UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

A Distributed Key-Value Database

Nikolaos Reskos
Academic ID: 7115112300030

Course: *Big Data Management (M111)*
Semester: *Fall Semester 2024*

Contents

Abstract

This project implements a distributed, fault-tolerant Key-Value (KV) database system that utilizes a trie-based storage structure for efficient data management. The implementation encompasses three main components: a data generation module for creating test datasets, a key-value broker for managing distributed operations and replication and multiple key-value servers for data storage and retrieval. The system successfully implements k-way replication for fault tolerance, supports nested data structures, and provides efficient query capabilities through a trie-based architecture.

1. System Architecture

The distributed key-value store consists of three primary components working in harmony to provide data generation, storage and retrieval capabilities. The developed architecture enables us to handle specific aspects of the system's functionality while maintaining robust error handling under various operational conditions.

1.1. Data Generation Module

The data generation module serves as the foundation for testing and developing the distributed key-value store system. This component generates syntactically correct data, based on the assignment's specified format, providing a dataset for system validation and performance testing. At its core is the `generate_nested_value` function, which implements a recursive approach to generate nested key-value structures.

The recursive implementation works by maintaining a current depth counter and a target depth, which is randomly selected between 0 and the maximum depth (specified by the user) individually for each generated line. At each level of recursion, the function randomly determines the number of keys that will contain nested structures and makes recursive calls to generate these nested values. Two key conditions control the structure generation: if the user specifies a maximum number of keys equal to 0, the function generates an empty object *"{}"* as the top-level value with no nesting. Otherwise, the recursion continues until the current depth matches the target depth for that line, at which point the function generates terminal values according to the specified types. This approach ensures that each generated line adheres to both the depth constraints and the key count parameters specified by the user.

Data formatting is handled through careful string manipulation. Values are generated according to their specified types: strings are wrapped in quotes and their length is constrained by the maximum character limit specified by the user, integers are randomly generated between 0 and 1000 and floats are also randomly generated in the same range but formatted with two decimal places. The module maintains consistent formatting by joining key-value pairs with semicolons, ensuring that the output matches the required format.

1.2. Key-Value Broker

The key-value broker acts as the central coordinator of the distributed system, managing server connections and implementing the replication strategy. This component handles several critical aspects of the system's operations, including data distribution, server health monitoring and query routing.

1.2.1. Replication Strategy and Fault Tolerance. The broker implements k-way replication by randomly selecting servers from a list provided in a server file (given as a command-line argument). The implementation follows a two-phase approach: first, the broker attempts to connect to all servers in the file to determine which ones are available, maintaining a set of active ones. Then, for each line of data, it randomly selects k servers from the active set for data distribution.

The system keeps track of which servers were initially active during the data distribution phase, as these servers contain the actual data. A continuous monitoring mechanism checks the health of these initial servers. The k-way replication strategy ensures system availability until k or more of the initially active servers become unavailable. This design choice guarantees that enough replicas remain accessible for reliable data retrieval.

1.2.2. Query Processing. After completing the data distribution, the broker enters an interactive mode in which it accepts three types of commands from the keyboard: GET, DELETE, or QUERY. Each command is transmitted to all currently active servers that were part of the initial distribution, and the broker processes their responses accordingly.

For both GET and QUERY commands, the broker attempts to retrieve the requested data from these servers, returning the first successful response found. The key difference between these commands lies in their functionality: GET retrieves data for exact high-level key matches, while QUERY supports nested path traversal using dot notation to access nested values within the stored data structures.

The DELETE command has stricter execution requirements. It can only proceed if all servers that were initially active during the data distribution are still available. This constraint ensures consistency across the distributed system by preventing partial deletions that could lead to data inconsistencies.

For each command execution, the broker first checks the current status of the active servers to ensure system health before proceeding with the operation. This continuous monitoring of health helps maintain the reliability of the system and the consistency of data throughout all operations.

1.3. Trie Implementation

The trie implementation consists of two main classes: TrieNode and Trie. Each TrieNode maintains a dictionary of child nodes and a value field that can store any type of data. This structure allows for efficient character-by-character storage of keys, where each node in the path from root to terminal node represents one character of the stored key.

The use of a dictionary for child nodes (implemented as a hash table in Python) is a key design choice, as it provides O(1) constant-time access when checking a specific character's node. This means that during any traversal operation, finding the next character's node happens in constant time regardless of how many children a node has, making the operations significantly more efficient.

Basic operations of the Trie class include `insert` and `search`. The `insert` operation builds the trie structure character by character, creating new nodes as needed and storing the corresponding value in the final node. The `search` operation traverses the trie following the input key's characters and returns the stored value if the key exists.

The Trie class also implements two more operations: `delete` and `query_path`. The `delete` operation implements a recursive approach that not only removes the terminal node's value, but also cleans up any nodes that become unused after deletion. The `query_path` operation extends the basic search functionality by supporting the dot-notation path traversal for nested structures. Firstly, it locates the top-level key using the trie structure, then it performs constant-time dictionary lookups for each subsequent path component in the nested value structure. Since both the trie traversal and hash table-based dictionary lookups are O(1) operations at each step, this provides efficient access to deeply nested values.

1.4. Key-Value Server

The Key-Value Server implementation provides a socket-based server that handles client connections and manages data storage using a trie data structure. The server uses Python's `socket` module to create a TCP socket and binds it to a specified IP address and port. Once bound, the server listens for incoming connections and handles them sequentially in a blocking I/O model. When a client connects, the server spawns a new socket dedicated to that client, allowing for independent processing of the client's requests through the `handle_client` method.

The server maintains a clear separation between command processing and data management, implementing four basic operations: PUT, GET, DELETE, and QUERY. Each command is first validated and normalized, with quotes and whitespace handled consistently. The parsing of key-value pairs is separated into its own method, which first splits the input on the initial colon and then processes the value part. Error handling is implemented throughout the server, with specific error messages for various failure scenarios such as invalid IPs, unavailable ports, malformed commands, invalid data formats, or missing keys.

A key design choice was the utilization of Python's built-in JSON functionality for parsing and formatting the nested value structures. This approach converts the input strings into Python dictionaries for storage in the trie, enabling efficient nested value access through dictionary operations during QUERY commands. The conversion between the assignment's semicolon-separated format and JSON format is accomplished through a simple string replacement (semicolons to commas) before parsing, and the reverse transformation when formatting output. This solution avoids the complexity of character-by-character parsing while ensuring reliable handling of nested structures and providing a natural way to store and access nested data.

In the process of testing the Key-Value store with complex data structures, an issue appeared when handling large amounts of data, which occurred in two scenarios: when dealing with deeply nested structures (around 9-10 levels of nesting) or when allowing a large number of maximum keys per level (greater than 10). The problem showed up as JSON parsing errors at specific character positions (around position 4080), suggesting limitations in handling large data chunks during socket transmission. While the data generation itself was correct and produced valid nested structures, the default socket buffer size was insufficient for transmitting these large data structures in a single operation.

The solution implemented was to modify both the server and broker components to handle data transmission in chunks rather than attempting to process everything at once. This was achieved by implementing a buffered reading approach in the server's `handle_client` method and a corresponding chunked sending mechanism in the broker's `send_to_server` method. These modifications successfully resolved the data transmission limitations, allowing the system to handle complex nested structures without data loss or parsing errors.

2. Conclusion

The implemented distributed key-value store successfully meets all requirements specified in the assignment, providing a robust and efficient solution for distributed data storage and retrieval. The system's architecture combines both fault tolerance through k-way replication, ensuring data availability even when some servers fail, and efficient data retrieval through the optimized trie-based storage system with dictionary-based value storage.

Future enhancements could focus on scaling the system's capabilities. An extension would be supporting multiple broker connections, which would require significant architectural changes. In such a scenario, the server implementation would need to handle concurrent requests, necessitating proper thread synchronization mechanisms for the shared trie structure. Additionally, the replication strategy could be enhanced to include dynamic rebalancing when servers join or leave the system, in order to maintain optimal data distribution across the active servers and ensure that the k-replication factor is maintained when sufficient servers are available.