Laboratorio: Sincronización de almacenes KV de NATS mediante CRDT

Requisitos previos:

- Conocimientos básicos de NATS, JetStream y almacenes KV
- Familiaridad con Go o typescript
- Comprensión de conceptos básicos de sistemas distribuidos
- Docker y Docker Compose instalados

---

1. Objetivo

En este laboratorio diseñarás e implementarás un **agente de sincronización** capaz de mantener **varios almacenes NATS KV** consistentes entre sí usando un enfoque **CRDT (Conflict-Free Replicated Data Type)**.

Cada sitio ejecutará su propio servidor NATS y mantendrá su bucket KV.
Tu agente deberá:

- Detectar cambios locales en el bucket KV.
- Publicar dichos cambios como **operaciones CRDT** en un canal de replicación.
- Recibir operaciones de otros sitios.
- Resolver conflictos de forma determinista para garantizar **convergencia eventual**.

---

2. Escenario

Imagina que gestionas una aplicación desplegada en **sitios distribuidos (edge)**.
Cada sitio tiene su propio servidor NATS y un bucket KV local llamado `config`.
Los sitios pueden desconectarse temporalmente y seguir operando de forma autónoma.

Tu tarea: construir un agente (`nats-kv-syncd`) que garantice que, al volver a conectarse, **todos los sitios converjan hacia el mismo estado**, incluso si hubo actualizaciones concurrentes.

---

3. Arquitectura general

Cada sitio ejecutará:

1. Un **servidor NATS local** con JetStream habilitado.
2. Un **bucket KV** (por ejemplo, `config`).
3. El **agente de sincronización**, que:
   - Vigila los cambios en el KV local.
   - Publica operaciones CRDT a un **subject de replicación**, como `rep.kv.ops` .
   - Recibe operaciones de otros sitios y aplica las que ganen según las reglas CRDT.

subject de replicación:

```
rep.kv.ops
```

Diagrama conceptual

```
Error parsing Mermaid diagram!

Cannot read properties of null (reading 'getBBox')
```

---

4. Diseño CRDT

Para este laboratorio usaremos el **registro LWW (Last-Writer-Wins)**, el tipo más simple de CRDT.

Para cada clave del KV registrará:

- `value`
- `ts` : marca de tiempo lógica o física
- `node_id` : identificador del nodo de origen

Regla de resolución de conflictos:

```
(ts_remoto > ts_local) OR
(ts_remoto == ts_local AND node_id_remoto > node_id_local)
    → aplicar el valor remoto
else
    → conservar el valor local
```

De este modo, todas las réplicas llegarán al mismo resultado de manera determinista.

---

5. Preparación del entorno

Paso 1 · Docker Compose con dos servidores NATS

Se crea un entorno de pruebas que nos permita experimentar

Crea un archivo `docker-compose.yml`:

```yaml
version: "3.8"
services:
  nats-a:
    image: nats:latest
    command: ["-js"]
    ports:
      - "4222:4222"
  nats-b:
    image: nats:latest
    command: ["-js"]
    ports:
      - "5222:4222"
```

Lanza ambos servidores:

```
docker compose up -d
nats --server localhost:4222 ping
nats --server localhost:5222 ping
```

Paso 2 · Crear los buckets KV

```
nats kv add config --server localhost:4222
nats kv add config --server localhost:5222
```

Paso 3 · Crear el agente

Crea un proyecto Go llamado `nats-kv-syncd`.

Dependencias:

```
go mod init nats-kv-syncd
go get github.com/nats-io/nats.go
```

Ejecución:

```
./nats-kv-syncd \
  --nats-url nats://localhost:4222 \
  --bucket config \
  --node-id site-a \
  --rep-subj rep.kv.ops
```

---

6. Lógica del agente

6.1 · Vigilar cambios locales

Usa el watcher del KV:

```
watch, _ := kv.WatchAll()
for update := range watch.Updates() {
    fmt.Printf("Cambio local: %s = %s\n", update.Key(), string(update.Value()))
}
```

Cuando detectes un `PUT` o `DELETE`:

- Crea una operación CRDT en formato JSON:

```
{
  "op": "put",
  "bucket": "config",
  "key": "app/theme",
  "value": "dark",
  "ts": 1730542200,
  "node_id": "site-a"
}
```

- Publícala en `rep.kv.ops`.
  - Pregunta: cómo? donde?

6.2 · Recibir operaciones remotas

Suscríbete al canal:

```
nc.Subscribe("rep.kv.ops", func(m *nats.Msg) {
    var op Operation
    json.Unmarshal(m.Data, &op)
    if op.NodeID == *nodeID {
        return // ignorar eco local
    }
    // aplicar si gana según la regla CRDT
})
```

6.3 · Realizar el merge

Guarda los metadatos en memoria o en un bucket paralelo (`config_meta`).

- Justifica la respuesta
-

---

7. Prueba de partición

1. Ejecuta ambos agentes:

   ```
   ./nats-kv-syncd --nats-url nats://localhost:4222 --node-id site-a
   ./nats-kv-syncd --nats-url nats://localhost:5222 --node-id site-b
   ```

2. Detén uno de los servidores (simula desconexión):

   ```
   docker stop nats-b
   ```

3. En `site-a`:

   ```
   nats kv put config theme dark
   ```

4. En `site-b` (desconectado):

   ```
   nats kv put config theme light
   ```

5. Reinicia `nats-b`:

   ```
   docker start nats-b
   ```

✅ **Resultado esperado:**
Ambos sitios deben terminar con

```
theme = light
```

porque la actualización con `ts = 12` gana.

---

8. Desafíos

- Implementar **tombstones** para borrados seguros. (Requerido)
- Usar **reloj lógico o contador** por nodo. (Requerido)
- Añadir un **modo de recuperación** cuando algún mensaje se pierde. (Requerido)

- Sincronizar varios buckets.

---

9. Actualizaciones perdidas y recuperación

En sistemas distribuidos reales, los mensajes de replicación pueden perderse.
Esto puede ocurrir si un sitio se desconecta o si el hub no conserva las operaciones.
Nuestro agente debe garantizar la **convergencia eventual**, incluso en esos casos.

Hay **dos estrategias** principales para manejarlo:

9.1. Reproducción de operaciones (JetStream duradero)

Convierte el tema `rep.kv.ops` en un **stream persistente** (Cómo?))

Cada agente:

- Publica usando JetStream (`js.Publish(...)`).
- Se suscribe como **consumidor duradero** (`js.PullSubscribe(...)`).
- Al reconectarse, JetStream le deberá entregar los mensajes pendientes.

✅ Ventaja: ¿?
⚠️ Inconveniente: ¿?

9.2. Reconciliación basada en estado (fusión CRDT)

Cuando un sitio vuelve a estar en línea:

1. Lista las claves del KV local y remoto.
2. Compara marcas de tiempo y node IDs.
3. Aplica el valor ganador.

Esto puede ejecutarse periódicamente (“anti-entropy”) o al reconectar:

✅ Ventaja: ¿?
⚠️ Inconveniente: ¿?

9.3. Estrategia híbrida

La opción ideal combina ambos mecanismos:

- **JetStream** para la replicación en tiempo real.
- **Fusión periódica** como respaldo.

✅ Ventaja: ¿?
⚠️ Inconveniente: ¿?

9.4. Desafío adicional

Implementa un hilo o goroutine que ejecute la reconciliación cada 5 minutos.
Prueba desconectando un sitio, haciendo cambios y reconectándolo. ✅ Si tu CRDT está bien diseñado, el sistema acabará convergiendo sin intervención manual.

9.5 Desafío adicional 2

En NATS es posible mediante el uso de "leaf nodes" hacer que nats mismo sincronice mensajes. Propón una solución usando leaf nodes

10. Resumen

En este laboratorio habrás construido un sistema distribuido **capaz de sincronizar almacenes KV de NATS** mediante:

- Vigilancia de cambios locales (Watch)
- Replicación de operaciones (JetStream / NATS subjects)
- Fusión CRDT (Last-Writer-Wins)
- Recuperación ante mensajes perdidos

Has aprendido que:

- La **consistencia eventual** no depende de la entrega perfecta de mensajes,
- Sino de que las operaciones sean **conmutativas, asociativas e idempotentes**.

---

11. Cuestiones de autoevaluación

1. ¿Por qué los CRDT no necesitan consenso global para mantener consistencia?
2. ¿Qué ventaja aporta usar `node_id` como desempate?
3. ¿Qué ocurre si dos sitios tienen relojes de sistema desincronizados?
4. ¿En qué se diferencia la replicación basada en operaciones de la basada en estado?
5. ¿Qué garantiza la propiedad de idempotencia en una actualización CRDT?
6. ¿Por qué es útil combinar JetStream con reconciliación periódica?
7. ¿Qué pruebas podrías hacer para demostrar convergencia tras una partición?
8. ¿Qué notables diferencias encuentras con respecto a las garantias de cr-sqlite?

---

12. Entregables

- Código fuente (`main.go` y módulos)
- Archivo `README.md` con:
  - Cómo ejecutar el agente
  - Explicación de la lógica CRDT
  - Estrategia de almacenamiento de metadatos
  - Resultados de prueba
- Scripts varios que faciliten las pruebas

Todo lo anterior deberá desarrollarse en un git repo y la entrega consistirá en **el archivo TGZ de ese repo git**.