Red Social Distribuida - MongoDB

Jesus Eduardo Cornejo Clavel
ALUMNO

Eduardo Cornejo-Velazquez
PROFESOR

23 de abril de 2025

Índice

1. Arquitectura del Sistema

1.1. Configuración de MongoDB

1.1.1. Configuración del Replica Set

La aplicación implementa una arquitectura de alta disponibilidad basada en MongoDB con una configuración de Replica Set. Esta configuración consta de tres nodos:

- **Nodo Primario (Primary)**: Gestiona todas las operaciones de escritura y coordina la sincronización con los nodos secundarios.
- **Nodos Secundarios (Secondary)**: Dos nodos que mantienen copias sincronizadas de los datos y pueden asumir el rol primario en caso de fallos.

Esta arquitectura proporciona:

- **Alta disponibilidad**: Si el nodo primario falla, uno de los secundarios puede ser promovido automáticamente.
- **Redundancia de datos**: Los datos se replican en múltiples nodos, evitando pérdidas en caso de fallos.
- **Balanceo de lecturas**: Las operaciones de lectura pueden distribuirse entre los nodos secundarios.

La configuración se implementa mediante Docker Compose:

```
services:
  # Nodo MongoDB primario
  central-mongodb:
    image: mongo:latest
    container_name: central-mongodb
    command: mongod --replSet rs0 --keyFile /etc/mongo-keyfile --
   bind_ip_all --auth --port 27017
    ports:
      - "27017:27017"
    volumes:
      - ./mongo-keyfile:/etc/mongo-keyfile:ro
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
    networks:
      - ddbp_mongo-network

  # Primer nodo MongoDB secundario
  secondary-mongodb-1:
    image: mongo:latest
    container_name: secondary-mongodb-1
    command: mongod --replSet rs0 --keyFile /etc/mongo-keyfile --
   bind_ip_all --auth --port 27017
    ports:
      - "27018:27017"
    volumes:
      - ./mongo-keyfile:/etc/mongo-keyfile:ro
```

```
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
    networks:
      - ddbp_mongo-network
    depends_on:
      - central-mongodb

  # Segundo nodo MongoDB secundario
  secondary-mongodb-2:
    image: mongo:latest
    container_name: secondary-mongodb-2
    command: mongod --replSet rs0 --keyFile /etc/mongo-keyfile --
   bind_ip_all --auth --port 27017
    ports:
      - "27019:27017"
    volumes:
      - ./mongo-keyfile:/etc/mongo-keyfile:ro
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
    networks:
      - ddbp_mongo-network
    depends_on:
      - central-mongodb
```

El proceso de inicialización del Replica Set se maneja mediante un contenedor adicional que ejecuta un script para configurar los nodos como un conjunto replicado:

```
  # Contenedor de configuraci n de MongoDB - inicializa el replica
   set
  mongo-setup:
    image: mongo:latest
    container_name: mongo-setup
    restart: "no"
    depends_on:
      - central-mongodb
      - secondary-mongodb-1
      - secondary-mongodb-2
    networks:
      - ddbp_mongo-network
    volumes:
      - ./setup-replica.sh:/setup-replica.sh:ro
    entrypoint: ["/bin/bash", "/setup-replica.sh"]
```

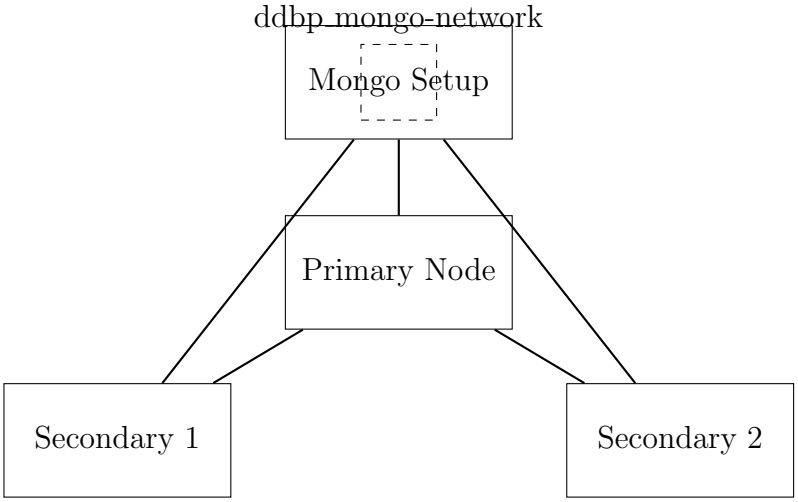

Figura 1: Arquitectura de contenedores MongoDB con Replica Set

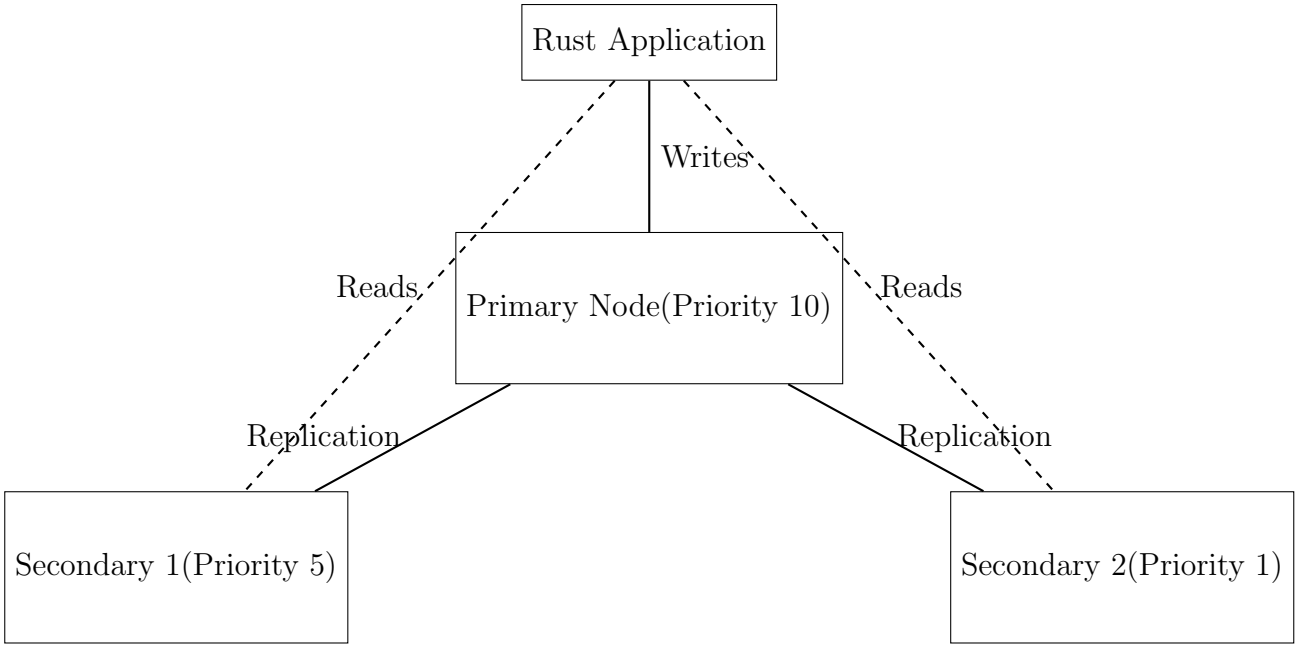

Figura 2: Arquitectura del ReplicaSet MongoDB con esquema de prioridad y flujo de datos

1.1.2. Arquitectura de ReplicaSet y Priorización

La implementación del ReplicaSet en nuestra arquitectura sigue un esquema de priorización estratégica (10-5-1) que garantiza un comportamiento predecible durante las elecciones de nodo primario y la distribución de cargas de trabajo:

- **Nodo Central (Prioridad 10)**: Configurado como el servidor preferido para el rol primario, este nodo tiene la prioridad más alta para garantizar estabilidad en condiciones normales de operación.
- **Primer Nodo Secundario (Prioridad 5)**: Mantiene una prioridad intermedia, sirviendo como primer candidato de respaldo para asumir el rol primario en caso de fallo del nodo central.

- **Segundo Nodo Secundario (Prioridad 1)**: Con la prioridad más baja, este nodo está diseñado principalmente para mantener redundancia y servir lecturas, asumiendo el rol primario solo en situaciones extremas.

La configuración del ReplicaSet se implementa mediante un script de inicialización automatizado:

```
rs.initiate({
    _id: 'rs0',
    members: [
        { _id: 0, host: 'central-mongodb:27017', priority: 10 },
        { _id: 1, host: 'secondary-mongodb-1:27017', priority: 5 },
        { _id: 2, host: 'secondary-mongodb-2:27017', priority: 1 }
    ]
})
```

Proceso de Elección El proceso de elección de nodo primario se rige por las siguientes reglas:

1. Los nodos participan en una elección cuando:
   - El nodo primario actual se vuelve inaccesible
   - Se produce un reinicio planificado del nodo primario
   - Se detecta una partición de red
2. El nodo con la prioridad más alta disponible gana la elección
3. Se requiere una mayoría de nodos (al menos 2 de 3) para completar una elección

1.1.3. Implementación de Seguridad

La seguridad del ReplicaSet se implementa mediante múltiples capas de protección:

Autenticación por Keyfile Los nodos del ReplicaSet utilizan un archivo de clave compartida para la autenticación interna:

```
command: mongod --replSet rs0 --keyFile /etc/mongo-keyfile --
   bind_ip_all --auth --port 27017
volumes:
    - ./mongo-keyfile:/etc/mongo-keyfile:ro
```

El keyfile se monta como volumen de solo lectura en cada contenedor, garantizando que solo los nodos autorizados puedan unirse al ReplicaSet.

Control de Acceso Basado en Roles (RBAC) Se implementa un esquema de RBAC que define roles específicos para diferentes tipos de operaciones:

```
db.createUser({
    user: 'admin',
    pwd: 'password',
    roles: [
        { role: 'readWrite', db: 'social_media_db' },
        { role: 'dbAdmin', db: 'social_media_db' },
        { role: 'userAdmin', db: 'social_media_db' }
    ]
});
```

Aislamiento de Red Los contenedores operan en una red dedicada (ddbp$_m$ongo−network)quepropor

- Aislamiento del tráfico de base de datos
- DNS interno para resolución de nombres entre contenedores
- Control de acceso a nivel de red

```
networks:
    ddbp_mongo-network:
        name: ddbp_mongo-network
```

1.1.4. Proceso de Inicialización Automatizada

La inicialización del ReplicaSet se automatiza mediante un contenedor dedicado (mongo-setup) que ejecuta un script de configuración:

1. **Verificación de Disponibilidad**: El script verifica que todos los nodos estén operativos antes de iniciar la configuración.
2. **Inicialización del ReplicaSet**: Se configura la topología del ReplicaSet con las prioridades correspondientes.
3. **Creación de Usuarios y Colecciones**: Se establecen los usuarios administrativos y las colecciones iniciales.
4. **Verificación de Estado**: Se confirma el correcto funcionamiento del ReplicaSet mediante pruebas de estado.

El script de configuración incluye mecanismos de reintento y validación:

```
#!/bin/bash
check_mongodb_ready() {
    local host=$1
    echo "Checking if $host is ready..."
    for i in {1..30}; do
        if mongosh --host $host --port 27017 -u admin -p password \
            --authenticationDatabase admin --eval "db.adminCommand('
   ping')" &>/dev/null; then
            echo "$host is ready!"
            return 0
        fi
        echo "Waiting for $host to be ready (attempt $i/30)..."
        sleep 2
    done
    return 1
}
```

Esta configuración automatizada garantiza una inicialización consistente y segura del clúster de MongoDB, estableciendo la base para operaciones distribuidas confiables.

1.1.5. Connection Pooling y Resiliencia

La aplicación implementa una configuración robusta para la conexión a MongoDB que garantiza alta disponibilidad, rendimiento óptimo y recuperación ante fallos:

```
// Configurar connection pooling
client_options.max_pool_size = Some(20);
client_options.min_pool_size = Some(5);
client_options.max_idle_time = Some(Duration::from_secs(60));
```

- **Max Pool Size**: Limita el número máximo de conexiones concurrentes a 20 para prevenir saturación.
- **Min Pool Size**: Mantiene al menos 5 conexiones disponibles para minimizar latencia en solicitudes.
- **Max Idle Time**: Cierra conexiones inactivas después de 60 segundos para optimizar recursos.

```
// Configurar timeouts
client_options.connect_timeout = Some(Duration::from_secs(10));
client_options.server_selection_timeout = Some(Duration::from_secs(15)
   );
```

- **Connect Timeout**: 10 segundos para establecer la conexión inicial.
- **Server Selection Timeout**: 15 segundos para seleccionar un servidor disponible.

```
// Configurar read/write concerns para mejor fiabilidad
client_options.read_concern = Some(ReadConcern::majority());
client_options.write_concern = Some(
    WriteConcern::builder()
        .w(mongodb::options::Acknowledgment::Majority)
        .build(),
);
```

- **Read Concern**: Configurado como `majority` para garantizar que las lecturas devuelvan datos confirmados por la mayoría de nodos.
- **Write Concern**: Configurado para requerir confirmación de la mayoría de nodos antes de considerar una escritura como exitosa.

```
// Establecer preferencia de lectura a SecondaryPreferred
let options = ReadPreferenceOptions::default();
client_options.selection_criteria = Some(
    ReadPreference::SecondaryPreferred {
        options: Some(options),
    }
    .into(),
);
```

- **Read Preference**: Configurado como `SecondaryPreferred` para dirigir operaciones de lectura a nodos secundarios cuando estén disponibles, descargando así al primario.

```
// Configurar comportamiento de reintentos
client_options.retry_reads = Some(true);
client_options.retry_writes = Some(true);

// Configurar heartbeat para detectar problemas r pidamente
client_options.heartbeat_freq = Some(Duration::from_secs(15));
```

- **Retry Reads/Writes**: Habilitados para reintentar automáticamente operaciones fallidas.
- **Heartbeat**: Configurado para verificar el estado de los servidores cada 15 segundos.

Esta configuración integral garantiza que la aplicación pueda manejar eficientemente situaciones como:

- Fallos temporales de red
- Caída de nodos individuales
- Elecciones de nuevo primario
- Picos de carga
- Desconexiones transitorias

1.2. Diagrama de Comunicación Rust

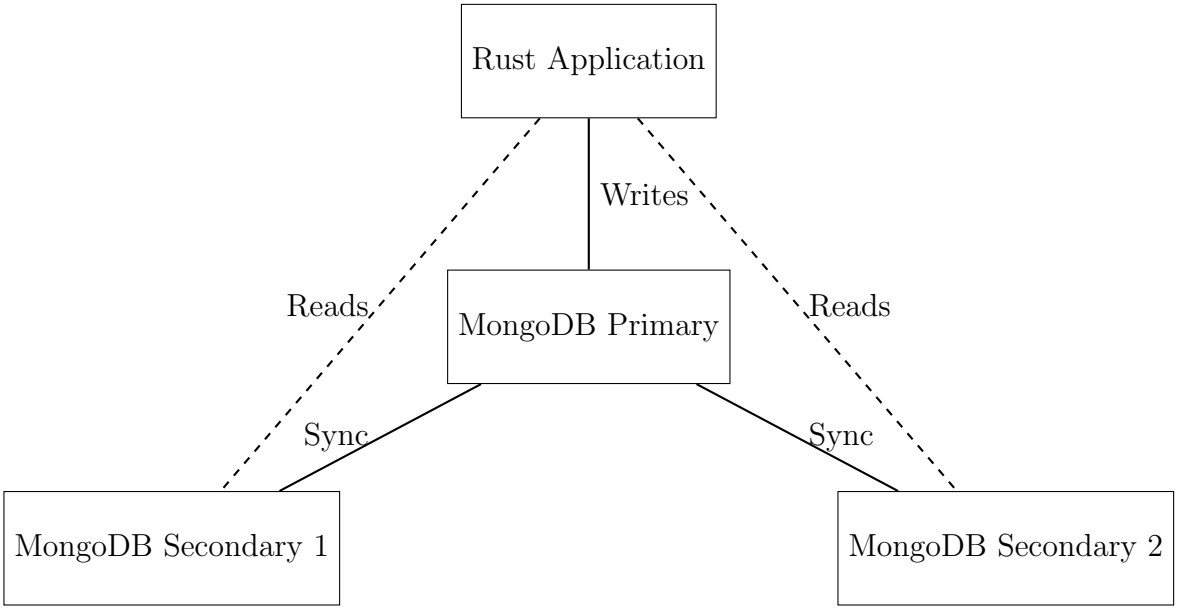

Figura 3: Diagrama de comunicación de la aplicación Rust con MongoDB ReplicaSet

Políticas de Reintento y Monitoreo

2. Descripción de Componentes

2.1. Configuración de MongoDB

2.1.1. Configuración del Replica Set

2.1.2. Connection Pooling y Resiliencia

2.2. Estructura de la Aplicación Rust

2.2.1. API Endpoints y Funcionalidades

2.2.2. Manejo de Errores y Responses

2.2.3. Modelos de Datos

3. Modelos de Datos

3.1. Diseño de Colecciones

El sistema implementa un modelo de datos distribuido optimizado para operaciones sociales y escalabilidad horizontal. Las colecciones están diseñadas para maximizar la eficiencia en un entorno distribuido mientras mantienen la consistencia de los datos.

3.1.1. Colección de Usuarios

```
{
    "_id": ObjectId(),
    "username": "string",          //  nico , indexado
    "email": "string",            //  nico , indexado
    "password_hash": "string",
    "profile": {
        "full_name": "string",
        "bio": "string",
        "avatar_url": "string",
        "created_at": ISODate(),
        "last_login": ISODate()
    },
    "followers_count": NumberInt, // Contador denormalizado
    "following_count": NumberInt, // Contador denormalizado
    "posts_count": NumberInt      // Contador denormalizado
}
```

Índices y Optimización

- Índice único en `username`: {`username: 1`}
- Índice único en `email`: {`email: 1`}
- Índice compuesto para búsquedas: {`username: 1, email: 1`}

3.1.2. Colección de Posts

```
{
    "_id": ObjectId(),
    "user_id": ObjectId(),      // Referencia al autor
    "content": "string",
    "media_urls": ["string"],   // URLs de contenido multimedia
    "created_at": ISODate(),
    "likes_count": NumberInt,   // Contador denormalizado
    "comments_count": NumberInt, // Contador denormalizado
    "tags": ["string"],
    "location": {
        "type": "Point",
        "coordinates": [NumberDouble, NumberDouble]
    }
}
```

Índices y Optimización

- Índice en `user_id`: {`user_id: 1`}
- Índice en `created_at`: {`created_at: -1`}
- Índice geoespacial: {`location: "2dsphere"`}
- Índice de texto en contenido: {`content: "text", tags: "text"`}

3.1.3. Colección de Comentarios

```
{
    "_id": ObjectId(),
    "post_id": ObjectId(),        // Referencia al post
    "user_id": ObjectId(),        // Referencia al autor
    "content": "string",
    "created_at": ISODate(),
    "likes_count": NumberInt,     // Contador denormalizado
    "parent_comment_id": ObjectId() // Para comentarios anidados
}
```

Índices y Optimización

- Índice compuesto: {`post_id: 1, created_at: -1`}
- Índice en `user_id`: {`user_id: 1`}
- Índice en `parent_comment_id`: {`parent_comment_id: 1`}

3.1.4. Colección de Likes

```
{
    "_id": ObjectId(),
    "user_id": ObjectId(),
    "target_id": ObjectId(),  // ID del post o comentario
    "target_type": "string",  // "post" o "comment"
    "created_at": ISODate()
}
```

3.1.5. Colección de Follows

```
{
    "_id": ObjectId(),
    "follower_id": ObjectId(),
    "following_id": ObjectId(),
    "created_at": ISODate()
}
```

3.2. Estrategias de Modelado de Datos

3.2.1. Denormalización Estratégica

El sistema implementa denormalización selectiva para optimizar las operaciones de lectura más comunes:

- **Contadores Precalculados**: Mantenimiento de contadores en documentos de usuario para followers, following y posts.
- **Feed Desnormalizado**: Almacenamiento de información frecuentemente accedida en la colección de feed.
- **Información de Perfil Embebida**: Datos de perfil almacenados directamente en el documento de usuario.

3.2.2. Gestión de Relaciones

Las relaciones entre entidades se manejan mediante referencias, optimizando para:

- **Escalabilidad**: Las referencias permiten que los documentos crezcan independientemente.
- **Flexibilidad**: Facilita la modificación de esquemas y la evolución de la aplicación.
- **Consultas Eficientes**: Permite agregaciones y búsquedas optimizadas.

3.3. Distribución de Datos

La distribución de datos en el ReplicaSet se optimiza para garantizar:

Consistencia

- Escrituras confirmadas por mayoría de nodos
- Lecturas consistentes desde secundarios
- Manejo de conflictos mediante timestamps

Disponibilidad

- Replicación automática entre nodos
- Failover transparente para aplicaciones cliente
- Recuperación automática de nodos caídos

Particionamiento Aunque el sistema actual utiliza un ReplicaSet sin sharding, está diseñado para facilitar la futura implementación de sharding horizontal:

- Claves de documento distribuibles (`_id`)
- Índices preparados para sharding
- Modelo de datos compatible con distribución

3.4. Estrategias de Indexación

La estrategia de indexación está diseñada para optimizar los patrones de acceso más comunes:

- **Índices Únicos**: Garantizan la unicidad de usernames y emails
- **Índices Compuestos**: Optimizan consultas multi-campo frecuentes
- **Índices Geoespaciales**: Facilitan búsquedas basadas en ubicación
- **Índices de Texto**: Permiten búsquedas eficientes en contenido

Esta estructura de datos distribuida proporciona una base sólida para las operaciones de la red social, permitiendo:

- Escalabilidad horizontal futura
- Alta disponibilidad de datos
- Rendimiento optimizado para operaciones frecuentes
- Flexibilidad para evolución del sistema

4. Endpoints de la API

4.1. Operaciones de Usuario

La API expone endpoints para gestionar operaciones relacionadas con usuarios:

4.1.1. Crear Usuario

```
POST /create_user
Content-Type: application/json

{
    "username": "string",
    "email": "string",
    "password": "string",
    "profile": {
        "full_name": "string",
        "bio": "string",
        "avatar_url": "string"
    }
}
```

Este endpoint:

- Valida datos de entrada
- Verifica unicidad de username y email
- Hashea la contraseña de forma segura
- Crea el documento de usuario en MongoDB
- Retorna el ID del usuario creado

4.1.2. Perfiles de Usuario

```
GET /user/{user_id}

Response:
{
    "user_id": "ObjectId",
    "username": "string",
    "profile": {
        "full_name": "string",
        "bio": "string",
        "avatar_url": "string",
        "created_at": "ISODate",
        "last_login": "ISODate"
    },
    "stats": {
        "followers_count": number,
        "following_count": number,
        "posts_count": number
    }
}
```

Funcionalidades:

- Recuperación eficiente de perfiles
- Proyección de campos según permisos

- Caché de perfiles frecuentes
- Estadísticas en tiempo real

4.2. Interacciones Sociales

La API proporciona endpoints para gestionar las interacciones entre usuarios:

4.2.1. Crear Posts

```
POST /create_post
Content-Type: application/json

{
    "user_id": "ObjectId",
    "content": "string",
    "media_urls": ["string"],
    "tags": ["string"],
    "location": {
        "type": "Point",
        "coordinates": [double, double]
    }
}
```

Características del endpoint:

- Validación de usuario existente
- Procesamiento de contenido multimedia
- Actualización atómica de contadores
- Indexación de contenido para búsqueda

4.2.2. Funcionalidad de Comentarios

```
POST /create_comment
Content-Type: application/json

{
    "post_id": "ObjectId",
    "user_id": "ObjectId",
    "content": "string",
    "parent_comment_id": "ObjectId"  // Opcional
}
```

El endpoint maneja:

- Comentarios en posts
- Respuestas a comentarios existentes
- Actualización de contadores de comentarios
- Notificaciones a usuarios relevantes

4.2.3. Relaciones de Follow

```
POST /follow_user
Content-Type: application/json

{
    "follower_id": "ObjectId",
    "following_id": "ObjectId"
}
```

Implementación:

- Verificación de usuarios existentes
- Validación de relación no duplicada
- Actualización atómica de contadores
- Generación de eventos de notificación

4.3. Operaciones del Sistema

La API incluye endpoints para monitoreo y mantenimiento del sistema:

4.3.1. Implementación del Health Check

```
GET /health

Response:
{
    "status": "Healthy",
    "mongo_status": {
        "primary": "connected",
        "secondaries": ["connected", "connected"],
        "replication_lag": [0, 1]
    }
}
```

El health check verifica:

- Conectividad con MongoDB
- Estado del ReplicaSet
- Lag de replicación
- Estado del pool de conexiones

4.3.2. Población de Base de Datos para Testing

```
POST /test/populate
Content-Type: application/json

{
    "users_count": number,
```

```
    "posts_per_user": number,
    "comments_per_post": number
}
```

Funcionalidades:

- Generación de datos de prueba realistas
- Creación de relaciones entre entidades
- Distribución estadística de interacciones
- Verificación de integridad de datos

4.3.3. Funcionalidad de Limpieza de Base de Datos

```
POST /test/clean

Response:
{
    "status": "success",
    "collections_cleaned": ["users", "posts", "comments", "likes", "
   follows"]
}
```

El endpoint:

- Limpia datos de prueba
- Mantiene configuración del sistema
- Verifica integridad post-limpieza
- Registra operación en logs

Todos los endpoints implementan:

- Manejo robusto de errores
- Validación de entrada
- Logging de operaciones
- Métricas de rendimiento
- Control de acceso basado en roles

5. Implementación Técnica

5.1. Error Handling

La implementación incluye un sistema robusto de manejo de errores que abarca múltiples niveles de la aplicación:

```
#[derive(Debug)]
pub enum AppError {
    // Errores de Base de Datos
    DatabaseError(mongodb::error::Error),
    ConnectionError(String),

    // Errores de Validaci n
    ValidationError(String),
    DuplicateKeyError(String),

    // Errores de Negocio
    NotFoundError(String),
    UnauthorizedError(String),

    // Errores del Sistema
    SystemError(String)
}
```

Estrategias de Manejo

- **Propagación Controlada**: Los errores se propagan de manera controlada mediante Result
- **Logging Estructurado**: Registro detallado de errores con contexto
- **Recuperación Automática**: Reintentos automáticos para errores transitorios
- **Respuestas HTTP**: Mapeo de errores a códigos HTTP apropiados

5.2. State Management

La gestión del estado de la aplicación se maneja mediante una estructura AppState que encapsula recursos compartidos:

```
pub struct AppState {
    pub db: Database,
    pub config: AppConfig,
    pub metrics: Arc<Metrics>,
}

impl AppState {
    pub async fn new(mongo_uri: &str, config: AppConfig) -> Result<
   Self, AppError> {
        let client_options = ClientOptions::parse(mongo_uri).await?;
        let client = Client::with_options(client_options)?;
        let db = client.database("social_media_db");

        Ok(Self {
            db,
            config,
            metrics: Arc::new(Metrics::new()),
        })
    }
}
```

Características Principales

- **Estado Compartido**: Acceso thread-safe a recursos compartidos
- **Configuración Centralizada**: Gestión unificada de configuración
- **Métricas en Tiempo Real**: Recolección de métricas de rendimiento
- **Conexión a Base de Datos**: Pool de conexiones administrado

5.3. Conexión a Base de Datos y Resiliencia

La conexión a MongoDB se implementa con énfasis en la resiliencia y rendimiento:

```
async fn setup_database(config: &Config) -> Result<Database, AppError>
    {
    let mut client_options = ClientOptions::parse(&config.mongo_uri).
   await?;

    // Configuraci n de timeouts
    client_options.connect_timeout = Some(Duration::from_secs(10));
    client_options.server_selection_timeout = Some(Duration::from_secs
   (15));

    // Configuraci n de pool de conexiones
    client_options.max_pool_size = Some(20);
    client_options.min_pool_size = Some(5);

    // Configuraci n de consistencia
    client_options.read_concern = Some(ReadConcern::majority());
    client_options.write_concern = Some(
        WriteConcern::builder()
            .w(mongodb::options::Acknowledgment::Majority)
            .build(),
    );

    let client = Client::with_options(client_options)?;
    Ok(client.database("social_media_db"))
}
```

5.4. Utilidades de Testing

El sistema incluye utilidades comprehensivas para testing:

```
#[cfg(test)]
mod tests {
    use super::*;

    async fn setup_test_db() -> TestDatabase {
        let mut options = ClientOptions::parse(TEST_MONGO_URI).await.
   unwrap();
        options.app_name = Some("test".to_string());

        TestDatabase::new(options).await
    }
```

```
    #[tokio::test]
    async fn test_user_creation() {
        let db = setup_test_db().await;
        let user_data = NewUser {
            username: "test_user",
            email: "test@example.com",
            password: "secure_password"
        };

        let result = create_user(&db, user_data).await;
        assert!(result.is_ok());
    }
}
```

Características del Testing

- **Base de Datos de Prueba**: Instancia dedicada para testing
- **Fixtures Automatizados**: Generación de datos de prueba
- **Limpieza Automática**: Restauración del estado inicial
- **Tests de Integración**: Pruebas end-to-end con base de datos real

Esta implementación técnica asegura:

- Alta calidad del código
- Mantenibilidad a largo plazo
- Facilidad de debugging
- Confiabilidad del sistema

6. Manejo de Consistencia y Durabilidad

6.1. Configuración de Garantías de Consistencia

La arquitectura implementa un modelo de consistencia fuerte mediante configuraciones específicas que garantizan la durabilidad y consistencia de los datos en el entorno distribuido:

Write Concern Mayoría La configuración de write concern asegura que las operaciones de escritura sean confirmadas por la mayoría de los nodos:

```
client_options.write_concern = Some(
    WriteConcern::builder()
        .w(mongodb::options::Acknowledgment::Majority)
        .build(),
);
```

Esta configuración proporciona:

- Garantía de durabilidad ante fallos de nodos
- Consistencia entre réplicas para operaciones posteriores
- Protección contra pérdida de datos en escenarios de failover

Read Concern Mayoría Las operaciones de lectura se configuran para garantizar consistencia:

```
client_options.read_concern = Some(ReadConcern::majority());
```

Este nivel de read concern asegura:

- Lecturas de datos confirmados por la mayoría de réplicas
- Prevención de lecturas sucias (dirty reads)
- Consistencia en operaciones distribuidas

7. Configuración de Resiliencia

7.1. Gestión de Timeouts

```
client_options.connect_timeout = Some(Duration::from_secs(10));
client_options.server_selection_timeout = Some(Duration::from_secs(15)
   );
client_options.max_idle_time = Some(Duration::from_secs(60));
```

Los timeouts se configuran para:

- **Connect Timeout (10s)**: Límite para conexiones iniciales
- **Server Selection (15s)**: Tiempo máximo para selección de servidor
- **Max Idle Time (60s)**: Duración máxima de conexiones inactivas

7.2. Políticas de Reintento

```
client_options.retry_reads = Some(true);
client_options.retry_writes = Some(true);
```

Las operaciones elegibles para reintento incluyen:

- Errores de red transitorios
- Timeouts de operaciones
- Errores de escritura no fatales
- Reconexiones post-failover

7.3. Monitoreo y Health Checks

Timeouts Operacionales

```
client_options.heartbeat_freq = Some(Duration::from_secs(15));
```

La implementación incluye:

- **Heartbeat**: Verificación periódica de nodos
- **Métricas de Replicación**: Monitoreo de lag y sincronización
- **Estadísticas Operacionales**: Seguimiento de latencia y throughput

```
async fn health_check_handler(
    app_state: web::Data<AppState>
) -> impl Responder {
    match app_state.db.run_command(doc! {"ping": 1}, None).await {
        Ok(_) => HttpResponse::Ok().json("Healthy"),
        Err(e) => HttpResponse::ServiceUnavailable().json(e.to_string
   ())
    }
}
```

El sistema implementa verificaciones de:

- Conectividad a MongoDB
- Estado del pool de conexiones
- Métricas de rendimiento
- Uso de recursos

7.4. Balanceo de Carga

Health Checks de Aplicación La aplicación implementa estrategias de balanceo en múltiples niveles:

```
client_options.selection_criteria = Some(
    ReadPreference::SecondaryPreferred {
        options: Some(ReadPreferenceOptions::default()),
    }
    .into(),
);
```

Características principales:

- Distribución de lecturas entre secundarios
- Escrituras centralizadas en primario
- Failover automático

Nivel de Aplicación

- Connection pooling optimizado (5-20 conexiones)
- Distribución de carga entre instancias
- Caché local para datos frecuentes

7.5. Error Handling

La aplicación implementa un manejo de errores multinivel:

- **Errores de Conexión**: Problemas de red o disponibilidad
- **Errores de Operación**: Fallos en operaciones específicas
- **Errores de Aplicación**: Problemas de lógica de negocio
- **Errores de Sistema**: Fallos de recursos o configuración

Estrategias de recuperación:

- **Circuit Breaker**: Prevención de cascada de fallos
- **Backoff Exponencial**: Reintento gradual
- **Fallback**: Rutas alternativas para operaciones críticas
- **Logging**: Registro detallado para análisis

8. Conclusiones

La implementación del sistema de base de datos distribuida MongoDB para nuestra aplicación de red social demuestra una arquitectura robusta y escalable que cumple con los requisitos de alta disponibilidad, consistencia y rendimiento. Los aspectos más destacados de la implementación incluyen:

8.1. Arquitectura Distribuida

- Implementación exitosa de un ReplicaSet MongoDB con esquema de prioridad 10-5-1
- Sistema de failover automático que garantiza continuidad operativa
- Balanceo de carga efectivo entre nodos primario y secundarios
- Aislamiento de red y seguridad mediante autenticación por keyfile

8.2. Garantías de Datos

- Consistencia fuerte mediante write concern y read concern mayoritarios
- Durabilidad asegurada por replicación síncrona entre nodos
- Prevención de pérdida de datos mediante confirmación distribuida
- Estrategias de de normalización optimizadas para lecturas frecuentes

8.3. Resiliencia y Monitoreo

- Sistema robusto de manejo de errores y recuperación
- Monitoreo continuo mediante heartbeats y health checks
- Connection pooling optimizado para rendimiento sostenido
- Políticas de reintento configuradas para operaciones críticas

8.4. Escalabilidad

- Diseño preparado para futuro sharding horizontal
- Modelo de datos optimizado para distribución
- Índices estratégicos para consultas frecuentes
- Arquitectura modular que facilita expansión

El sistema resultante proporciona una base sólida para las operaciones de la red social, con capacidad de crecimiento y adaptación a futuros requerimientos. La combinación de MongoDB como sistema de base de datos distribuida y Rust como lenguaje de implementación ha demostrado ser efectiva para crear una aplicación robusta y de alto rendimiento.

Las decisiones arquitectónicas tomadas en términos de consistencia, disponibilidad y tolerancia a particiones (CAP) han priorizado la consistencia y disponibilidad, mientras se mantiene la capacidad de escalar horizontalmente cuando sea necesario. La implementación actual sienta las bases para futuras mejoras y optimizaciones según evolucionen los requerimientos del sistema.