Einfacher Redis-Client mit RESP3
Beispiel 33

Marcel Dinhof, 5BHIF
Katalognummer: 3

April 2022

Contents

1 Einleitung

Das Ziel dieses Projektes ist die Umsetzung eines einfachen, auf dem Protokoll RESP3 basierenden, Redis-Clients. Dieser soll Funktionalitäten wie Pipelining, Publishing/Subscribing, Transaktionen, als auch verteilte Locks implementieren.

2 RESP3

RESP3 ist ein Protokoll für die Kommunikation zwischen einem Redis-Client und einem Redis Server. In der folgenden Auflistung werden grundlegende Datentypen des RESP3 Protokolls dargestellt. Aus Gründen der Lesbarkeit werden die Zeichen "\r\n" durch <CR><LF> dargestellt.

Datentyp	RESP Darstellung
Simple String	+MSG<CR><LF>
Bulk String	$MSG-SIZE<CR><LF>MSG<CR><LF>
Integer	:MSG<CR><LF>
Error	-MSG<CR><LF>
Array	*ARRAY-SIZE<CR><LF>ELEMENTS<CR><LF>
Map	%MAP-SIZE<CR><LF>KEY/VALUE-PAIRS<CR><LF>

Figure 1: Liste der in diesem Projekt benutzten Redis-Datentypen mit ihrem Aufbau

3 Funktionalitäten

3.1 Pipelining

Pipelining [5] ist eine Taktik zur Netzwerk-Optimierung, bei der Befehle nicht sofort zum Server gesendet, sondern beim Client gepuffert werden. Dieser sendet zu einem späteren Zeitpunkt alle Anweisungen gleichzeitig an den Server. Pipelining ist nicht mit Transaktionen (Sektion 3.3) zu verwechseln, da keine Garantie für die Ausführung der Befehle gewährleistet wird.

3.2 Publish und Subscribe

Die Befehle *Publish* und *Subscribe* können genutzt werden, um Nachrichten an alle Clients zu senden, die einen bestimmten Wert mittels Subscribe überwachen [6].

3.3 Transaktionen

Eine weitere Funktionalität von Redis ist die Nutzung von *Transaktionen* [7]. Transaktionen ermöglichen die gleichzeitige Ausführung von mehreren Befehlen, die als eine Einheit sequenziell abgearbeitet werden [7]. Die Kommandos für Transaktionen sind:

- *MULIT* - Startet eine Transaktion.
- *EXEC* - Führt eine Transaktion aus und beendet diese.
- *DISCARD* - Verwirft die gesammelten Befehle und beendet eine Transaktion.

3.4 Redlock

Redlocks [4] bieten die Möglichkeit, Sperren in einem Verbund an Redis-Clients anzufordern. Dabei wird einem Schlüsselelement ein zufällig generierter Wert zugewiesen. Wenn die Sperre angefordert wird, versucht der Client diesen Schlüssel mit seinem eindeutigen Wert zu setzen. Sollte dieser schon von einem anderen Client gesetzt worden sein, wartet der Client darauf, dass der Schlüssel am Server entfernt wird und er seinen eigenem Wert setzen darf. Beim Entsperren wird der Wert von Redis mit

dem lokalen Wert des Clients abgeglichen. Nur wenn diese identisch sind, darf die Sperre aufgehoben werden.

4 Struktur

4.1 Konventionen

Für Klassennamen wurde in diesem Projekt "Pascal case" und für Funktionsnamen sowie für Variablennamen "Snake case" verwendet. Verwendete Bibliotheken werden in Sektion 7 zusammengefasst und grundlegend beschrieben.

4.2 Klassendiagramm

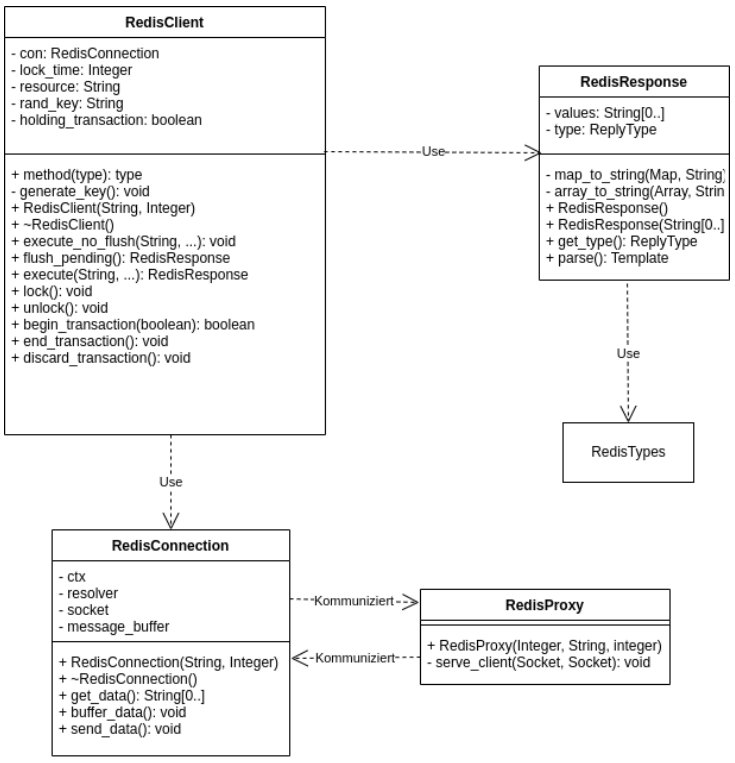

Figure 2: Klassendiagramm

4.3 RedisClient

Die Klasse RedisClient implementiert alle Funktionalitäten, die der Client zur Verfügung stellt.

4.4 RedisConnection

Die Klasse RedisConnection implementiert die Funktionen für eine Kommunikation mit einem Redis Server. Die Klasse verfügt über Methoden zum Lesen, Schreiben und Puffern der Daten.

4.5 RedisType Klassen

Für die Serialisierung und Speicherung der Rückgabewerte eines Redis-Servers wurden für jeden, in Sektion 2 beschriebenen, RESP3 Datentypen eine eigene Klasse implementiert. Dem Konstruktor dieser Klassen wird eine *deque* mit *strings* übergeben, deren Inhalt in die entsprechende Datenstruktur (z.B. SimpleString, BulkString, ...) überführt wird.

4.6 RedisResponse

Die RedisResponse Klasse implementiert einen Konstruktor, der den Typen eines empfangenen Rückgabewertes aus dem ersten Teil der Nachricht verarbeitet. Des Weiteren sind in der Klasse überladene *parse* - Methoden implementiert. Beim Aufruf einer *parse* - Methode muss ein Datentyp für die Rückgabe angegeben werden. Daraufhin wird versucht, die empfangene Nachricht in diesen Typen zu überführen, beispielsweise durch Konvertierung eines *integer* in einen *string*. Folgende Datentypen können für die Rückgabe ausgewählt werden:

- *string*
- *integer*
- *Redis::Array*
- *Redis::Map*

Sollte ein RedisType nicht in den spezifizierten Datentypen geparsed werden können, löst die Funktion eine Ausnahme aus.

4.7 RedisProxy

Implementiert einen Proxy-Server, der einerseits mittels Protobuf (beschrieben in Sektion 6.3) mit einem Redis-Client kommuniziert und dessen Nachrichten an einen Redis-Server weiterleitet, und andererseits die Nachrichten von einem Redis-Server in Protobuf überführt und an einen Client sendet.

4.8 Logger

In der Datei logger.h wird das Logging mit *spdlog* über Präprozessoranweisungen implementiert. Der Detaillierungsgrad des Loggens kann durch *SET_LOGLEVEL* angepasst werden.

5 Redis Befehle

In diesem Abschnitt werden Kommandos vorgestellt, die in diesem Client genutzt wurden um die geforderten Funktionalitäten zu implementieren. Eine vollständige Dokumentation zu allen Redis-Befehlen kann auf der offiziellen Website [2] nachgelesen werden.

SET [**key**] [**value**] Speichert einen angegebenen Wert mit einem eindeutigen Schlüssel.

SET [**key**] [**value**] **NX** NX gibt an, dass der Wert nur gesetzt wird, wenn er noch nicht existiert.

SET [**key**] [**value**] **PX** [**time**] Die Option PX kann benutzt werden, um ein Element nach einer angegebenen Zeit (in Millisekunden) zu löschen.

GET [**key**] Liefert den Wert zurück, der mit dem angegebenen Schlüssel gespeichert wurde. Falls kein Wert vorhanden ist, ist der Rückgabewert null.

MULTI Der MULTI Befehl wird genutzt, um Transaktionen zu starten.

DISCARD Transaktionen können mit DISCARD abgebrochen werden.

EXEC Transaktionen können mit EXEC ausgeführt werden.

6 Implementierungsbeispiele

6.1 Parsen in einen Redis Datentypen

6.1.1 SimpleString Klasse

Im Konstruktor der *RedisType* Klassen werden die empfangenen Nachrichten entsprechend angepasst. Da ein *SimpleString* mit einem '+' beginnt, wird dieses Zeichen entfernt.

```
class SimpleString {
private:
    std::string content{};
public:
    SimpleString(std::deque<std::string>& msg) {
        content = msg.at(0);
        msg.pop_front();
    }

    std::string& get() {
        return content;
    }
};
```

Source Code 1: SimpleString Klasse

6.1.2 Array Parsing

Sowohl ein Array als auch eine *Map* können alle, in Sektion 2 beschriebenen, Datentypen beinhalten. Aus diesem Grund wird bei jeder Einfügeoperation der Typ der empfangenen Nachricht ermittelt, und dementsprechend wird die dazugehörige Klasse erzeugt.

```
switch (determin_type(msg.at(0))) {
    case ReplyType::simple_string:
        content.push_front(std::make_shared<redis_types>(SimpleString(msg)));
```

```
            break;
        case ReplyType::bulk_string:
            content.push_front(std::make_shared<redis_types>(BulkString(msg)));
            break;
        case ReplyType::integer:
            content.push_front(std::make_shared<redis_types>(Integer(msg)));
            break;
        case ReplyType::error:
            content.push_front(std::make_shared<redis_types>(Error(msg)));
            break;
        case ReplyType::array:
            content.push_front(std::make_shared<redis_types>(Array(msg)));
            break;
        default:
            break;
        }
}
```

Source Code 2: Parsen eines Redis Arrays

6.2 Ausführung von Befehlen

Methoden zur Ausführung von Befehlen sind in der RedisClient Klasse implementiert. Es wird zwischen den beiden Methoden *execute_no_flush* und *execute* unterschieden.

6.2.1 execute no flush

Die execute_no_flush Methode wird in der execute Methode aufgerufen. Nach dem Parameteroperation können beliebig viele weitere Argumente übergeben werden. Ist die Verbindung zum Redis-Server aktiv, werden die übergebenen Werte in einen RESP3 konformen string überführt. Die daraus resultierende Zeichenkette wird an den Server gesendet.

```
template<typename ...T>
void execute_no_flush(std::string operation, T ... args) {
    if (con == nullptr) {
        LOG_ERROR("No connection!");
```

```
        return;
    }
    std::deque<std::string> cmd {
        "$" + std::to_string(operation.length()) +
        operation,
        (
            "$" + std::to_string(std::string{args}.length()) +
            std::string{args}
        ) ...
    };
    cmd.push_front("*" + std::to_string(cmd.size()));
    RESP3Request request;
    for (const auto& e : cmd) {
        request.add_argument(e);
    }
    con->bufferData(request);
}
```

Source Code 3: execute_no_flush Methode aus der RedisClient Klasse.

6.2.2 execute

Die Parameter der execute Methode werden der execute_no_flush Methode übergeben, und danach wird der Puffer der Verbindungs-Klasse geleert. Der Rückgabewert der Methode ist ein RedisResponse Objekt, dem die empfangenen Daten im Konstruktor übergeben werden.

```
template<typename ...T>
RedisResponse execute(std::string operation, T && ... args) {
    if (con == nullptr) {
        LOG_ERROR("No connection!");
        return RedisResponse{};
    }
    execute_no_flush(operation, args...);
    con->sendData();
    return RedisResponse{con->getData()};
}
```

Source Code 4: execute Methode aus der RedisClient Klasse.

6.3 Protocol buffers

Protobuf [3] ist Plattformunabhängig und kann in Kombination mit unterschiedlichen Programmiersprachen verwendet werden, um strukturierte Daten zu serialisieren. Die Struktur der Daten wird in einer *.proto* Datei festgelegt, deren Elemente sog. *Messages* sind. Folgende Struktur wurde für diesen Redis-Client implementiert.

```
syntax = "proto3";

message Message {
    repeated string argument = 1;
}

message MessageBundle {
    string mode = 1;
    repeated Message message = 2;
}
```

RESP3Responses werden zum Serialisieren der Daten verwendet, die vom Server gesendet wurden. RESP3Commands werden zum Serialisieren der Kommandos verwendet, die zum Server gesendet werden.

7 Verwendete Bibliotheken

7.1 CLI11

CLI11 [1] kann zum Verarbeiten von Kommandozeilenparametern verwendet werden und bietet unter anderem Möglichkeiten zur Implementierung von Interfaces und für die Validierung von Eingaben.

7.2 spdlog

Spdlog ist eine header-only Bibliothek zur Erstellung von Logging Ausgaben in einem Programm.

8 Verwendung des Clients

8.1 Kommandozeilenargumente

–ip Die IP-Addresse des Redis-Servers, mit dem der Client eine Verbindung aufbauen soll.

-p,–port Der Port des Redis-Servers, mit dem der Client eine Verbindung aufbauen soll.

```
./redis_client -h

A simple redis client
Usage: ./redis_client [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  --ip TEXT                   ip address to connect to
  -p,--port INT               port to connect to
```

Source Code 5: Ausgabe der Hilfestellung des Clients

8.2 Beispiele

Folgende Beispiele veranschaulichen, wie dieser Client zu benutzen ist. In den Beispielen wird davon ausgegangen, dass jeder Rückgabewert zu einer Zeichenkette konvertierbar ist.

8.2.1 SET und GET

```
Redis::RedisClient client{ip_address, destination_port};
if (!client.is_connected()) {
    //error handling
    return 1;
}
std::string output;
output = client.execute("SET", "name", "MaxMuster123").parse<std::string>();
LOG_INFO(output);
output = client.execute("GET", "name").parse<std::string>();
LOG_INFO(output);

>> +OK
>> MaxMuster123
```

8.2.2 Pipelining

```
client.execute_no_flush("SET", "name", "MaxMuster321");
client.execute_no_flush("GET", "name");
std::vector<Redis::RedisResponse> responses{client.flush_pending()};
for (Redis::RedisResponse& response : responses) {
    LOG_INFO(response.parse<std::string>() << std::endl);
}

>> +OK
>> MaxMuster123
```

8.2.3 Transaktionen

```
output = client.execute("SET", "name", "MaxMuster321").parse<std::string>();
LOG_INFO(output);
client.begin_transaction();
output = client.execute("SET", "name", "MaxMuster123").parse<std::string>();
LOG_INFO(output);
output = client.execute("GET", "name").parse<std::string>();
LOG_INFO(output);
//client.end_transaction();       -> Transaktion wird durchgeführt
client.discard_transaction(); // -> Transaktion wird abgebrochen
```

```
output = client.execute("GET", "name").parse<std::string>();
LOG_INFO(output);

>> +QUEUED
>> +QUEUED
>> +OK
>> MaxMuster321
```

8.2.4 Redlock

```
client.lock("resource_1");
output = client.execute("SET", "name", "MaxMuster123").parse<std::string>();
LOG_INFO(output);
output = client.execute("GET", "name").parse<std::string>();
LOG_INFO(output);
client.unlock("resource_1");

>> +OK
>> MaxMuster123
```

8.2.5 Subscribe

```
client.subscribe("subscribe_object");
while (true) {
    LOG_INFO("Subscriber waiting for data");

    std::vector<Redis::RedisResponse> responses{client.fetch_data()};
    for (Redis::RedisResponse& response : responses) {
        LOG_INFO(response.parse<std::string>());
    }
}

>> ...
```

8.2.6 Publish

```
std::string input{};
while (true) {
    std::cout << "Element: ";
```

```
    std::getline(std::cin, input);
    output = client.execute("PUBLISH", "subscribe_object", input).parse<std::strir
    LOG_INFO(output);
}

>> ...
```

9 Projektstruktur

```
/
├── LICENSE
├── meson_options.txt
├── meson.build
├── README.md
├── CHANGELOG.org
├── .gitignore
├── include
│   ├── logger.h
│   ├── proxy.hpp
│   ├── redis_client.hpp
│   ├── redis_connection.hpp
│   ├── redis_response.hpp
│   └── redis_types.hpp
├── src
│   ├── logger.cpp
│   ├── main.cpp
│   ├── proxy.cpp
│   └── redis.proto
├── doc
│   ├── dokumentation.tex
│   ├── references.bib
│   └── dokumentation.pdf
└── build
```

References

[1] CLI11 command line parser for c++11. https://github.com/CLIUtils/CLI11. Accessed: 2022-04-05.

[2] Commands. https://redis.io/commands/. Accessed: 2022-04-04.

[3] google.com. Protocol Buffers. https://developers.google.com/protocol-buffers. Accessed: 2022-04-05.

[4] redis.io. Distributed Locks with Redis a distributed lock pattern with redis. https://redis.io/docs/reference/patterns/distributed-locks/. Accessed: 2022-04-04.

[5] redis.io. Redis pipelining how to optimize round-trip times by batching redis commands. https://redis.io/docs/manual/pipelining/. Accessed: 2022-04-04.

[6] redis.io. Redis Pub/Sub how to use pub/sub channels in redis. https://redis.io/docs/manual/pubsub/. Accessed: 2022-04-04.

[7] redis.io. Transactions how transactions work in redis. https://redis.io/docs/manual/transactions/. Accessed: 2022-04-04.