**NTHOEKHE SYSTEM ARCHITECTURE**

**Introduction**

The purpose of this document is to outline the system architecture for a peer-to-peer distributed database utilizing RESTful APIs, SQLite as the underlying database engine, and Flask for building the Python-based web application framework.

**System Overview**

The system is designed to provide a decentralized database solution where multiple peer nodes can collaboratively store and retrieve data using RESTful APIs. Each peer node hosts a SQLite database instance and exposes RESTful endpoints for data interaction. Flask, a lightweight WSGI web application framework, is used to develop and deploy the RESTful API endpoints.

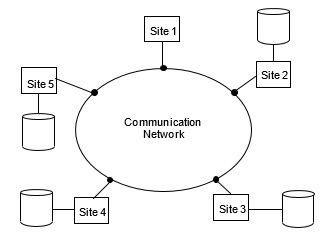
**Tools used:**

\*Python

 \*Flask

 \*SQLite

 \*RESTful APIs



**FRONTEND**

**Client Application:** This is a web application that interacts with the Ntshoekhe distributed database through RESTful APIs.

**RESTful APIs:** Designed to be scalable and to easily extend so that it allows the three nodes to communicate with each other.

It also allows applications to interact with different nodes, via HTTP protocols. So, an app running on either web, desktop, or mobile request a resource via HTTP methods and the response is returned normally in JSON format for interoperability. In our use case, our business logic includes:

\* Create operation-  adding patients to our database.

\*Read operation- gets a list of patients or a single patient.

\* Update operation- updating patients.

\* Delete operation- delete the patients from the database.

These operations will be done via HTTP requests.

**BACKEND**

**Flask:** Flask will be used to create the RESTful API endpoints. Each Flask instance will run on different nodes to handle incoming requests from clients.

**SQLite Database:** SQLite is a lightweight and is compatible with python and Flask. It will serve as the database management system. Each node will have its SQLite database instance to ensure that they share the same data, and the data is horizontally fragmented across these databases based on regions.

**Load Balancer (not yet implemented):** It is a mechanism that can evenly distribute requests across all nodes to ensure scalability and fault tolerance.

**ER Diagram**

Currently we have implemented only one table (Patients) and will soon start working on other tables.

**System Components**

**Peer Nodes**

Each peer node in the network hosts a local SQLite database instance.

Peer nodes communicate with each other using RESTful APIs for data exchange and synchronization.

Peers are responsible for storing and retrieving data items, as well as participating in query processing and data replication.

**RESTful APIs**

Expose a set of HTTP endpoints for CRUD (Create, Read, Update, Delete) operations on the database.

Endpoints are designed to be RESTful, following best practices for resource naming, HTTP methods, and response formats.

APIs provide interfaces for data synchronization, querying, and administrative tasks such as node registration and discovery.

**SQLite Database**

SQLite is used as the embedded relational database engine on each peer node.

Provides ACID (Atomicity, Consistency, Isolation, Durability) compliance for data integrity and transaction management.

**Flask Framework**

Flask is utilized to develop the web application framework for exposing RESTful APIs.

Provides routing, request handling, and response generation for the RESTful API endpoints.

**Functionality**

**RESTful API:** Flask will expose a RESTful API for interacting with the local SQLite database on each peer. The API will support standard HTTP methods (GET, POST, PUT, DELETE) for CRUD operations on data.

**Data Consistency:** The system will employ an eventual consistency model. When a data update occurs on one peer, it propagates the change to other peers eventually, ensuring consistency over time. Conflict resolution can be implemented for situations where concurrent modifications occur.

**Data Replication (Optional):** To improve data availability, data can be replicated across a configurable number of neighbouring peers.

**Technology Stack**

*Programming Language:* Python 3

*Web Framework:* Flask

*Database:* SQLite (lightweight, embedded database)

*Communication Protocol:* Custom Python implementation within Flask

**Deployment**

The system can be deployed on individual machines or containers. Each peer would run the Python application independently. A peer discovery service can be utilized to facilitate initial connection between peers and maintain a directory of active nodes.

**Security Considerations**

**Authentication**

Implement mechanisms to verify the identity of peers before allowing data access or updates.

**Authorization**

Enforce access control rules to restrict which peers can perform specific operations on the data.

**Data Encryption**

Encrypt data at rest (within SQLite databases) and in transit during communication between peers.

**Scalability**

The system is designed to scale horizontally by adding more peer nodes to the network.

Load balancing mechanisms ensure even distribution of data and workload across peer nodes.

Partitioning strategies may be employed to divide the database into smaller shards for efficient data management.

**Fault Tolerance**

The system is inherently fault tolerant. If a peer becomes unavailable, data remains accessible on other peers holding replicas (if enabled) or the original partition.