This is a simple nodejs package that emulates two nodes inA distributed system synchronizing their sqlite databases through a `todos` table.

Note that in this example, a sync server intermediates theExchange of changes between nodes.

Think how you would rearrange this code making it reactive if, instead of usingThe REST sync server, you were to use a NATS pub/sub.

0) Prereqs

- Node 20 installed
- macOS/Linux/WSL (Windows works too).
- Docker if you want to containerize later.

---

1) Scaffold the project

```
mkdir crsqlite-lab && cd crsqlite-lab
npm init -y
npm i better-sqlite3 @vlcn.io/crsqlite express
npm i -D nodemon typescript ts-node @types/node @types/express
npx tsc --init
```

You may want to use the following tsconfig.json file.

```
{
  "compilerOptions": {
    "target": "ES2022",
    "lib": ["ES2022"],
    "module": "nodenext",
    "moduleResolution": "NodeNext",
    "esModuleInterop": true,
    "strict": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "outDir": "dist",
    "rootDir": "src",
    "types": ["node"]
  },
  "include": ["src"]
}
```

`package.json` would look like this.

```
{
  "name": "crpruebasqlite",
  "version": "0.1.0",
```

```json
  "private": true,
  "type": "module",
  "engines": {
    "node": ">=20.17"
  },
  "scripts": {
    "dev": "tsx watch src/main.ts",
    "build": "tsc -p tsconfig.json",
    "start": "node dist/main.js"
  },
  "dependencies": {
    "@vlcn.io/crsqlite": "^0.16.3",
    "better-sqlite3": "^12.4.1",
    "dotenv": "^17.2.3",
    "express": "^5.1.0",
    "node-fetch": "^3.3.2"
  },
  "devDependencies": {
    "@types/better-sqlite3": "^7.6.13",
    "@types/express": "^5.0.5",
    "@types/node": "^24.9.2",
    "@types/node-fetch": "^2.6.13",
    "ts-node": "^10.9.2",
    "tsconfig-paths": "^4.2.0",
    "tsx": "^4.20.6",
    "typescript": "^5.9.3"
  }
}
```

> `@vlcn.io/crsqlite` ships the **extension binary path** you can load directly; we'll use `better-sqlite3`)

---

2) Minimal database helper (`src/db.ts`)

```ts
import SQLiteDB, { type Database } from "better-sqlite3";
import { extensionPath } from "@vlcn.io/crsqlite";

export function open(dbFile: string): Database {
  const db = new SQLiteDB(dbFile);
  // recommended pragmas from docs
  db.pragma("journal_mode = WAL");
  db.pragma("synchronous = NORMAL");
  db.loadExtension(extensionPath); // <- load cr-sqlite
  return db;
}
```

---

3) Define schema + mark table as CRR (`src/schema.ts`)

```
import type { Database } from "better-sqlite3";

export function applySchema(db: Database) {
  db.exec(`
    CREATE TABLE IF NOT EXISTS todos (
      id TEXT PRIMARY KEY NOT NULL,
      text TEXT NOT NULL DEFAULT '',
      done INTEGER NOT NULL DEFAULT 0
    );
    -- Turn table into a CRR so it can merge across peers:
    SELECT crsql_as_crr('todos');
  `);
}
```

> `crsql_as_crr('table')` upgrades a normal table into a CRR.

---

4) Peer script that writes & syncs (`src/peer.ts`)

This file simulates a device. It:

- opens a local DB file,
- writes a todo,
- **pulls** remote changes since last version,
- **pushes** its local changes.

```
import { open } from "./db.js";
import { applySchema } from "./schema.js";
import type { Database } from "better-sqlite3";
import express from "express";
import fetch from "node-fetch"; // if using Node<20, npm i node-fetch
// @ts-ignore — or add types if you like
const app = express();
app.use(express.json());

// Simple in-memory cursors (per remote "room")
const lastSent: Record<string, number> = {};
const lastSeen: Record<string, number> = {};

function dbVersion(db: Database): number {
  return (db.prepare(`SELECT crsql_db_version() AS v`).get() as any).v as number;
}

function siteId(db: Database): string {
  return (db.prepare(`SELECT hex(crsql_site_id()) AS id`).get() as any).id as string;
}

// Select only *local* changes since lastSent[room]
```

```
function selectChanges(db: Database, since = 0) {
  return db.prepare(`
    SELECT
"table","pk","cid","val","col_version","db_version","site_id","cl","seq"
    FROM crsql_changes
    WHERE db_version > ? AND site_id = crsql_site_id()
    ORDER BY db_version ASC
  `).all(since);
}

// Apply a changeset from remote
function applyChanges(db: Database, rows: any[]) {
  const stmt = db.prepare(`
    INSERT INTO crsql_changes

("table","pk","cid","val","col_version","db_version","site_id","cl","seq")
    VALUES (?,?,?,?,?,?,?,?,?)
  `);
  const trx = db.transaction((batch: any[]) => {
    for (const r of batch) {
      stmt.run(r.table, r.pk, r.cid, r.val, r.col_version, r.db_version,
r.site_id, r.cl, r.seq);
    }
  });
  trx(rows);
}

export async function runPeer(opts: {
  dbFile: string;
  name: string;
  restServer?: string; // e.g., http://localhost:4000
  room?: string;       // e.g., "demo"
}) {
  const { dbFile, name, restServer = "http://localhost:4000", room =
"demo" } = opts;
  const db = open(dbFile);
  applySchema(db);

  console.log(`[${name}] site: ${siteId(db)}`);

  // A helper endpoint so you can poke the peer to create data
  app.post("/add", (req, res) => {
    const id = crypto.randomUUID();
    db.prepare(`INSERT INTO todos (id, text, done) VALUES (?, ?, 0)`)
      .run(id, `[${name}] item ${new Date().toISOString()}`);
    res.json({ ok: true, id, dbVersion: dbVersion(db) });
  });

  // Pull changes from server (since lastSeen)
  app.post("/pull", async (_req, res) => {
    const since = lastSeen[room] ?? 0;
    const r = await fetch(`${restServer}/changes/${room}?since=${since}`);
    const rows = await r.json() as any[];
    applyChanges(db, rows);
    if (rows.length) {
      // Track the highest db_version we've *seen* from server
      const max = Math.max(...rows.map((r: any) => r.db_version));
      lastSeen[room] = max;
    }
    res.json({ pulled: rows.length, lastSeen: lastSeen[room] ?? 0 });
  });

  // Push local changes to server (since lastSent)
  app.post("/push", async (_req, res) => {
    const since = lastSent[room] ?? 0;
    const rows = selectChanges(db, since);
    if (rows.length) {
      const max = Math.max(...rows.map((r: any) => r.db_version));
      await fetch(`${restServer}/changes/${room}`, {
        method: "POST",
        headers: { "content-type": "application/json" },
        body: JSON.stringify(rows),
      });
      lastSent[room] = max;
    }
    res.json({ pushed: rows.length, lastSent: lastSent[room] ?? 0 });
  });

  app.get("/todos", (_req, res) => {
    const rows = db.prepare(`SELECT * FROM todos ORDER BY id`).all();
    res.json(rows);
  });

  const port = name === "peerA" ? 3001 : 3002;
  app.listen(port, () => console.log(`[${name}] up on http://localhost:${port}`));
}
```

The **four key ideas** (track last sent/seen versions; select changes with `crsql_changes`; insert changes back) mirror the official REST example and Whole-CRR guide.

---

5) Tiny REST sync server (`src/server.ts`)

This is a super-minimal “hub” that remembers nothing except the SQLite DB per “room”.

```
import express from "express";
import { open } from "./db";
import { applySchema } from "./schema";

const app = express();
app.use(express.json());

const dbs = new Map<string, ReturnType<typeof open>>();
```

```
function dbFor(room: string) {
  if (!dbs.has(room)) {
    const db = open(`./server_${room}.sqlite`);
    applySchema(db);
    dbs.set(room, db);
  }
  return dbs.get(room)!;
}

// GET: pull server changes since ?since
app.get("/changes/:room", (req, res) => {
  const since = Number(req.query.since ?? 0);
  const db = dbFor(req.params.room);
  const rows = db.prepare(`
    SELECT "table","pk","cid","val","col_version","db_version","site_id","cl","seq"
    FROM crsql_changes
    WHERE db_version > ?
    ORDER BY db_version ASC
  `).all(since);
  res.json(rows);
});

// POST: apply client changes
app.post("/changes/:room", (req, res) => {
  const db = dbFor(req.params.room);
  const rows = req.body as any[];
  const stmt = db.prepare(`
    INSERT INTO crsql_changes ("table","pk","cid","val","col_version","db_version","site_id","cl","seq")
    VALUES (?,?,?,?,?,?,?,?,?)
  `);
  const trx = db.transaction(() => {
    for (const r of rows) stmt.run(r.table, r.pk, r.cid, r.val, r.col_version, r.db_version, r.site_id, r.cl, r.seq);
  });
  trx();
  res.json({ applied: rows.length });
});

const PORT = 4000;
app.listen(PORT, () => console.log(`REST sync on http://localhost:${PORT}`));
```

---

6) Orchestrator to spin everything up (`src/main.ts`)

```
import { runPeer } from "./peer";
import "./server";
```

```
runPeer({ dbFile: "./peerA.sqlite", name: "peerA" });
runPeer({ dbFile: "./peerB.sqlite", name: "peerB" });
```

Run it:

```
mkdir src
# add the three files above
npm run dev
```

---

7) The lab flow (hands-on steps)

1. **Create offline writes**

```
# peer A adds one item
curl -XPOST http://localhost:3001/add
# peer B adds two items
curl -XPOST http://localhost:3002/add
curl -XPOST http://localhost:3002/add
```

2. **Show divergence**

```
curl http://localhost:3001/todos
curl http://localhost:3002/todos
```

3. **Sync A → server → B**

```
# push A up
curl -XPOST http://localhost:3001/push
# B pulls from server
curl -XPOST http://localhost:3002/pull
```

4. **Sync B → server → A**

```
curl -XPOST http://localhost:3002/push
curl -XPOST http://localhost:3001/pull
```

5. **Verify convergence**

```
curl http://localhost:3001/todos
curl http://localhost:3002/todos
```

Stretch goals

- **Conflict demo:** write the same `id` with different `text` on each peer, then sync; discuss **LWW** per column (or other CRDTs) and how to choose types.
- **WebSocket sync:** replace the REST server with the turnkey `@vlcn.io/ws-server` attached to an Express HTTP server; clients call `useSync(...)`. Great to show “live” sync.
- **Use `NATS` for syncing:** Make synching reactive by means of NATS pub/sub and watches. Need to include the nats library and run a NATS docker container as a server.