Design Optimization of Computing Systems

Assignment 3

DOCS-DB (Part-C)

Submitted by:

Bratin Mondal (21CS10016)
Swarnabh Mandal (21CS10068)
Somya Kumar (21CS30050)

Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur

1 System Architecture

The architecture of the Redis-compatible server is designed to maximize throughput and scalability while adhering to the RESP-2 protocol. The system incorporates asynchronous programming paradigms, efficient storage mechanisms, and a robust connection management layer. Below are the key components:

1.1 Connection Management Layer

The connection management layer facilitates interaction between clients and the server, handling all connections with a focus on scalability and low latency. By utilizing efficient socket-based implementations and non-blocking I/O, this layer ensures the server can manage a high volume of concurrent connections seamlessly.

- The server operates on a single TCP port to accept client connections using a socket-based implementation.
- The connection layer leverages the `asyncio` library for non-blocking I/O, enabling simultaneous processing of multiple client requests without the overhead of thread or process-based context switching.
- An independent coroutine represents each connection managed within the server's event loop.

1.2 RESP-2 Protocol Parsing

Parsing and generating commands in RESP-2 format is central to the server's Redis compatibility. This subsection explores how the server interprets client requests, extracts command details, and encodes responses for communication, ensuring adherence to the RESP-2 standard.

- The server adheres to the RESP-2 protocol for communication. Incoming client requests are parsed to extract commands (e.g., `GET`, `SET`, `DEL`) and their respective arguments.
- Responses are encoded in RESP-2 format (e.g., simple strings, bulk strings, or errors) and returned to the client.

1.3 Storage Layer

At the core of the server's functionality lies the storage layer, which is responsible for data persistence and retrieval. Using an LSMTree-based design, this layer provides robust performance for both read and write operations while ensuring data durability and optimized resource utilization.

- An **LSMTree-based storage engine** serves as the backend, optimized for high write performance and low read latency.
- The LSMTree's write-ahead log and sorted string tables ensure durability and efficient key-value pair retrieval.

1.4 Concurrency and Parallelism

The ability to handle multiple simultaneous client requests efficiently is achieved through the server's concurrency and parallelism strategies. This subsection discusses how lightweight asynchronous operations and multi-threaded execution work in tandem to maximize throughput and scalability.

- **Concurrency Layer:** Enables lightweight, non-blocking handling of multiple client requests.
- **Parallelism Layer:** Distributes workload across CPU cores using multi-threading or multiprocessing.

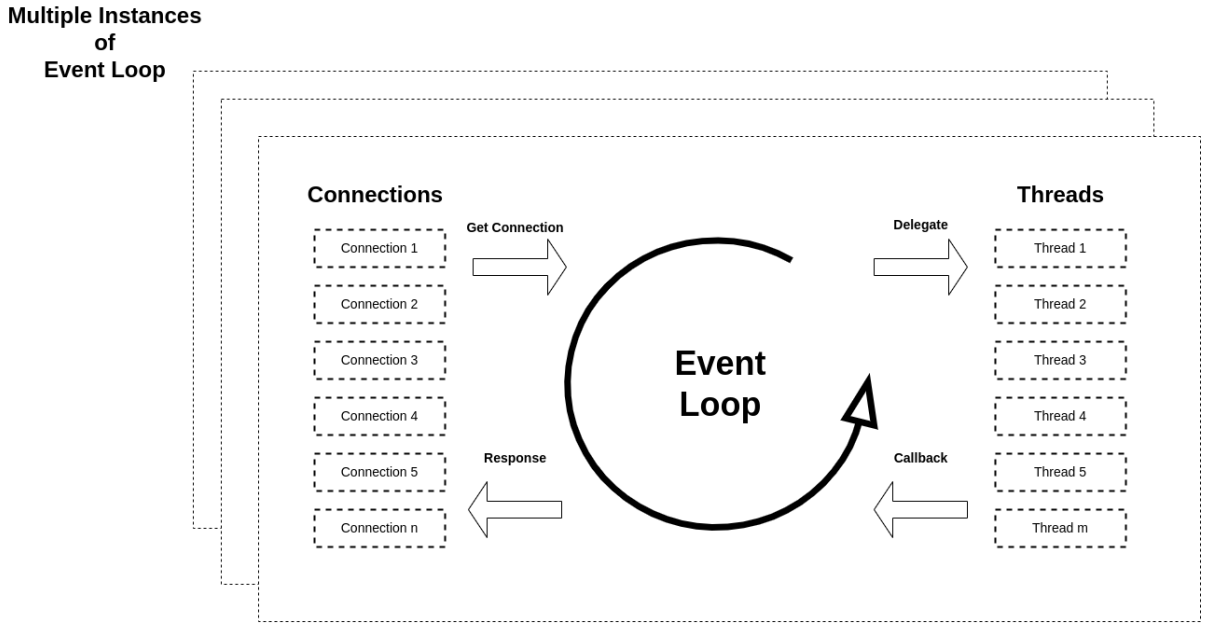

Figure 1: System architecture for the server

2 High Concurrency and Parallel Connection Handling

The server's design leverages both `asyncio`-based asynchronous I/O and a worker-thread model for parallel execution to achieve high concurrency and parallelism.

2.1 Concurrency via `asyncio`

2.1.1 Asynchronous Event Loop

The asynchronous event loop underpins the server's concurrency model, enabling it to handle I/O-bound tasks without blocking operations. By employing coroutines and an event-driven architecture, the server efficiently manages thousands of connections concurrently.

- The server uses the `asyncio` event loop to handle I/O-bound tasks without blocking.
- Each client connection is represented by a coroutine that reads requests, processes commands, and writes responses asynchronously.
- The `uvloop` library (an alternative event loop) is used to boost performance by leveraging low-level system calls (e.g., `epoll` or `kqueue`).

2.1.2 Non-Blocking I/O

Efficient I/O handling is critical for the server's performance. Non-blocking I/O ensures that the server processes requests and sends responses without waiting, minimizing latency and allowing for smooth handling of numerous simultaneous connections.

- `StreamReader` and `StreamWriter` objects are used for efficient reading and writing to client sockets.
- Non-blocking I/O ensures that no client monopolizes the server, allowing seamless handling of thousands of simultaneous connections.

2.1.3 Fair Scheduling

To ensure equitable resource allocation, the server employs fair scheduling mechanisms within its asynchronous architecture. These mechanisms guarantee that no single task monopolizes the server's resources, maintaining consistent performance across all client requests.

- The `asyncio` scheduler employs a cooperative multitasking model, ensuring fair distribution of CPU time across tasks.
- Coroutines yield control to the event loop when waiting for I/O, ensuring no task starves.

2.2 Parallelism via Worker Threads

2.2.1 Threaded Event Loops

For effective parallelism, the server uses threaded event loops that distribute client connections across multiple worker threads. This design ensures high CPU utilization and prevents bottlenecks, allowing the server to maintain performance even under heavy loads.

- The server spawns a fixed number of worker threads (typically equal to the number of CPU cores).
- Each thread runs an independent `asyncio` event loop, managing its subset of client connections.
- This design utilizes multi-core processors effectively, ensuring maximum CPU utilization.

2.2.2 Load Balancing

Load balancing mechanisms are essential for distributing incoming client connections evenly across threads or processes. By avoiding hotspots and ensuring an equitable workload, this subsystem maximizes server throughput and minimizes latency.

- New connections are distributed among worker threads using a load-balancing mechanism (e.g., round-robin or least-loaded worker selection).
- This prevents any single thread from becoming a bottleneck under heavy load.

2.2.3 Thread Safety

Thread safety is critical for maintaining data integrity and consistency in a multi-threaded environment. This subsection delves into the techniques used to synchronize shared resources and prevent conflicts in concurrent operations.

- Shared resources, such as the LSMTree, are synchronized using thread-safe primitives (e.g., locks or queues).
- Write operations are serialized to maintain consistency, while read operations leverage the LSMTree's immutable SSTable structure for high-speed access.

3 Performance Benefits

The server architecture achieves remarkable scalability, low latency, and optimal CPU utilization through its well-integrated concurrency and parallelism strategies. This subsection highlights the key performance advantages that make the server resilient and capable of handling demanding workloads efficiently.

- **Scalability:** The combination of `asyncio`-based concurrency and multi-threaded parallelism ensures the server scales to thousands of connections with minimal resource overhead.

- **Low Latency:** Non-blocking I/O and fair scheduling minimize request-response latency, even under heavy workloads.
- **CPU Utilization:** Multi-threaded event loops ensure all CPU cores are utilized effectively, avoiding bottlenecks on single-threaded workloads.

By combining asynchronous programming, efficient I/O handling, and multi-threaded parallelism, the server achieves a robust and performative architecture that meets high concurrency demands with low resource utilization.