Storage fabric for autonomous collaborative edge devices

Pantelis Ypsilantis
Department of Informatics
Aristotle University Of Thessaloniki
Thessaloniki, Greece
panteliy@csd.auth.gr

Theodoros Toliopoulos
Department of Informatics
Aristotle University Of Thessaloniki
Thessaloniki, Greece
tatoliop@csd.auth.gr

Anastasios Gounaris
Department of Informatics
Aristotle University Of Thessaloniki
Thessaloniki, Greece
gounaria@csd.auth.gr

***Abstract*—We propose a distributed fabric over a set of autonomous databases over an edge network that enables decentralized querying, addressing privacy concerns and enhancing performance. Employing SQLite instances per edge device, Docker, Kafka, and the Spring Framework, the system demonstrates efficient synchronization and scalability in querying.**

***Index Terms*—Edge computing, SQLite, peer-to-peer, decentralized storage**

I. Introduction

In today's rapidly advancing technological landscape, traditional data handling pipelines often struggle with inefficiency. Edge computing [1] is a transforming paradigm offering efficient and decentralized data processing capabilities. It accounts for applications closer to data sources, like IoT devices or local edge servers, promising benefits such as improved latency times and enhanced bandwidth availability. However, to date there is no system for autonomous edge devices supporting CRUD operations locally, but also allowing each device to join a network so that it can receive queries even for remote data. Imagine, for example, a telemetry application, where each edge device locally stores its monitored information, such as CPU and memory load, and a user can submit queries to any known device to retrieve data referring to all devices in the network. To support such applications, a novel storage fabricate is proposed, designed to store data locally, with the usage of a local database like SQLite, while enabling seamless data access and retrieval across distributed edge nodes. The complete solution is available as an open-source system[1].

II. System Description

The proposed system architecture for the edge computing framework employs a combination of advanced technologies to ensure high performance, scalability, and security. More specifically, each node hosts a dedicated SQLite [2] instance to locally store and manage data, ensuring that data generated by each node is preserved and can be accessed with minimal latency. Additionally, the framework includes a communication layer to facilitate interactions between different nodes in the network, enabling the execution of distributed queries. The nodes are connected in a peer-to-peer manner [3], [4] and employ a distributed hash table (DHT) [4], [5] to achieve their synchronization. A visualization of the architecture can be seen in Figure 1.

[1]https://github.com/pantelis17/StorageFabricForEdgeDevices

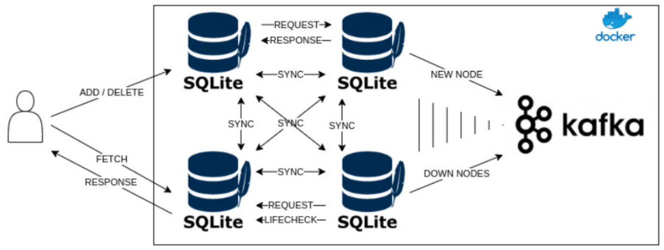

Fig. 1. Architecture of the implemented framework

The system is implemented using Java 21, Spring 3.1 [6], JPA [7] and Hibernate 6 [8], providing a robust foundation for the application. Furthermore, Apache Kafka [9] is integrated for its event-driven architecture, which enables seamless communication between distributed nodes and autonomous task execution without extensive coordination. This integration is critical for maintaining system resilience and scalability.

Main Functionalities. The framework supports several key functionalities, which are called flows, that are all essential for efficient edge computing operations. Currently, it contains seven flows: "*New Node*", "*Sync*", "*Add*", "*Delete*", "*Fetch*", "*Lifecheck*" and "*Down Nodes*". With the usage of these flows, the framework is capable of supporting the following functionalities:

- ***New Node***: This flow is initiated when a new node inserts the fabric (i.e., joins the network). This entails announcing its existence and initiating a fast warm-up, including syncing its data with the rest of the nodes.
- ***Lifecheck***: it verifies the status of a node, i.e. active and inactive, using periodic checks.
- ***Down Nodes***: This flow is responsible for removing an inactive node from the network. Along with the node, the flow removes also the metadata concerning the specific node.
- ***Sync***: It allows the nodes to exchange metadata regarding their locally stored data. After the sync process, each node has the necessary information to fetch a specific

data point from its source node.

- ***Add* new metrics to a node**: This flow allows users to add a metric, e.g., CPU usage, from a container or pod to the database of a target node.
- ***Delete* an existing metric from a node**: It enables users to remove a metric from the database of the target node. This flow ensures that the deletion is propagated to other nodes, maintaining data integrity.
- ***Fetch* values**: This flow allows users to retrieve metrics from a node even if the data are stored on another node of the cluster. Also, if the user specifies a specific metric type, then the system can return the metrics from every node in the network which, according to its metadata, contains metrics of the specified type.

More details along with evaluation experiments can be found in the provided GitHub repository.

Extensibility. The architecture is designed to be extensible, allowing for future enhancements and integrations. Additionally, it is configurable to adapt to different requirements. Here are some key extensibility points:

- **Modular Components**: The architecture is designed with modular components, allowing individual modules to be updated or replaced without affecting the entire system. This facilitates easy integration of new features and technologies. For example, it is easy to replace SQLite with any other database as it requires changes only in one class of the codebase.
- **Configurable Parameters**: The synchronization mechanism between nodes can be adjusted to balance load and performance. This allows the system to maintain efficiency even as the number of nodes increases significantly. Also, the cutoff threshold for the response time, currently set to 4 seconds by default, can be adjusted if the system is expected to return a vast amount of data.
- **Enhanced Security Measures**: Future enhancements can include implementing security protocols, such as user authentication and encryption, to protect sensitive data and ensure secure communications within the network.
- **Customizable Data Management**: The data management strategies can be tailored to meet specific application needs, including data aggregation, real-time analytics, and optimized data querying mechanisms, e.g., use of semi-joins.

III. Demo Scenario Description

During the demonstration of our application's performance and response validity, a complete workflow of setup and execution will be presented. Docker [10] will be used for the deployment of the required volumes, network and nodes. Initially, Kafka services will be started, and a node will be initiated. Metrics will be added through Postman requests, followed by the initiation of an additional node. This new node will send a Kafka message, it will be processed by the first node, which will then send back an abstract description of its data. Metrics will be added to the new node, and requests for memory usage metrics will be performed. The first node will return only its metrics due to the lack of synchronization, but the second node will return metrics from both nodes because of initial synchronization.

After waiting a few minutes for synchronization, identical results will be returned from both nodes. Another node will then be started, and the Kafka message will be sent. One of the two active nodes will receive the message and synchronize with the new node, producing the same results as the other nodes. After shutting down the second node and resending the request from the new node, the second node will fail to respond. A lifecheck request will be sent, and upon failure, a Kafka message will be broadcasted to all nodes, identifying the second node as down. Resending the request will confirm the second node was skipped.

The second node will be restarted, retrieve its data from the previous database, and send a Kafka request. Again, one of the two active nodes will receive it and send a sync request. Until synchronization is complete, only the second node and the node who receive the Kafka message will return full results. After synchronization, the remaining node will update its metadata, and requests will be accurate. A new cpu usage metric will be added to the third node, and only this node will return results until synchronization, after which all nodes will return the full result.

Finally, we will delete the cpu usage metrics from the third node. After synchronization, cpu usage will be removed from the other nodes' metadata. Re-adding the metric and sending a request will result in the third node responding as expected. During synchronization, information about cpu usage will be sent to the first and second nodes but will be ignored due to the active deletion constraint. This will persist through the second sync request, but by the third sync, all nodes will respond correctly to cpu usage requests.

IV. Conclusion

We present a novel approach to implementing a distributed storage fabric system over a set of independent databases, designed to address the inefficiencies and challenges of traditional centralized data handling pipelines and the absence of a distributed database for autonomous collaborative edge devices. By leveraging SQLite and Kafka, the proposed system achieves efficient data processing, enhanced security, and scalability. The modular architecture allows for easy integration of new technologies and future enhancements, making it a versatile solution for various edge computing applications.

References

[1] "Edge computing architecture: A practical guide," https://www.run.ai/guides/edge-computing/edge-computing-architecture.
[2] "Sqlite home page," https://www.sqlite.org/.
[3] J. Hoek, Y. Dai, E. Lai, and T. Zhang, "Peer to peer architecture."
[4] D. Suo, "Peer-to-peer systems and distributed hash tables."
[5] "Distributed hash tables (dhts)," https://www.tutorialspoint.com/distributed-hash-tables-dhts.
[6] "Spring boot," https://spring.io/projects/spring-boot.
[7] "Spring data jpa," https://spring.io/projects/spring-data-jpa.
[8] "6.0 series - hibernate orm," https://hibernate.org/orm/releases/6.0/.
[9] "Apache kafka," https://kafka.apache.org/.
[10] "Docker," https://www.docker.com/.