**ISTANBUL TECHNICAL UNIVERSITY**
**Faculty of Computer Science and Informatics**

**AI-ASSISTED LOG ANALYSIS AND VARIOUS OPERATIONS ON THE REDIS DATABASE**

# INTERNSHIP PROGRAM REPORT

**MUSTAFA CAN ÇALIŞKAN**
**150200097**

**SUMMER / 2024**

**Istanbul Technical University**

**Faculty of Computer Science and Informatics**

# INTERNSHIP REPORT

| | |
|---|---|
| Academic Year: | 2023/2024 |
| Internship Term: | ☒ Summer ☐ Spring ☐ Fall |

## Student Information

| | |
|---|---|
| Name Surname: | MUSTAFA CAN ÇALIŞKAN |
| Student ID: | 150200097 |
| Department: | Computer Engineering |
| Program: | 100% English |
| E-Mail: | caliskanmu20@itu.edu.tr |
| Mobile Phone: | +90 (533) 192 8068 |
| Pursuing a Double Major? | ☐ Yes (Faculty/Department of DM: ______________) ☒ No |
| In the Graduation Term? | ☐ Yes ☒ No |
| Taking a class at Summer School? | ☐ Yes (Number of Courses:_____) ☒ No |

## Institution Information

| | |
|---|---|
| Company Name: | HAVELSAN A.Ş. |
| Department: | Command and Control & Defense Technologies |
| Web Address: | https://www.havelsan.com/en |
| Postal Address: | Kaynarca Mah Liman Cad. No:2/57 P.K.: 11 34890 Pendik / İstanbul |

## Authorized Person Information

| | |
|---|---|
| Department: | Command and Control & Defense Technologies |
| Title: | Software Engineering Manager |
| Name Surname: | İbrahim Onuralp Yiğit |
| Corporate E-Mail: | staj@havelsan.com.tr |
| Corporate Phone: | +90 (216) 677 22 50 |

## Internship Work Information

| | |
|---|---|
| Internship Location: | ☒ Turkey<br>☐ Abroad |
| Internship Start Date: | 05.08.2024 |
| Internship End Date: | 09.09.2024 |
| Number of Days Worked: | 20 |
| During your internship, did you have insurance? | ☒ Yes, I was insured by İTÜ.<br>☐ Yes, I was insured by institution.<br>☐ No, I did my internship abroad.<br>☐ No. |

## Table of Contents

# 1 INFORMATION ABOUT THE INSTITUTION

HAVELSAN, established in 1982 as an affiliate of the Turkish Armed Forces Foundation, is one of Turkey's leading defense and software companies [1]. With its headquarters in Ankara and various departments focused on defense technologies, it stands out particularly through the Naval Combat Management Technology Center specializing in developing command and control systems for naval operations. This center engages in designing complex software solutions, system integration, real-time data processing, and developing training simulation environments. HAVELSAN offers unique products and solutions in sectors including command control and defense, simulation and training, information and communication technologies, robotics, and cybersecurity. It has successfully executed significant national and international projects, such as the National Judiciary Informatics System (UYAP) and the MİLGEM project [1]. Its mission is to lead digital transformation by providing innovative, high-tech products and solutions.

# 2 INTRODUCTION

The first two days of my internship were dedicated to an introduction to HAVELSAN, as well as training on occupational health, information security, and orientation. Following the orientation, I conducted a literature review on AI-assisted log analysis and delivered a presentation. Subsequently, I developed modules that could be utilized with the Redis for database integration that the team was working on. Additionally, I carried out various simulations with different configuration settings to observe how system failures in the Redis could be handled.

A significant portion of my time was spent researching libraries I was unfamiliar with and reading documentation related to Docker and Redis.

# 3 DESCRIPTION AND ANALYSIS OF THE INTERNSHIP PROJECT

## 3.1 Research on Log Analysis

Initially, the problem encountered by the team I worked with was the consolidation of error logs from various distributed services into a common location. This consolidation aimed to facilitate the analysis of these logs and to proactively prevent potential issues that could arise in the future, thereby informing operators about possible errors they might encounter while using the system. The intern team was tasked with researching AI-assisted methods that could be applied to address this issue. I focused my research on studies related to Root Cause Analysis (RCA).

### 3.1.1 Root Cause Analysis (RCA)

Root Cause Analysis (RCA) is a systematic process aimed at identifying the fundamental causes of a problem and eliminating those causes to prevent recurrence or the emergence of similar issues [2]. By focusing on the underlying factors rather than just addressing the superficial symptoms, RCA encourages a thorough investigation into an incident or problem, enabling organizations to implement effective solutions that not only resolve the immediate issue but also enhance overall performance and reliability in the long run.

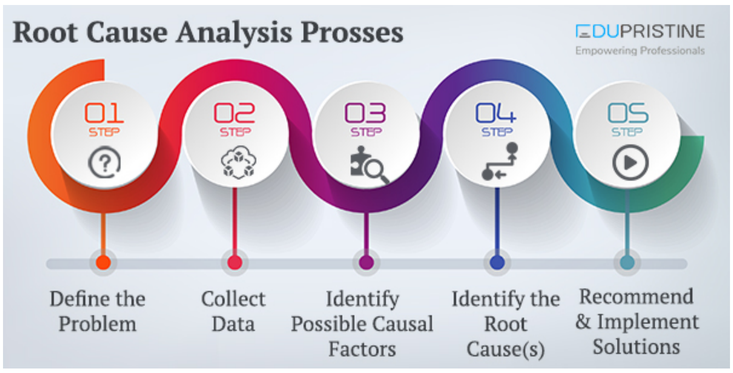
Root Cause Analysis Prosses
EDUPRISTINE
Empowering Professionals
01 STEP
02 STEP
03 STEP
04 STEP
05 STEP
Define the Problem
Collect Data
Identify Possible Causal Factors
Identify the Root Cause(s)
Recommend & Implement Solutions

Figure 1: RCA Processes [2]

### 3.1.2 AI-Powered RCA

Through the AI-powered RCA conducted on event logs, the first service where the error began to occur can be identified, allowing the operator to be alerted to a potential error related to this service (including the hardware it operates on), enabling the repair of that hardware.

### 3.1.3 Literature Review and Presentation

As a result of the literature review I conducted, I found a model called LogRCA [3] that includes a transformer-based approach for addressing this problem, and I delivered a presentation explaining the features of this model.

## 3.2 Work on Redis

After the literature review on log analysis, I worked on various tasks related to Redis integration, which is another project the team is involved in. To facilitate a better understanding of the project, I will provide a brief overview of Redis and its various versions, including Sentinel and Cluster.

### 3.2.1 What is Redis?

Redis is an advanced key-value store known for its high performance, scalability, and versatility, making it a preferred choice in distributed systems [4]. Its in-memory data structure allows for rapid data access, which is crucial for applications requiring real-time analytics and low-latency responses. Redis supports various data types, including strings, hashes, lists, and sets, enabling developers to model complex data structures efficiently. Moreover, its built-in replication, persistence options, and support for clustering enhance data availability and fault tolerance in distributed environments. As a result, Redis plays a pivotal role in ensuring robust data management and seamless operations across distributed architectures.

#### Redis Sentinel and Cluster

Redis Sentinel is a system designed to provide high availability for Redis databases. It monitors Redis master and replica instances, automatically handling failover in case the master becomes unreachable. Sentinel provides notification services to inform clients of changes in the Redis environment, such as failovers or the addition of new nodes. This ensures that applications can continue operating smoothly without manual intervention [5]. Additionally, Sentinel supports configuration management, enabling it to dynamically update the configuration of clients connecting to the Redis cluster. The working principle of Redis Sentinel can be found in Figure 2.

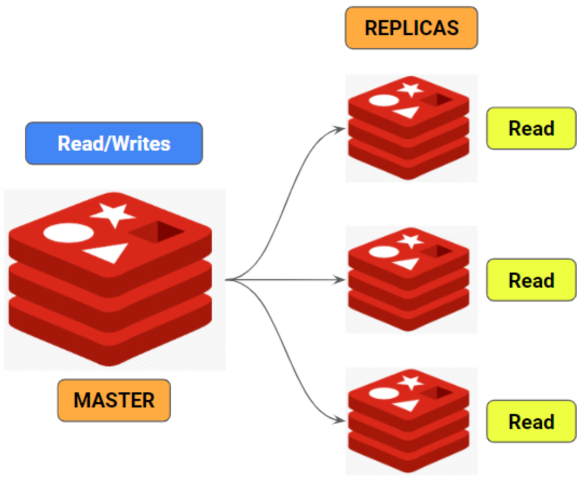

Figure 2: Redis Sentinel [5]

Redis Cluster is a distributed implementation of Redis that allows data to be automatically sharded across multiple nodes. Each node in a Redis Cluster is responsible for a subset of the keyspace, which enhances performance and scalability. Redis Cluster enables horizontal scaling, allowing users to add more nodes to accommodate growing data and traffic needs [6]. It also provides fault tolerance through data replication, where each shard can have one or more replicas. Redis Cluster ensures that the system remains operational even if some nodes fail, allowing applications to continue accessing data with minimal disruption. The working principle of the Redis cluster can be seen in Figure 3.

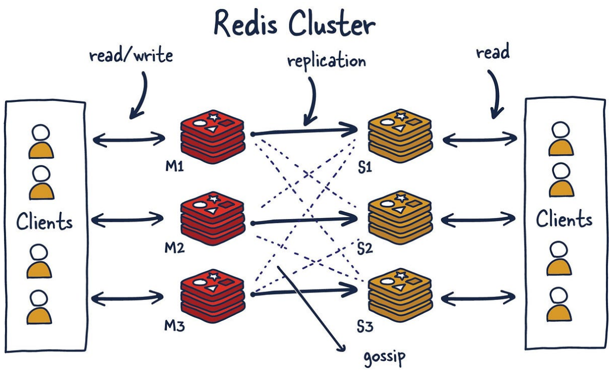

Figure 3: Redis Cluster [6]

### 3.2.2 Test Environment Setup

Throughout the project, I conducted test operations on Redis servers set up in Docker containers. I utilized the official Redis Docker image [7] and employed different Docker Compose files for Redis Sentinel and Redis Cluster. However, for demonstration purposes, I will only provide the Docker Compose file for Redis Sentinel.

#### Docker Compose

Docker Compose file similar to the one I used can be found below.

```
services:
  redis-master:
    image: redis:latest
    command: ["redis-server", "--appendonly", "yes"]
    ports:
      - "6379:6379"
```

The first part defines the Redis master service. It uses the official Redis image and runs the Redis server in append-only mode, which ensures persistence by logging all write operations to a file. The default Redis port (6379) is exposed and mapped to the same port on the host machine.

```
  redis-slave:
    image: redis:latest
    command: "redis-server --slaveof redis-master 6379"
    depends_on:
      - redis-master
    ports:
      - "6380:6379"
```

The second section defines the Redis slave service, which is configured as a replica of the master server by using the `--slaveof` option. This service depends on the Redis master to be up and running. The internal Redis port (6379) is mapped to port 6380 on the host machine.

```
  redis-sentinel:
    image: redis:latest
    command: "redis-sentinel /etc/redis/sentinel.conf"
    depends_on:
      - redis-master
      - redis-slave
    ports:
      - "26379:26379"
```

The final part defines the Redis Sentinel service, which is responsible for monitoring both the master and the slave for failover management. It uses the official Redis image and runs the Sentinel with a configuration file. This service depends on both the Redis master and slave services. It exposes the Sentinel monitoring port (26379) to the host machine.

## Makefile

Additionally, a Makefile was utilized to automate the process of building and managing the Docker environment. Below is an example of the Makefile similar to the one I used.

```
CXX = g++
CXXFLAGS = -std=c++17 -Wall -Werror -I/usr/local/include -O2
LDFLAGS = -L/usr/local/lib -lredis++ -lhiredis -pthread
TARGET = simulation
SRCS = simulation.cpp
```

The first section defines the C++ compiler (g++), compiler flags (enabling C++17, warnings, optimizations, etc.), and linking flags for Redis++ and Hiredis libraries with pthread support. The target executable is named `simulation`, and the source file is `simulation.cpp`.

```
all: up $(TARGET) run
$(TARGET):
  $(CXX) $(CXXFLAGS) $(SRCS) -o $(TARGET) $(LDFLAGS)
```

The default `all` target builds the Docker environment, compiles the C++ source, and runs the program. The `$(TARGET)` rule compiles the source into the executable using the defined flags and libraries.

### 3.2.3 SQL to Redis Conversion

At a certain stage of the project, the need arose for a program to facilitate the integration of automatically generated SQL commands (such as table creation and updates) for a MySQL database into Redis Cluster (a NoSQL database). Initially, I attempted to develop a module that could meet this requirement from within Redis itself (as a Redis module). However, due to various issues associated with this approach, I implemented a Python code that operates outside of the database.

#### Redis Module API Approach

The Redis Module API allows developers to extend Redis functionality by creating custom modules that introduce new data types and commands [8]. By leveraging this API, developers can build specialized features tailored to their applications, enabling enhanced data processing capabilities. Modules can be loaded into Redis at runtime, providing the flexibility to integrate various functionalities without modifying the core Redis server. Additionally, the API supports command registration, keyspace notifications, and custom memory management, making it a powerful tool for creating optimized data handling solutions within the Redis ecosystem.

A code similar to the one I implemented for converting SQL commands to Redis commands using the Redis module API is provided below.

```
int InsertCommand_RedisCommand(RedisModuleCtx *ctx,
RedisModuleString **argv, int argc) {
    if (argc != 2) {
        return RedisModule_WrongArity(ctx);
    }

    size_t sql_len;
    const char *sql = RedisModule_StringPtrLen(argv[1], &sql_len);
```

The `InsertCommand_RedisCommand` function starts by validating the number of arguments. It extracts the SQL string passed as the second argument, using Redis's API to get its length and pointer.

```
    char *trimmed_sql = trim_quotes(sql);
    if (!trimmed_sql) {
        return RedisModule_ReplyWithError(ctx, "ERR memory allocation error");
    }

    char table_name[64];
    char columns[256];
    char values[256];
```

The SQL string is then trimmed of any quotes using a helper function, `trim_quotes`. It proceeds to declare arrays to store the table name, columns, and values extracted from the SQL query.

```
    if (sscanf(trimmed_sql, "INSERT INTO %63s (%255[^)]) VALUES
    (%255[^)])", table_name, columns, values) != 3) {
        free(trimmed_sql);
        return RedisModule_ReplyWithError(ctx, "ERR invalid SQL command");
    }
```

The `sscanf` function is used to parse the SQL query, extracting the table name, columns, and values. If parsing fails, an error response is sent back to the client.

```
char *columns_copy = strdup(columns);
char *values_copy = strdup(values);
if (!columns_copy || !values_copy) {
    free(columns_copy);
    free(values_copy);
    free(trimmed_sql);
    return RedisModule_ReplyWithError(ctx, "ERR memory allocation error");
}
```

Copies of the parsed column and value strings are created using `strdup` to allow manipulation. If memory allocation fails at any point, an error is returned.

```
char *column_ptr = columns_copy;
char *value_ptr = values_copy;

char *column_token = next_token(&column_ptr, ", ");
char *value_token = next_token(&value_ptr, ", ");
```

The code sets up pointers to iterate through the columns and values. The `next_token` function is used to tokenize the column and value strings based on the delimiter (a comma and space).

```
RedisModuleString *key_name =
RedisModule_CreateStringPrintf(ctx, "%s:%s", table_name, value_token);
```

A Redis key is created by combining the table name and the first value (presumably the unique identifier, like a user ID).

```
RedisModuleKey *key = RedisModule_OpenKey(ctx, key_name, REDISMODULE_WRITE);
if (RedisModule_KeyType(key) != REDISMODULE_KEYTYPE_EMPTY &&
    RedisModule_KeyType(key) != REDISMODULE_KEYTYPE_HASH) {
    RedisModule_CloseKey(key);
    return RedisModule_ReplyWithError(ctx, REDISMODULE_ERRORMSG_WRONGTYPE);
}
```

The key is opened for writing. The code checks if the key already exists, ensuring it's either empty or a hash type. If not, an error is returned to indicate a type mismatch.

```
int result = -1;
while (column_token != NULL && value_token != NULL) {
    RedisModuleString *field = RedisModule_CreateString(ctx, column_token,
    strlen(column_token));
    RedisModuleString *value = RedisModule_CreateString(ctx, value_token,
    strlen(value_token));
    result = RedisModule_HashSet(key,
    REDISMODULE_HASH_NONE, field, value, NULL);
    RedisModule_FreeString(ctx, field);
    RedisModule_FreeString(ctx, value);
```

```
        column_token = next_token(&column_ptr, ", ");
        value_token = next_token(&value_ptr, ", ");
    }
```

The code enters a loop, creating Redis strings for each column and value. These pairs are inserted into the Redis hash using `RedisModule_HashSet`. Memory is freed after each insertion, and the loop continues until all tokens are processed.

```
    free(columns_copy);
    free(values_copy);
    free(trimmed_sql);

    RedisModule_CloseKey(key);
    return RedisModule_ReplyWithSimpleString(ctx, "Successfully inserted.");
}
```

After the loop, the allocated memory for column, value strings, and the SQL string is freed. The key is closed, and a success message is returned to the client.

```
int RedisModule_OnLoad(RedisModuleCtx *ctx, RedisModuleString
**argv, int argc) {
    if (RedisModule_Init(ctx, "sqlparser", 1,
    REDISMODULE_APIVER_1) == REDISMODULE_ERR) {
        return REDISMODULE_ERR;
    }

    if (RedisModule_CreateCommand(ctx, "sqlparser.insert",
    InsertCommand_RedisCommand, "admin", 0, 0, 0) == REDISMODULE_ERR) {
        return REDISMODULE_ERR;
    }

    return REDISMODULE_OK;
}
```

The final section implements the module's entry point, `RedisModule_OnLoad`. It initializes the module with the name `sqlparser` and registers the `sqlparser.insert` command, which invokes the `InsertCommand_RedisCommand` function. If any initialization step fails, an error is returned.

The example input and output of the code are provided below. The commands are running on the Redis Cluster server inside the container.

```
127.0.0.1:6379> sqlparser.insert
"INSERT INTO users (id, username, email, password)
VALUES (1, exampleUser, user@example.com,
securepassword123);"
Successfully inserted.
127.0.0.1:6379> hgetall users:1
1) "id"
2) "1"
3) "username"
4) "exampleUser"
5) "email"
6) "user@example.com"
```

```
7) "password"
8) "securepassword123"
127.0.0.1:6379>
```

I abandoned this approach due to the lack of documentation on memory management for the Redis module API and the incompatibility of the module API with the automatic data balancing feature of Redis Cluster. Furthermore, even if this approach were implemented, it would have been inefficient because it required entering commands one by one. Adding a batch command processing feature also created additional problems in terms of memory management.

## Python Approach

I wrote a Python script to solve the current problem, where SQL commands are bulk imported from an external SQL file. The commands are parsed using various regex patterns and converted into Redis commands. An example SQL file can be seen below.

```sql
CREATE TABLE customers (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100),
    phone VARCHAR(15)
);

INSERT INTO customers VALUES
(1, 'Mark Taylor', 'mark.taylor@example.com', '555-1234'),
(2, 'Sara Connor', 'sara.connor@example.com', '555-5678'),
(3, 'Liam Smith', 'liam.smith@example.com', '555-8765');
```

The Python script can be seen below.

```python
def extract_create_table_commands(sql_content):
    create_table_pattern = r'CREATE TABLE\s+(\w+)\s*\(\s*([^;]*?)\);'
    matches = re.findall(create_table_pattern, sql_content, re.DOTALL)

    return {
        table_name: [
            col.strip().split()[0] for col in columns_definition.strip().split(',')
            if col and 'FOREIGN' not in col
            and col.strip().split()[0].isidentifier()
        ]
        for table_name, columns_definition in matches
    }
```

This function extracts `CREATE TABLE` commands from the given SQL content. It uses a regular expression to find table names and their columns, returning a dictionary mapping table names to lists of their respective columns, filtering out foreign key definitions.

```python
def extract_insert_into_commands(sql_content):
    insert_into_pattern =
    r'INSERT INTO\s+(\w+)\s+VALUES\s*((?:\(\s*[^)]+\s*\)(?:,\s*)?)+);'
    return re.findall(insert_into_pattern, sql_content, re.DOTALL)
```

This function extracts `INSERT INTO` commands from the SQL content, returning a list of tuples containing table names and their corresponding values.

```
def format_insert_command(insert_table_name, fields, values):
    result = []

    for match in re.findall(r'\(\s*[^)]+\s*\)', values):
        values_list = match.strip()[1:-1].split(',')

        grouped_values = " ".join([
            f"{fields[i]} {values_list[i].strip()}"
            for i in range(len(fields))
        ])

        table_with_id = f"{insert_table_name}:{values_list[0].strip()}"
        result.append(f"HSET {table_with_id} {grouped_values}".replace("'", ""))

    return result
```

This function formats the `INSERT` values into an HSET command for Redis. It processes each match of values, creates a list of grouped values, and constructs a new command with the table name and corresponding field-value pairs.

The example input and output of the code are provided below.

```
canetizen@mcc:~/Documents/havelsan/parser$ make all
docker-compose -f ./compose.yml up -d
[+] Running 4/4
  Network cluster_redis-cluster Created
  Container can_redis_node3  Started
  Container can_redis_node1  Started
  Container can_redis_node2  Started
Creating Redis cluster...
'
Some log lines related to the Redis Cluster
connection were discarded for simplicity.
'
[OK] All 16384 slots covered.
Connection successful.
Command running: HSET customers:1 id 1 name Mark Taylor
email mark.taylor@example.com phone 555-1234
Result: 4
Command running: HSET customers:2 id 2 name Sara Connor
email sara.connor@example.com phone 555-5678
Result: 4
Command running: HSET customers:3 id 3 name Liam Smith
email liam.smith@example.com phone 555-8765
Result: 4
```

The number 4 represents the successful insertions of four attributes related to the existing key.

### 3.2.4 Failover Simulations

After the conversion of SQL queries, I took on the task of simulating the handling of failure scenarios (failover) that Redis Sentinel servers may encounter. This simulation involved various configuration settings related to Sentinel, such as the minimum number of Sentinel servers that must vote for a Redis server to be considered failed, the time a server must exceed to be deemed failed, the recovery tolerance period for a server after it has failed, and similar parameters.

I distributed the containers into different Compose files to correspond to the actual locations of the master, replica, and Sentinel servers. Subsequently, I facilitated the complete and temporary shutdown of the Compose files containing the master and replica servers at a specific point during data transfer, allowing for the observation of how the Sentinel servers responded to this situation. The simulation was implemented in C++ using the redis-plus-plus [9] library, which is the C++ library for Redis. The code can be found below.

```
class SentinelManager {
public:
    SentinelManager() {
        sentinel_opts.nodes = {{"127.0.0.1", 26379},
                               {"127.0.0.1", 26380},
                               {"127.0.0.1", 26381}};

        connection_opts.connect_timeout = std::chrono::milliseconds(200);
        connection_opts.socket_timeout = std::chrono::milliseconds(200);

        sentinel = std::make_shared<Sentinel>(sentinel_opts);
    }
```

This defines the `SentinelManager` class. In the constructor:

- Sentinel nodes are initialized to three local Redis Sentinel instances.
- Connection options for timeouts are set.
- A shared pointer to the `Sentinel` instance is created.

```
    void connect() {
        try {
            master_node = std::make_shared<Redis>
            (sentinel, "mymaster", Role::MASTER, connection_opts);
        } catch(const std::exception &e) {
            std::cerr << "Master node connection error: "
            << e.what() << std::endl;
            exit(1);
        }
    }
```

The `connect` method attempts to establish a connection to the Redis master node. If it fails, it logs an error message and exits the program.

```
    void keyset_keyget(const std::string& key, const std::string& val) {
        this->operation_retry([&]{
            auto replication_info = this->get_master_node()->info("replication");
            auto master_info = this->get_master_node()->info("server");

            std::regex pattern("tcp_port:(\\d+)");
            std::smatch match;

            std::string master_port;
            if (std::regex_search(master_info, match, pattern)) {
```

```
            master_port = match[1];
        } else {
            master_port = "Port cannot be found";
        }

        this->get_master_node()->set(key, val);
        auto result = this->get_master_node()->get(key);

        if (result.has_value()) {
            std::cout << "-----------------------" << std::endl;
            std::cout << "Master value at " << key << ": "
            << result.value() << std::endl;
            std::cout << "Master port: " << master_port << std::endl;
            std::cout << "Replication info: " << replication_info << std::endl;
            std::cout << "-----------------------" << std::endl;
        } else {
            std::cout << "Master key " << key << " not found" << std::endl;
        }
    });
}
```

The keyset_keyget method sets a key-value pair in Redis and retrieves it. It also logs the master node's replication information and TCP port. It uses a retry mechanism for operations.

```
void simulate_master_failure(SentinelManager& manager,
bool permanent_failure, int step, int temp_start, int temp_finish) {
    if (permanent_failure) {
        if (step == temp_start) {
            manager.wait_for_slave_to_sync();
            master_disconnect_command();
            int wait_millisec_after_failover = 0;
            std::this_thread::sleep_for
            (std::chrono::milliseconds(wait_millisec_after_failover));
        }

    } else {
        int wait_millisec_after_failover = 3000;
        if (step == temp_start) {
            master_disconnect_command();
            std::this_thread::sleep_for
            (std::chrono::milliseconds(wait_millisec_after_failover));
        } else if (step == temp_finish) {
            master_reconnect_command();          std::this_thread::sleep_for
            (std::chrono::milliseconds(wait_millisec_after_failover));
        }
    }
}
```

This method simulates a master node failure. It can be either permanent or temporary. The behavior varies depending on the step in the simulation, including logging and waiting as appropriate.

```
private:
    std::shared_ptr<Redis> master_node;
    std::shared_ptr<Sentinel> sentinel;
    SentinelOptions sentinel_opts;
    ConnectionOptions connection_opts;

    // Generic retry
    template<typename Func>
    auto operation_retry(Func func, int retry_count = 30,
    int retry_delay_ms = 100) -> decltype(func()) {
        int attempts = 0;
        while (attempts < retry_count) {
            try {
                return func();
            } catch (const Error &err) {
                attempts++;
                if (attempts >= retry_count) {
                    throw std::runtime_error("Maximum retry count reached.");
                }
                std::cerr << "Attempt " << attempts
                << "/" << retry_count << " failed: "
                << err.what() << ".
                Retrying in " << retry_delay_ms << "ms..." << std::endl;
                std::this_thread::sleep_for
                (std::chrono::milliseconds(retry_delay_ms));
            }
        }
        return func();
    }
```

This section defines private member variables for the SentinelManager class, including shared pointers to the Redis master node and the Sentinel instance. The operation_retry template method allows for retrying operations upon failure.

```
    void wait_for_slave_to_sync(int max_allowed_lag_ms = 100,
                                int timeout_ms = 30000) {
        auto start_time = std::chrono::steady_clock::now();
        bool slaves_synced = false;

        while (!slaves_synced) {
            try {
                auto master_info = this->get_master_node()->info("replication");

                std::regex slave_regex("slave\\d+:.*,lag=(\\d+)");
                std::smatch match;
                std::string::const_iterator search_start(master_info.cbegin());

                int max_lag = 0;
                bool all_slaves_online = true;
                bool any_slave_found = false;
```

This method begins by initializing a timeout mechanism. It captures the start time and sets a boolean flag, ‘slaves_synced‘, to ‘false‘. A ‘while‘ loop continuously checks if the slaves are synchronized with the master node. Inside the loop, it attempts to retrieve the replication information from the master node.

The method uses a regular expression (‘std::regex‘) to identify slave nodes and their lag times. It also initializes variables to track the maximum lag, whether all slaves are online, and if any slaves were found during the search.

```
        while (std::regex_search(search_start,
                                 master_info.cend(), match, slave_regex)) {
            int lag = std::stoi(match[1]);
            max_lag = std::max(max_lag, lag);
            any_slave_found = true;

            if (lag == -1) {
                all_slaves_online = false;
            }

            search_start = match.suffix().first;
        }

        if (!any_slave_found) {
            std::cout << "No slaves found to sync. Waiting for 3 seconds..."
            << std::endl;
            std::this_thread::sleep_for(std::chrono::seconds(3));
            break;
        } else {
            if (all_slaves_online && max_lag <= max_allowed_lag_ms) {
                slaves_synced = true;
                std::cout << "All slaves synced. Max lag: "
                << max_lag << " ms" << std::endl;
            } else {
                std::cout << "Waiting for slaves to sync. Current max lag: "
                << max_lag << " ms" << std::endl;
            }
        }
```

If any slave nodes are found, their lag times are extracted and compared. If a slave’s lag is ‘-1‘, it indicates that the slave is offline. The loop checks if no slaves were found and waits for 3 seconds if true. If slaves are found and all are online with acceptable lag, it sets the ‘slaves_synced‘ flag to ‘true‘. Otherwise, it logs the current lag and continues checking until a timeout occurs.

```
    void master_disconnect_command() {
        std::system("docker stop redis-master");
    }

    void master_reconnect_command() {
        std::system("docker start redis-master");
    }
```

These two methods handle the disconnection and reconnection of the Redis master node through Docker commands.

```
void test_data_exchange(SentinelManager &manager) {
    log_info("Failover simulation started.");
    int temp_failure_started = 6000;
    int temp_failure_finished = 12000;
    bool permanent_failure = false;
    for (int i = 1; i < 18000; i++) {
        try {
            manager.keyset_keyget("key" + std::to_string(i), std::to_string(i));
            manager.simulate_master_failure(manager, permanent_failure, i,
            temp_failure_started, temp_failure_finished);
        } catch (const std::exception& err) {
            log_error("Error: " + std::string(err.what()));
        }
    }

    log_info("Simulation completed.");
}
```

The `test_data_exchange` function simulates data exchanges and failover scenarios. It logs the start of the simulation, executes key set and get operations, and manages master failure simulations, handling any exceptions.

When the code is executed, it will stop after 6000 iterations. Since the master node is taken out of service, the slave node will take over as the new master. Since the error on the master node is configured as a temporary error, after a while, the old master node will resume operation and continue to function as the slave of the new master node.

Throughout the internship, the number of iterations, the location of the containers, and the configuration settings of Redis Sentinel were altered, allowing this simulation to be continuously repeated and the obtained results to be analyzed. The outputs and logs were not included in the report due to their large volume and complexity.

# 4 CONCLUSIONS

My internship at HAVELSAN has provided invaluable hands-on experience with advanced software development, particularly in database management and distributed systems. While I gained insights into AI-assisted log analysis, the majority of my learning centered on Redis integration projects. Working with Redis in various configurations enhanced my understanding of NoSQL databases, distributed architectures, and fault-tolerant system design. The challenges encountered during SQL to Redis conversion and failover simulations significantly improved my problem-solving skills.

HAVELSAN's multidisciplinary approach to technology integration, especially in database management, has broadened my perspective on modern software engineering. This experience has deepened my appreciation for robust, scalable database solutions in mission-critical applications. The practical skills gained in configuring and testing Redis deployments for high availability have been particularly valuable. I am grateful for this opportunity and eager to apply these learnings in my future endeavors in computer engineering.

# 5 REFERENCES

[1] HAVELSAN. "About Us." [Online]. Available: `https://www.havelsan.com/en/about-us`. [Accessed: Oct. 17, 2024].

[2] EduPristine. "Root Cause Analysis." [Online]. Available: `https://www.edupristine.com/blog/root-cause-analysis/`. [Accessed: Oct. 17, 2024].

[3] T. Wittkopp, P. Wiesner, O. Kao. “LogRCA: Log-based Root Cause Analysis for Distributed Services.” [Online]. Available: `https://arxiv.org/abs/2405.13599`. [Accessed: Oct. 17, 2024].

[4] IBM. “What is Redis?” [Online]. Available: `https://www.ibm.com/topics/redis`. [Accessed: Oct. 17, 2024].

[5] A. Sahay. “Redis Multi-node deployment: Replication vs. Cluster vs. Sentinels.” [Online]. Available: `https://ankitsahay.medium.com/redis-multi-node-deployment-replication-vs-cluster-vs-sentinels-8099d15dcc09`. [Accessed: Oct. 17, 2024].

[6] R. Pachauri. “What are Redis Cluster and How to Setup Redis Cluster Locally?” [Online]. Available: `https://medium.com/@rajatpachauri12345/what-are-redis-cluster-and-how-to-setup-redis-cluster-locally-69e87941d573`. [Accessed: Oct. 17, 2024].

[7] Docker Hub. “Redis Docker.” [Online]. Available: `https://hub.docker.com/_/redis`. [Accessed: Oct. 17, 2024].

[8] Redis. “Redis Module API.” [Online]. Available: `https://redis.io/docs/latest/develop/reference/modules/`. [Accessed: Oct. 17, 2024].

[9] sewenew. “redis-plus-plus: Redis client written in C++.” [Online]. Available: `https://github.com/sewenew/redis-plus-plus`. [Accessed: Oct. 17, 2024].