Design Document for BlinkDB Network Infrastructure (Part B)

Overview

BlinkDB's network infrastructure extends the storage engine from Part 1 to create a full-fledged database server that communicates using the Redis wire protocol (RESP-2). The implementation includes a TCP server that handles multiple client connections asynchronously using epoll for I/O multiplexing, and a load balancer that distributes client connections between multiple server instances.

System Architecture

The Part 2 implementation consists of two main components:

1. BlinkServer: A TCP server that listens for client connections, processes Redis commands, and interacts with the BlinkDB storage engine
2. LoadBalancer: A separate component that distributes client connections between multiple BlinkServer instances using a round-robin algorithm

BlinkServer Architecture

The BlinkServer follows a layered architecture:

1. Network Layer: Handles TCP connections and I/O multiplexing using epoll
2. Protocol Layer: Implements the RESP-2 protocol for communication with Redis clients
3. Command Processing Layer: Processes commands and interacts with the storage engine
4. Storage Layer: The BlinkDB storage engine from Part 1

LoadBalancer Architecture

The LoadBalancer uses a simple architecture:

1. Connection Acceptance Layer: Accepts client connections on a configurable port
2. Backend Selection Layer: Selects a backend server using round-robin algorithm

3. Data Forwarding Layer: Forwards data between clients and backend servers

Implementation Details

BlinkServer Implementation

Network Layer

The network layer uses epoll for efficient I/O multiplexing, allowing the server to handle thousands of concurrent connections with minimal overhead:

```
int epoll_fd = epoll_create1(0);
// Add server socket to epoll
```

The main event loop processes events from epoll, handling new connections and client data efficiently.

Protocol Layer

The protocol layer implements the RESP-2 (Redis Serialization Protocol) for communication with Redis clients:

1. Decoding: Parses incoming RESP-2 commands into a vector of strings
2. Encoding: Formats responses according to the RESP-2 protocol with methods like:

```
std::string encodeSimpleString(const std::string& msg);
std::string encodeBulkString(const std::string& msg);
```

Command Processing Layer

The command processing layer routes commands to the appropriate handlers:

```
std::string cmd = command[0];
std::transform(cmd.begin(), cmd.end(), cmd.begin(), ::toupper);
```

LoadBalancer Implementation

The LoadBalancer distributes client connections between multiple backend servers using a round-robin algorithm:

```
// Round-robin selection of backend server
if (current_server == 0) {
    server_ip = server1_ip;
    // ...
}
```

Design Decisions

1. I/O Multiplexing with epoll

The BlinkServer uses epoll for I/O multiplexing instead of creating a thread per connection for scalability, resource efficiency, and performance.

2. RESP-2 Protocol Implementation

The implementation fully supports the RESP-2 protocol for compatibility with Redis clients and benchmarking tools.

3. Process-per-Connection in LoadBalancer

The LoadBalancer creates a new process for each client connection using fork() for isolation, simplicity, and resource management.

4. Round-Robin Load Balancing

The LoadBalancer uses a simple round-robin algorithm for distributing connections for simplicity, fairness, and statelessness.

Performance Considerations

1. Connection Handling

The BlinkServer is designed to handle a large number of concurrent connections efficiently using epoll in edge-triggered mode.

2. Protocol Parsing

The RESP-2 protocol implementation is optimized for efficient parsing and generation, avoiding unnecessary memory allocations.

3. Command Processing

The command processing layer minimizes overhead when interacting with the storage engine.

4. Load Balancing

The LoadBalancer efficiently distributes client connections using a simple round-robin algorithm.

Running Methods for Testing

Part A: Storage Engine

To Compile

1. *Removes compiled files (benchmark executable and flush_data.txt)*

   ```
   make clean
   ```

2. *Compiles both benchmark and REPL binaries*

   ```
   make
   ```

To Run

3. *For benchmark testing, executes the benchmark tests to measure performance*

   ```
   make run_benchmark
   ```

4. *For interactive REPL interface, runs the interactive REPL interface for manual testing*

   ```
   make run_repl
   ```

Part B: Network Infrastructure

To Compile BlinkServer

5. *Removes compiled files*

   ```
   make clean
   ```

6. *Compiles the BlinkServer with BlinkDB storage engine*

   ```
   make
   ```

To Run BlinkServer

7. *Starts the BlinkServer on port 9001*

   ```
   ./blink_server
   ```

To Run benchmark from Client side

8. *results in a `result` directory, 3 × 3 files, named as result_{concurrent_requests}_{parallel_connections}.txt, for example result_1000000_100.txt for the case with 1,000,000 concurrent requests, 100 parallel connections*

   ```
   make benchmark
   ```

To Compile LoadBalancer

9. *Compiles the LoadBalancer*

   ```
   make load_balancer
   ```

To Run LoadBalancer

10. `./load_balancer <lb_port> <server1_ip> <server1_port> <server2_ip> <server2_port>`

For example, to run the LoadBalancer on port 9000 with two backend servers on localhost:

```
./load_balancer 9000 10.145.185.144 9001 192.168.1.100 9001
```

To Run redis-benchmark with the LoadBalancer from client side

Executes the benchmark tests to measure performance

11. `redis-benchmark -h [load_balancer_ip] -p 9000 -c 1000 -n 10000`

For example, to run the redis-benchmark with the LoadBalancer on port 9000 with two backend servers on localhost:

```
redis-benchmark -h 10.145.185.144 -p 9000 -c 1000 -n 10000
```

Purpose of Each Command:

1. `make clean`:
    - Removes the compiled executables and any persistence files
    - Ensures a clean state before rebuilding the application
2. `make`:
    - Compiles the source code using g++ with C++17 standard and pthread library
    - Creates the necessary executables for running the system
3. `make run_benchmark`:
    - Runs the benchmark executable to test performance of the storage engine
    - Executes three different workload scenarios (read-heavy, write-heavy, mixed)
4. `make run_repl`:
    - Runs the interactive REPL interface for the storage engine
    - Allows manual testing with commands like SET, GET, and DEL
5. `./blink_server`:
    - Starts the BlinkDB server that listens on port 9001
    - Handles client connections using epoll and processes Redis commands
6. `./load_balancer <parameters>`:
    - Starts the load balancer with specified configuration
    - Distributes client connections between multiple backend servers

Conclusion

The BlinkDB network infrastructure extends the storage engine from Part 1 to create a full-fledged database server that communicates using the Redis wire protocol. The implementation includes a TCP server that handles multiple client connections asynchronously using epoll for I/O multiplexing, and a load balancer that distributes client connections between multiple server instances.

The design prioritizes scalability, performance, and compatibility with Redis clients and tools. The use of epoll for I/O multiplexing allows the server to handle thousands of concurrent connections efficiently, while the RESP-2 protocol implementation ensures compatibility with Redis clients and benchmarking tools.

The LoadBalancer provides a simple but effective way to distribute client connections between multiple backend servers, allowing for horizontal scaling of the database

system. The process-per-connection model ensures isolation between connections, preventing a single connection from affecting others.