CSE 512 Portfolio
Distributed Database System For AVNs

Cheng Yen, Tsai
Computer Science
Arizona State University
Tempe, Arizona
ctsai61@asu.edu

***Abstract*—This project develops a fault-tolerant distributed database system for Autonomous Vehicle Networks (AVNs) to ensure reliable data management in dynamic data environments. By addressing challenges like data sharding, data consistency, and recovery, the system supports efficient and robust operations for AVN functionality.**

I. Introduction

As part of a CSE 512 distributed database system project, I collaborated with a team to build a fault-tolerant distributed database system for Autonomous Vehicle Networks (AVNs). These networks rely on seamless communication between autonomous vehicles and supporting infrastructure to improve traffic efficiency, safety, and sustainability. More importantly, AVNs need a data management system to ensure reliability, consistency, and scalability to deal with dynamic and intensive data.

Our project shows how distributed databases could meet these demands, focusing on optimizing data management. First, we implemented a data sharding mechanism based on the data location to reduce communication overhead and enhance CRUD (Create, Read, Update, Delete) operation performance. This approach minimizes latency and improves efficiency in real-time applications by ensuring data is stored closer to the source.

Second, we incorporated read-only replica nodes to optimize system performance for handling SELECT queries. These nodes offloaded read operations from primary nodes, which reduces their traffic. Moreover, our solution ensures data consistency and availability using fault tolerance mechanisms when network disruptions or hardware failures occur.

II. Challenges of Building A Distributed Database System

The development of a fault-tolerant distributed database system for autonomous vehicle networks (AVNs) required several challenges to be addressed. These challenges spanned communication reliability, efficient data sharding, data consistency, fault tolerance, and scalable deployment, as described below.

A. Ensuring Reliable Node Communication

Reliable communication between nodes is essential for system performance in a distributed database system. AVNs operate in highly dynamic environments, so timely data exchange is critical for decision-making. We must implement robust communication protocols to ensure seamless data transmission to deal with this intense real-time data. Additionally, a non-blocking architecture was required to simultaneously handle multiple requests, preventing delays and improving system responsiveness.

B. Data Sharding

To improve database access performance and reduce latency, we choose Sharding based on data location to minimize communication costs by ensuring that data frequently accessed by vehicles and infrastructure was stored locally. Query parsing gives an additional challenge, as the system had to accurately identify query types and forward them to the appropriate nodes. We direct SELECT queries to read-only replica nodes to balance read workloads while UPDATE and CREATE queries to update nodes to maintain data integrity.

C. Data Consistency

Node failures often address a significant risk to system reliability. Network disruptions or hardware malfunctions usually cause them. To solve this problem, we incorporate mechanisms based on the master-slave method mentioned by Grolinger et al. [2] to bring up read-only replicas when an update server fails. This fault-tolerance approach ensured that CRUD operations could continue uninterrupted, even during node failures, while maintaining data availability and integrity.

D. Scalable Deployment

Deploying the project across six nodes introduces additional challenges in our system design. We carefully deploy it in an environment that simulates real-world data distribution. This step is necessary to validate the system's ability to handle the scalability and complexity in real-world distribution environments, ensuring that the database can operate efficiently in production scenarios.

These challenges drove the designing and implementing a robust distributed database system, highlighting the need for solutions tailored to the unique demands of AVNs.These challenges drove the designing and implementing a robust distributed database system, highlighting the need for solutions tailored to the unique demands of AVNs.

TABLE I
IMPLEMENTATION DETAILS

Component	Technology/Tool	Purpose
Programming Language	Python	Socket programming and system logic implementation
Database	PostgreSQL	Local database on update nodes and replicas for sharded data storage and replication
Virtual Machines	GCP Compute Engine	Hosting six nodes (client, master, update nodes, and replica nodes).
Containerization	Docker	Creating isolated and portable environments for each node.
CI/CD Pipeline	GitHub Actions	Automating code deployment and updates to all nodes.

III. EXPLANATION OF THE SOLUTIONS

To address the challenges identified in developing a fault-tolerant distributed database system for Autonomous Vehicle Networks (AVNs), we implement several solutions as outlined below, also refer to"Table. I".

A. Ensuring Reliable Node Communication

We implement socket connections over the TCP protocol to achieve reliable node communication. TCP ensures data reaches its destination in the correct order. This provides reliable communication that we don't need to worry about data loss. Additionally, our system employs a non-blocking polling mechanism to handle high volumes of simultaneous connections efficiently. This mechanism allows the server to process multiple requests concurrently, ensuring responsiveness and scalability under heavy loads.

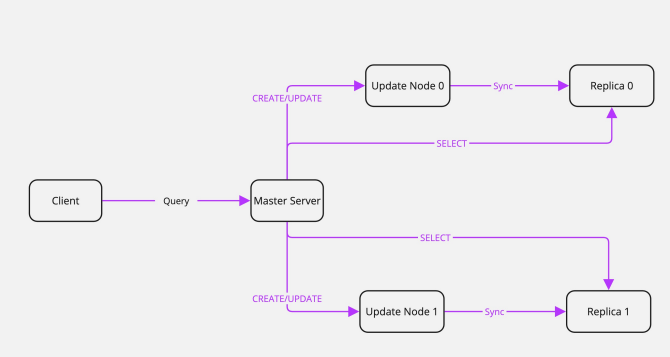

Fig. 1. Overall system design

B. Optimizing Data Sharding

We design an efficient query processor to handle data distribution and improve system performance. The master server, acting as a load balancer, parses incoming queries to determine whether the query includes the "state" column, which is used as the basis for hashing. The hashing method is similar to Hashed Sharding of MongoDB [1]. The master server also reconstructs the queries as necessary before forwarding them to the appropriate update or read-only nodes.

When deciding which node should handle a given query, we hash the abbreviated state value into one of two possible outcomes as shown in "Fig. 1". Queries with a hashed value of 0 are directed to read-only replica nodes for SELECT operations, while those with a hashed value of 1 will be sent to update nodes for CREATE and UPDATE operations. This mechanism ensures balanced workloads and reduces communication costs.

C. Maintaining Data Consistency

Data consistency is maintained by synchronizing replicas with the update node. When update servers process update-related queries, the same operations will be forwarded to the corresponding replica nodes to ensure data consistency. We implement this with master-slave replication mechanism [2]. Additionally, query logs are stored in both update nodes and replicas, enabling future recovery of missed operations when failure happens.

D. Fault-Tolerant Data Recovery

When dealing with an update server failure, our system ensures continued operation. Once a failure or timeout is detected from update servers, the master server will redirect all queries to replica nodes. After the update server is rebooted, the replica node will send the missed queries to update servers, ensuring that the operations performed during the downtime will be replicated. The roles of update and replica nodes are switched until another failure occurs, providing fault tolerance and uninterrupted service.

E. Scalable Deployment

For fast deployment and scalability, the project is containerized using Docker. This approach simplifies the setup and configuration process across nodes. Autonomous deployment is achieved through GitHub Actions, which automates the deployment process whenever pull requests are merged into the main branch. The containerized programs are deployed on Google Cloud Platform (GCP) Compute Engine virtual machines, enabling rapid scaling and testing in a distributed environment. The method is provided by Olususi Oluyemi in this article [3]

IV. DESCRIPTION OF THE RESULTS

A. Results

The functionalities of the project were tested locally using three machines. Machine 1 hosted the master server and client, Machine 2 was designated for sharding node 0 and its corresponding replica node, and Machine 3 managed sharding node 1 and its replica. The system performed successfully during all tested scenarios. When CREATE queries were submitted through the client, all sharding and replica nodes correctly created the necessary tables as shown in "Fig. 2". For INSERT queries, records were appropriately distributed to the sharding

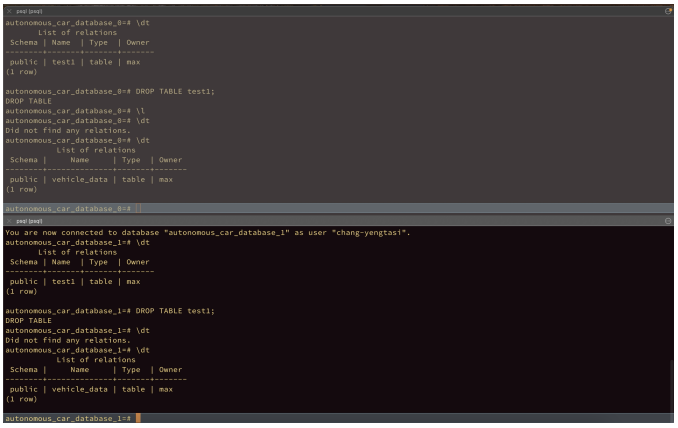

Fig. 2. Create Result

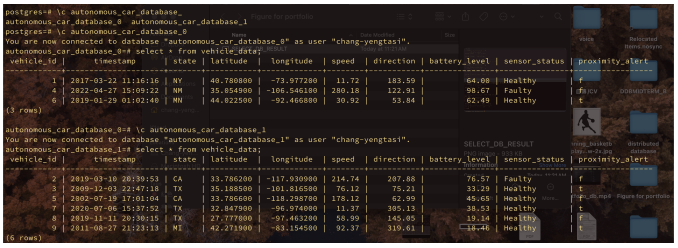

Fig. 3. Insert Result

and replica nodes based on the value of the state column, demonstrating proper data partitioning. In my test cases, I inserted 9 records of data with states NY, NM, MN, CA, TX, and MI. Based on the hashing algorithm, then will be stored in corresponding nodes as shown in "Fig. 3". SELECT queries without a WHERE state condition retrieved data from both replica nodes, while SELECT queries with a WHERE state condition fetched data only from the corresponding replica node as shown in "Fig. 4" and "Fig. 5". In terms of fault tolerance, the system effectively handled update node failures by allowing their corresponding replica nodes to take over all database operations. Once the update nodes were restored, they reverted to their roles as replica nodes, ensuring a seamless transition.

B. Future Works

Although the system operates successfully in a local environment, it has not been fully tested in a cloud-based deployment due to time constraints. However, the necessary infrastructure and deployment pipelines have been established on Google Cloud Platform (GCP), with automation facilitated through GitHub Actions for continuous integration and deployment (CI/CD). Furthermore, the current implementation lacks a user-friendly interface for input and output, which is crucial for improving the usability and accessibility of the system. Future work should focus on creating an intuitive interface and conducting comprehensive testing in a distributed cloud environment to validate the system's performance and scalability.

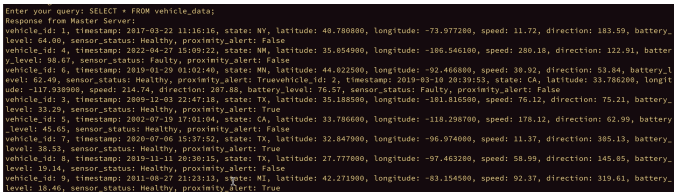

Fig. 4. Select all Result

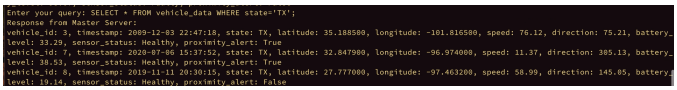

Fig. 5. Select state TX Result

V. Contribution

To ensure the efficient development of the system, tasks were distributed among the four team members: Yan-Syun Shen, Chun-Chih Yang, Ti-Mo Lin, and myself. Yan-Syun and Chun-Chih were responsible for designing and implementing the Update Server and Replica Server. At the same time, Ti-Mo focused on developing the Data Recovery and Role Switch mechanisms.

As part of the project, my responsibilities included system design, tool selection, and task distribution. My contributions to the development of the distributed database system were as follows:

- Designed the overall architecture of the Distributed PostgreSQL System using Python, ensuring scalability and fault tolerance.
- Implemented the query processor to efficiently parse and route queries to the appropriate nodes.
- Developed the multiple socket connection mechanism using non-blocking I/O `poll()`, see [4], for handling concurrent requests efficiently.
- Deployed the system on Google Cloud Platform (GCP) Compute Engine virtual machines using Docker for containerization and GitHub Actions to enable seamless Continuous Integration and Continuous Deployment (CI/CD).

VI. Takeaways

Working on this project provided invaluable learning opportunities, allowing me to develop technical and interpersonal skills that will undoubtedly benefit my career. On the technical side, I gained hands-on experience building a distributed database system by integrating multiple local databases into a cohesive architecture. Network programming using sockets was one of the first challenges I encountered. Addressing issues such as concurrent connections, race conditions, and potential deadlocks required in-depth research and practical application, enabling me to build a strong understanding of these foundational concepts.

Additionally, I designed and implemented critical components of the distributed system, including a global query processor, sharding mechanism, replication mechanism, and recovery mechanism, in collaboration with my team. These fea-

tures deepened my understanding of how distributed database systems work in practice, connecting academic concepts with real-world applications. This process is exciting and rewarding to me.

Furthermore, I acquired modern deployment skills by utilizing CI/CD pipelines with Docker and GitHub Actions. This experience familiarized me with deploying software efficiently to production environments, a crucial skill for participating in any programming-related projects in today's industry.

Beyond technical expertise, this project also improves my soft skills. As the lead for system design, workflow planning, and task distribution, I took on significant leadership responsibilities. Effective communication was essential as I coordinated with team members, monitored their progress, and facilitated problem-solving discussions. Organizing and leading meetings during critical moments taught me to ensure team alignment and productivity. This leadership experience was invaluable, equipping me with transferable skills to any collaborative setting.

In summary, this project enhanced my technical knowledge and proficiency. It strengthened my ability to lead and collaborate within a team, preparing me for future challenges in both academic and professional environments.

REFERENCES

[1] Mongo DB, "Hashed Sharding", https://www.mongodb.com/docs/manual/core/hashed-sharding/.

[2] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari and Miriam AM Capretz,"Data management in cloud environments: NoSQL and NewSQL data stores" ournal of Cloud Computing: Advances, Systems and Applications 2013, 2:22 http://www.journalofcloudcomputing.com/content/2/1/22

[3] Olususi Oluyemi, "How to build a CI/CD pipeline with Docker" https://circleci.com/blog/build-cicd-pipelines-using-docker/

[4] Tharun Appu, Event Loop and Non-Blocking I/O , https://medium.com/@tharunappu2004/event-loop-and-non-blocking-i-o-d6a5ffcbd70d