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

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:

Connection Management Layer

- 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.

RESP-2 Protocol Parsing

- 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.

Storage Layer

- 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.

Concurrency and Parallelism

The architecture is divided into two layers to handle concurrent client connections and achieve parallel execution:

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

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.

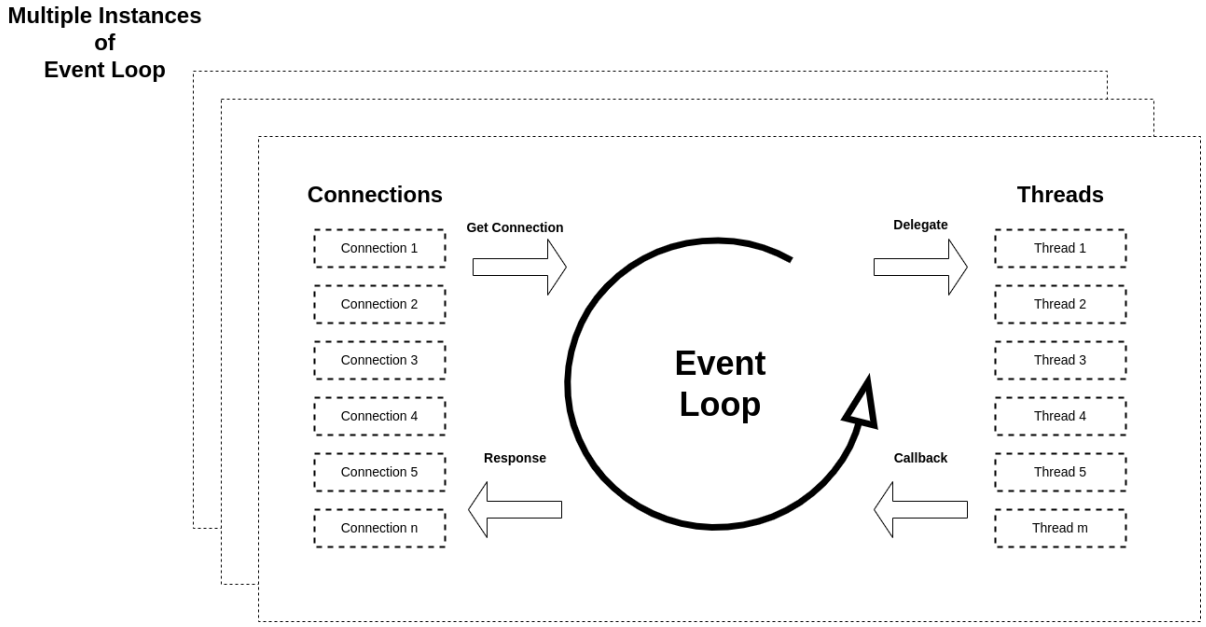

Figure 1: System architecture for the server

Concurrency via `asyncio`

Asynchronous Event Loop

- 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`).

Non-Blocking I/O

- `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.

Fair Scheduling

- 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.

Parallelism via Worker Threads

Threaded Event Loops

- 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.

Load Balancing

- 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.

Thread Safety

- 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.

Performance Benefits

- **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.