo contrario verás un error de "Connection Refused".

El `sensorId` es el **DNI de tu dispositivo**. En un sistema real de IoT, es fundamental para saber qué sensor físico está enviando la información (por ejemplo: "Sensor_Planta_1", "Sensor_Pasillo_Norte", etc.).

Tienes dos formas principales de generarlo, pero la más "correcta" en programación de servicios es que **el cliente se identifique al conectar**.

5.4 Generarlo en el Cliente

El propio código del cliente tiene un identificador único (puede ser un nombre fijo, una dirección MAC o un UUID).

En el Cliente (`SensorClient.java`):

Java

```
// Podemos usar un ID fijo para este cliente específico
private static final String SENSOR_ID = "SENSOR_VALLE_01";

// O generar uno aleatorio para cada sesión si fuera necesario
// private static final String SENSOR_ID = UUID.randomUUID().toString();

// Al enviar el mensaje, cambiamos el protocolo para incluirlo:
out.println(SENSOR_ID + ":" + temp);
```

5.5 Cómo integrarlo en tu flujo actual

Para que el sistema sea profesional, así es como quedaría el reparto de responsabilidades:

1. **El Cliente** envía: `"SENSOR_A1;TEMP:38.5"`
2. **El Servidor** recibe esa cadena y hace un `split(";")`.
3. **El Servidor** extrae:
    - `partes[0]` -> `SENSOR_A1` (este es tu **sensorId**).
    - `partes[1]` -> `TEMP:38.5` (de aquí sacas el **valor numérico**).

Modificación rápida en el Servidor para capturarlo:

Java

```
// Dentro del run() del hilo del servidor
String mensajeRecibido = in.readLine();
String[] partes = mensajeRecibido.split(";");

String sensorId = partes[0]; // "SENSOR_A1"
double temp = Double.parseDouble(partes[1].split(":")[1]); // 38.5

// Ahora ya tienes el sensorId listo para pasarlo a:
// 1. DatabaseService.guardarLectura(sensorId, temp);
// 2. generarDataHash(sensorId, temp, timestamp);
```

Cambia tu variable `URL` por esta:

Java

```
private static final String URL = "jdbc:mysql://" + IP + ":3306/" + DB +
    "?useSSL=false" +
    "&serverTimezone=UTC" +
    "&allowPublicKeyRetrieval=true"; // <--- ESTA ES LA CLAVE
```

Autorizar al usuario en MySQL (Permiso %)

Por defecto, los usuarios de MySQL suelen estar restringidos a `localhost`. Debes entrar en la consola de MySQL del servidor remoto y ejecutar:

SQL

```
-- 1. Crear el usuario especificando que puede conectar desde cualquier IP ('%')
CREATE USER 'usuario_bd'@'%' IDENTIFIED BY 'tu_password';

-- 2. Asignar los privilegios sobre tu base de datos específica
GRANT ALL PRIVILEGES ON sistema_monitoreo.* TO 'usuario_bd'@'%';

-- 3. Refrescar la memoria de privilegios
FLUSH PRIVILEGES;
```

Cambiar el "Bind Address" (Configuración del Servidor)

Incluso con el usuario autorizado, MySQL a menudo solo escucha peticiones que vienen de sí mismo.

1. En el servidor remoto, busca el archivo de configuración (suele ser `/etc/mysql/mysql.conf.d/mysqld.cnf` en Linux o `my.ini` en Windows).
2. Busca la línea que dice: `bind-address = 127.0.0.1`.
3. Cámbiala por: `bind-address = 0.0.0.0` (esto le dice a MySQL que escuche en todas las interfaces de red).
4. Reinicia el servicio MySQL.

Notas importantes

- **Creacion del usuario:** Si el usuario ya existe: Si ya habías intentado crearlo antes, el comando `CREATE USER` te dará error. En ese caso, usa:

  ```
  ALTER USER 'usuario_bd'@'%' IDENTIFIED BY 'tu_password';
  ```

  seguido del `GRANT`.

- **Verificación:** Puedes comprobar si el usuario se creó correctamente para conexiones externas con esta consulta:

  ```
  SELECT user, host FROM mysql.user WHERE user = 'usuario_bd';
  ```

  Deberías ver una fila donde el `host` sea `%`.

Abrir el Firewall

Asegúrate de que el firewall del equipo remoto permita tráfico entrante por el puerto **3306**.

- **Si el remoto es Windows:** Ve a "Panel de Control" > "Firewall" > "Reglas de entrada" > "Nueva regla" > "Puerto" > **TCP 3306** > "Permitir conexión".
- **Si el remoto es Linux (Ubuntu/Debian):** Ejecuta `sudo ufw allow 3306/tcp`