HTBLuVA Wiener Neustadt
Höhere Lehranstalt für Informatik

NVS PROJEKT

gRPC-Tutorial

Ausgeführt im Schuljahr 2019/20 von:

gRPC - Benutzerverwaltung
Niklas KLAUSBERGER 5BHIF-12

Lehrer:

Dipl.-Ing. Dr. Günter Kolousek

Wiener Neustadt, am 3. Mai 2020

Abgabevermerk: Übernommen von:

Inhaltsverzeichnis

1 gRPC **1**
- 1.1 Einführung . . . 1
  - 1.1.1 gRPC . . . 1
  - 1.1.2 Protocol Buffers . . . 1
  - 1.1.3 spdlog . . . 2
  - 1.1.4 Meson . . . 2
- 1.2 Leitfaden . . . 2
  - 1.2.1 Installation . . . 2

2 Aufgabestellung **5**

3 Anleitung **6**
- 3.1 Ordnerstruktur . . . 6
- 3.2 CLI11.hpp . . . 6
- 3.3 Meson . . . 7
- 3.4 Protocol Buffers . . . 8
- 3.5 Server . . . 9
  - 3.5.1 header inkludieren . . . 9
  - 3.5.2 main . . . 9
  - 3.5.3 Run . . . 10
  - 3.5.4 User . . . 12
  - 3.5.5 Manager . . . 16
- 3.6 Benutzerverwaltung am Server . . . 20
  - 3.6.1 Service-Implementierung . . . 22
- 3.7 Client . . . 23
  - 3.7.1 includes . . . 23
  - 3.7.2 main . . . 24
  - 3.7.3 Run . . . 24

4 Kompilieren und Ausführen **28**

Kapitel 1

gRPC

1.1 Einführung

1.1.1 gRPC

Abbildung 1.1: grpc Logo

gRPC ist ein modernes Open-Source-Hochleistungs-RPC-Framework, das in jeder Umgebung ausgeführt werden kann. Es kann Dienste in und zwischen Rechenzentren effizient miteinander verbinden und unterstützt Plug-ins für Lastausgleich, Rückverfolgung, Zustandsprüfung und Authentifizierung. Es ist auch in der letzten Meile des verteilten Rechnens anwendbar, um Geräte, mobile Anwendungen und Browser mit Backend-Diensten zu verbinden.

gRPC ist Open Source, d. h. der Quelltext ist öffentlich zugänglich, Änderungen und Weiterentwicklungen des Codes durch Dritte sind erwünscht und erlaubt.

Den Transport von Datenströmen zwischen entfernten Computern wickelt gRPC standardmäßig mit HTTP/2 ab, für die Strukturierung und Verarbeitung der Daten sind die von Google entwickelten Protocol Buffers zuständig. Letztere werden in Form von einfachen Textdateien mit der Endung .proto gespeichert.

REST vs. gRPC

1.1.2 Protocol Buffers

Protocol Buffers sind Googles sprachneutraler, plattformneutraler, erweiterbarer Mechanismus zur Serialisierung strukturierter Daten - denken Sie an XML, aber kleiner, schneller und einfacher. Sie legen einmal fest, wie Ihre Daten strukturiert werden sollen, und können dann mit Hilfe von speziell generiertem Quellcode Ihre strukturierten Daten einfach in eine Vielzahl von Datenströmen und in verschiedenen Sprachen schreiben und lesen.

Protokollpuffer unterstützen derzeit generierten Code in Java, Python, Objective-C und C++. Mit der neuen proto3-Sprachversion können Sie auch mit Dart, Go, Ruby und C# arbeiten, wobei weitere Sprachen folgen werden.

1.1.3 spdlog

spdlog ist eine sehr schnelle, header-only-C++-Bibiliothek für das Durchführen von Logging.

1.1.4 Meson

Meson ist ein Open-Source-Bausystem, das sowohl extrem schnell als auch, was noch wichtiger ist, so benutzerfreundlich wie möglich sein soll.
Der Hauptentwurfspunkt von Meson ist, dass jeder Moment, den ein Entwickler mit dem Schreiben oder Debuggen von Build-Definitionen verbringt, eine verlorene Sekunde ist. So wird jede Sekunde damit verbracht, darauf zu warten, dass das Build-System tatsächlich mit der Kompilierung von Code beginnt.

1.2 Leitfaden

1.2.1 Installation

Im Dokumentations-Bereich der Installation wird ausschließlich auf die Installationen auf Ubuntu/Debian-Systemen (getestet in Ubuntu 19.04) eingegangen.

Vorraussetzungen

Um protobuf aus den Quellen zu bauen, werden die folgenden Werkzeuge benötigt:

- autoconf
- automake
- libtool
- make
- g++
- unzip

Zusätzlich zu den oben genannten werden für die Installation von gRPC noch folgende Pakete benötigt:

- build-essential
- pkg-config

Als erstes müssen die erforderlichen Voraussetzungen mittels Linux-Terminal installiert werden wie folgt (davor starten wir am Besten mit dem Aktualisieren des Paketquellen):

```
1  $ [sudo] apt update
2  $ [sudo] apt-get install build-essential autoconf    \
3                           libtool pkg-config automake \
4                           curl make g++ unzip
```

gRPC

Für das automatisierte Erzeugen des Ausführungsprogrammes verwenden wir CMake, also installieren wir dieses:

```
[sudo] apt-get install cmake
```

Vor dem schlussendlichen Build, muss zuerst das Repository geclont werden, das machen wir mit git und dafür muss zuerst git installiert werden: Die aktuelle Release-Version findet man auf https://github.com/grpc/grpc/releases z.B. v1.28.1.

```
$ [sudo] apt install git
$ [sudo] git clone -b REALEASE_TAG --recursive https://github.com/grpc/
```

Vor dem Kompilieren müssen Sie das gRPC github-Repository klonen und Untermodule herunterladen, die den Quellcode für die Abhängigkeiten von gRPC enthalten (dies geschieht mit dem Befehl submodule oder dem Flag –recursive).

```
$ cd grpc
$ git submodule update --init
```

Zum kompilieren mit CMake führen wir folgende Befehle im grpc-Ordner aus:

```
$ mkdir -p cmake/build
$ cd cmake/build
$ cmake ../..
$ make
```

Die Installation nach dem Kompilieren durch CMake erfolgt folgendermaßen:
(Der Installationsort im Dateisystem wird durch die Variable $CMAKE\_INSTALL\_PREFIX$ gehandhabt)

```
$ cmake ../.. -DgRPC_INSTALL=ON                \
              -DCMAKE_BUILD_TYPE=Release       \
              -DgRPC_ABSL_PROVIDER=package     \
              -DgRPC_CARES_PROVIDER=package    \
              -DgRPC_PROTOBUF_PROVIDER=package \
              -DgRPC_SSL_PROVIDER=package      \
              -DgRPC_ZLIB_PROVIDER=package
$ make
$ make install
```

Protocol Buffers

Den Quellcode erhalten wir durch das Klonen des Git-Repositories. Auch hier wichtig: das Klonen der Untermodule nicht vergessen und das Konfigurationsskript generieren, wie folgt:

```
$ git clone https://github.com/protocolbuffers/protobuf.git
$ cd protobuf
$ git submodule update --init --recursive
$ ./autogen.sh
```

Um die C++-Protokollpuffer-Runtime und den Protokollpuffer-Compiler (protoc) zu erstellen und zu installieren, führen Sie Folgendes aus:

```
$ ./configure
$ make
$ make check
$ sudo make install
$ sudo ldconfig
```

spdlog

Zum Herunterladen klonen wir das Reporitories von spdlog wie folgt:

```
$ git clone https://github.com/gabime/spdlog.git
```

Meson

Meson inklusive des Build-Systems ninja wird wie folgt installiert:

```
$ sudo apt-get install python3 python3-pip python3-setuptools \
                       python3-wheel ninja-build
```

CLI11.hpp

Zum Installieren von CLI11.hpp downloaden wir die neueste Version von https://github.com/CLIUtils/CLI11/releases.

Kapitel 2

Aufgabestellung

In folgendem Kapitel wird beschrieben wie mittels gRPC ein Server implementiert wird, der eine kleine Benutzerverwaltung implementiert. Außerdem wird eine Auswahl zwischen ungesicherter und gesicherter Kommunikation mittels SSL/TLS implementiert.

Des weiteren wird auch ein Client implementiert, der die gesetzten Benutzerrechte verwenden kann.

Kapitel 3

Anleitung

3.1 Ordnerstruktur

Zuerst erstellen wir folgende Ordnerstruktur mit den benötigten Dateien für Meson, protobuf und die benötigten C++-Dateien.

```
build
src
    server
        main.cpp
    client
        main.cpp
    um.proto
include
    server
    client
    CLI11.hpp
meson.build
meson-options.txt
openssl
```

In den nachfolgenden Kapiteln wird auf die Struktur eingegangen und in weiterer Folge werden folgende zusätzliche Dateien erzeugt:

- src/user.cpp
- include/user.h
- src/manager.cpp
- include/manager.h

Außerdem werden in weiterer Folge noch die benötigten Lizenzen für SSL im Ordner *openssl* erstellt.

3.2 CLI11.hpp

Wie oben dargestellt wird die heruntergeladene Datei $CLI11.hpp$ in den include-Ordner kopiert.

3.3 Meson

Zuerste wird auf die Dateien meson.build und meson-options.txt eingegangen
Die erstellte Datei meson.build erweitern wir um das Einbinden der benötigten Pakete und um die Zeile zum Erstellen des Meson-Projektes, das c++ verwendet:

```
project('grpc-project', 'cpp',
        default_options : ['cpp_std=c++11'])

protoc = find_program('protoc', required : true)
grpc_cpp_plugin = find_program('grpc_cpp_plugin', required : true)
protobuf_dep = dependency('protobuf', required : true)
grpc_dep = dependency('grpc', required : true)
grpcpp_dep = dependency('grpc++', required : true)
threads = dependency('threads')
```

Danach wird dafür gesorgt, dass aus der Datei um.proto Dateien gemacht werden, die von Server und Client verwendet werden, das passiert wie folgt:

```
gen = generator(protoc, \
  output    : ['@BASENAME@.pb.cc', '@BASENAME@.pb.h', '@BASENAME@.grpc.pb.cc', '
    @BASENAME@.grpc.pb.h'], \
  arguments : ['--proto_path=@CURRENT_SOURCE_DIR@/src/', '--plugin=protoc-gen-grpc=' +
     grpc_cpp_plugin.path(),
              '--grpc_out=@BUILD_DIR@', '--cpp_out=@BUILD_DIR@', '
    @CURRENT_SOURCE_DIR@/src/@PLAINNAME@'])

generated = gen.process('src/um.proto')
```

Die erzeugte Datei auf der Grundlage von um.proto, die danach in den C++-Dateien inkludiert wird heißt *um.grpc.pb.h*, außerdem werden die C++-Dateien *um.grpc.cc* und *um.pb.cc* erzeugt und eben diese Dateien speichern wir in der Variable *generated* ab.

Um ein ausführbares Programm für beispielsweise unseren Client mittels Meson zu erzeugen wird unsere meson.build mit folgendem erweitert:

```
inc_dir = include_directories('include')
inc_client = include_directories('include/client')

src_client = ['src/client/main.cpp',generated]

executable('client',
           sources : src_client,
           include_directories : [inc_dir, inc_client],
           dependencies : [threads, protobuf_dep, grpc_dep,
           grpcpp_dep]
          )
```

Unser include-Ordner enthält eben die header-Dateien, die wir in den C++-Dateien verwenden wollen.
Als Source-Dateien zählen unsere C++-Dateien und die generierten C++-Dateien, die vorher in der Variable *generated* abgespeichert worden sind.
Aus diesen include- und source-Dateien und den vorher eingebundenen Paketen für gRPC und protobuf wird das ausführbare Programm mit dem Namen *client* erzeugt.

Eben das gleiche machen wir in weiterer Folge für das Server-Programm.
Aufpassen, dass die richtigen Pfäde verwendet werden!

Da später mittels spdlog geloggt werden soll, müssen wir jetzt die Datei $meson_options.txt$ mit folgendem Inhalt befüllen (darum $get_option('spdlog_include_dir')$):

```
option('spdlog_include_dir', type : 'string', value : '/home/niklas/Desktop/Includes/
    spdlog/include/', description : 'the include dir of spdlog')
```

Der Pfad muss natürlich individuell angepasst werden.
Und in der Datei *meson.build* muss die Variable inc_dir jetzt folgendermaßen gesetzt werden:

```
inc_dir = include_directories(['include', get_option('spdlog_include_dir')])
```

3.4 Protocol Buffers

Jetzt wird das Erstellen der *.proto*-Datei beschrieben. In unserem Fall *um.proto*, was für User Management steht. Die Datei ist wie folgt aufgebaut:

```
syntax = "proto3";

option java_package = "ex.grpc";

package um;

// Defines the service
service UM {
    // Function invoked to send the request
    rpc Login (LoginRequest) returns (LoginReply) {}
}

// The request message containing requested answer
message LoginRequest {
    string user = 1;
    string pw = 2;
}

// The response message containing response
message LoginReply {
    string result = 1;
}
```

Die ersten Zeile legt die verwendete proto-Version fest, in unserem Fall *proto*3. Die nächste Zeile ist zusätzliche Konfiguration.

Danach legen wir den Namen des Paketes fest, der später eingebunden wird, also *um*.

Das Service bildet die Verbindung zwischen proto und C++, die wir später verwenden werden.

Da eine Benutzerverwaltung implementiert wird, muss sich der Client anmelden können. Der Funktionsname unserer ersten Funktion ist daher *Login*, der Auslöser dieser ist die Anfrage des Clients mit dem Namen *LoginRequest* und die Antwort den Servers ist der *LoginReply*.
Der Request besteht aus dem Benutzernamen (Feldname: *user*) und dem Passwort (Feldname: *pw*) des Benutzers und der Response besteht aus der Reaktion des Servers auf die Anfrage (Feldname: *result*).
Die Felder der Nachrichten müssen eine Nummerierung tragen, daher nummerieren wir diese immer fortlaufend mit 1 beginnend.

3.5 Server

In diesem Kapitel kümmern wir uns um die Implementierung des Servers, dazu bearbeiten wir die Datei /src/server/main.cpp.

3.5.1 header inkludieren

Zuerst inkludieren wir die notwendigen header:

```
1 #include "um.grpc.pb.h"
2 #include "CLI11.hpp"
3
4 #include <grpcpp/grpcpp.h>
5
6 using namespace std;
7
8 using grpc::Server;
9 using grpc::ServerBuilder;
10 using grpc::ServerContext;
11 using grpc::Status;
12 using grpc::StatusCode;
13
14 using um::UM;
15 using um::LoginRequest;
16 using um::LoginReply;
```

3.5.2 main

Da wir den Server mit der Auswahl zwischen ungesicherter und gesicherter Kommunikation implementieren wollen, fügen wir mittels CLI11 eine Auswahl beim Starten des Programms hinzu.
CLI11 kümmert sich um die Verarbeitung der Kommandozeilenparameter.
Über unserer main legen wir eine *classenum* an, mit welcher die Auswahl abgespeichert werden kann (natürlich könnten wir auch einfach *bool* oder was anderes verwenden, aber das macht das Ganze übersichtlicher). Diese nennen wir Communication

```
1 enum class Communication{Insecure, Secure};
```

Danach die benötigte *main*: Zuerst initialisieren wir unsere Kommandzeilenschnittstelle mit einem aussagekräftigen, kurzen Titel und legen das Format der Logging-Ausgabe mittels spdlog fest. Deklaration: $intmain(intargc, char * argv[])$

```
1     spdlog::set_pattern("[%H:%M:%S] [%^%l%$] %v");
2     CLI::App app{"Sets communiation security"};
```

Weiter fügen wir dieser eine Options-Gruppe für die Auswahl zwischen gesicherter und ungesicherte Kommunikation hinzu und fügen danach dieser Options-Gruppe die zur-Auswahl-stehenden Optionen hinzu.
Beim Erstellen der Gruppe geben wir dieser eine Bezeichnung und legen fest, dass man sich für maximal eine der Optionen (das machen wir mit $require_option(0, 1)$). Falls keine der Optionen gewählt wird, verwendet der Server automatisch ungesicherte Kommunikation.

```
1     auto comm_group = app.add_option_group("Communication Security")
2         ->require_option(0, 1);
```

Danach implementieren wir die Auswahl durch die Flags $-s, --secure$ und $-i, --insecure$. Damit man weiß, welche Flag tatsächlich gesetzt wurde, speichern wir das jeweils in einer Variable vom Typ *bool* ab (true = gesetzt, false = nicht gesetzt). Außerdem

hat jede Option eine Beschreibung vom Typ *string*, welche wir ausschließlich mittels des Typen *stringstream* machen, da sonst die Coding-Konventionen nicht immer eingehalten werden (manche Zeilen werden länger als 79 Zeichen) und Stringverkettung mittels $+=$ ist nicht besonders elegant.
Eine *stringstream*-Variable muss initialisiert werden, danach kann diese wie ein Stream erweitert werden mittels << und auf den String wird mittels *stringstream.str*(); zugegriffen.
Damit keine neue Variable erstellt werden muss, verwenden wir die bereits erstellte weiter, aber setzen diese auf einen Leerstring zurück: *ss.str*(*string*());.

```
1     stringstream ss;
2
3     ss << "use insecure communication [default]";
4     bool insecure = false;
5     comm_group->add_flag("-i,--insecure", insecure, ss.str());
6     ss.str(string());
7
8     ss << "use ssl/tls secured communication";
9     bool secure = false;
10    comm_group->add_flag("-s,--secure", secure, ss.str());
11    ss.str(string());
```

Dazu inkludieren wir am Beginn der Datei folgende Bibliotheken:

```
1 #include <iostream>
2 #include <string>
3 #include <sstream>
```

Danach starten wir die Kommandozeilenverarbeitung und je nachdem welche Art der Kommunikation gewählt wurde, starten wir unserer Run-Methode dementsprechend.
Die Variable *servername* ist der Name des Programms, das wir mittels Meson erzeugt haben und bei der Kommandozeilenverarbeitung am Server mittels CLI11 als Servername verwenden.

```
1     try {
2         app.parse(argc, argv);
3     } catch (const CLI::ParseError &e) {
4         return app.exit(e);
5     }
6
7     servername = argv[0];
8
9     if (secure) {
10        Run(Communication::Secure);
11    } else {
12        Run(Communication::Insecure);
13    }
```

Mit den oberen Schritten schließen wir die Implementierung der *main* ab.

Die globale Variable *servername* müssen wir allerdings noch zur Datei hinzufügen:

```
1 string servername;
```

3.5.3 Run

Jetzt kommen wir zu unserer Run-Funktion, in welcher der Server gestartet wird, auf welchem die Benutzerverwaltung möglich sein wird.
In der Variable *address* speichern wir die Adresse des Ports, der beim Starten des Servers

erzeugt wird. Noch dazu instaziieren wir ein Object der Klasse UmServiceImplementation, auf die im nachfolgenden Kapitel eingegangen wird.
Deklaration: *void Run(Communication security)*

```
1     string address("localhost:8888");
2     UmServiceImplementation service{&manager, &connections};
```

Zum Erstellen des Servers wird ein Objekts der Klasse *ServerBuilder* erstellt und für beide Möglichkeiten der Kommunikation werden die entsprechenden Referenzen erzeugt, zuerst für die ungesicherte Variante und dann für die SSL-Variante.

```
1     ServerBuilder builder;
2     auto creds_insec = grpc::InsecureServerCredentials();
3
4     grpc::SslServerCredentialsOptions sslOpts{};
5
6     sslOpts.pem_key_cert_pairs.push_back(
7         grpc::SslServerCredentialsOptions::PemKeyCertPair{
8             readFile("../openssl/server.key"),
9             readFile("../openssl/server.crt")});
10
11    auto creds_sec = grpc::SslServerCredentials(sslOpts);
```

Außerdem benötigen wir jetzt die Funktion *readFile*, die die Datei mit dem übergebenen Namen einliest:

```
1  grpc::string readFile(const grpc::string& filename) {
2      string data;
3      ifstream file(filename.c_str(), ios::in);
4
5      if (file.is_open()) {
6        stringstream ss;
7        ss << file.rdbuf ();
8
9        file.close ();
10
11       data = ss.str ();
12     } else {
13        cout << "problem reading file: " << filename << endl;
14     }
15
16     return data;
17 }
```

Dazu inkludieren wir folgende Direktiven:

```
1 #include <stdio.h>
2 #include <fstream>
```

Und wir benötigen die Zertifikat-Dateien, die wie folgt (mittels Terminal im angelegten Ordner *openssl*) erzeugt werden:

```
1 openssl req -newkey rsa:2048 -nodes -keyout server.key -x509 -out  \
2 server.crt   -subj "/C=CH/O=Test/OU=Server/CN=localhost"
3 openssl req -newkey rsa:2048 -nodes -keyout client.key -x509 -out  \
4 client.crt   -subj "/C=CH/O=Test/OU=Client/CN=client
```

Je nachdem welche Art der Kommunikation gewählt wurde, wird jetzt ein Port mit den dementsprechenden Referenzen dem *ServerBuilder* hinzugefügt und das erstellte Service wird zum *ServerBuilder* hinzugefügt.

Danach wird der eigentliche Server mithilfe des ServerBuilders initialisiert und gestartet

und eine entsprechende Meldung wird mittels spdlog ausgegeben. Danach lassen wir den Server noch auf Clients warten.

```
1
2     if (security == Communication::Secure) {
3         builder.AddListeningPort(address, creds_sec);
4     } else {
5         builder.AddListeningPort(address, creds_insec);
6     }
7
8     builder.RegisterService(&service);
9
10    std::unique_ptr<Server> server(builder.BuildAndStart());
11    spdlog::info("Server listening on port: {}", address);
12
13    server->Wait();
```

Dazu müssen wir spdlog wie folgt einbinden:

```
1 #include "spdlog/spdlog.h"
2 #include "spdlog/sinks/basic_file_sink.h"
```

Als nächstes wird die Benutzerverwaltung am Server implementiert, dazu werden die Hilfsklassen *User*, welche einen Benutzer implementiert, und *Manager*, welcher die Benutzer verwaltet, erstellt.

3.5.4 User

Zuerst implementieren wir die header-Datei der User-Klasse (include/user.h).
Dazu schreiben wir zuerst einen include-Guard und es werden alle notwendigen Präprozessordirektiven eingebunden.
Ein Benutzer hat bestimmte Rechte für ein Objekt, für das Abspeichern dieser Rechte wird eine *enumclass* mit dem Namen *Action* erstellt. *Action* enthält alle Rechte, die einem Benutzer zugeteilt werden können, also das Löschen, das Schreiben, das Ausführen und das Lesen eines Objekts sollen zugeteilt werden können.

```
1 #ifndef USER_H
2 #define USER_H
3
4 #include <string>
5 #include <map>
6 #include <list>
7
8 enum class Action {Read,Write,Execute,Delete};
```

Jetzt zur eigentlichen Klasse:
Ein Benutzer hat einen Namen und ein Passwort, außerdem hält er für Objekte bestimmte Rechte, was in Form einer *map* implementiert ist, bei der jedem Objekt eine Liste von Rechten zugeordnet wird.

Danach wird ein Konstruktor implementiert, der dem Benutzer den übergebenen Namen und das übergebene Passwort zuweist, gefolgt von den getter- und setter-Methoden für diese beiden Felder.

Danach folgt eine print-Methode und eine Operator-Überladung, sodass später im Manager überprüft werden kann, ob zwei Nutzer identisch sind, sodass man alle Benutzer nach dem Namen filtern kann.

Die nachfolgenden Funktionen dienen dem Setzen, dem Entfernen, dem Ausgeben und dem Überprüfen von den Rechten des Benutzers.

```
1  class User {
2  private:
3      std::string name;
4      std::string pw;
5
6      std::map<std::string, std::list<Action>> rights;
7  public:
8      User (std::string _name, std::string _pw) : name{_name}, pw{_pw} {}
9
10     std::string get_name();
11
12     void set_name(std::string);
13
14     std::string get_pw();
15
16     void set_pw(std::string);
17
18     void print();
19
20     bool operator == (const User& user) const { return name == user.name; }
21
22     bool set_right(std::string, std::string);
23
24     bool rem_right(std::string);
25
26     void print_rights(std::string);
27
28     void print_rights();
29
30     std::string get_rights(std::string object);
31
32     std::string get_rights();
33
34     bool has_rights_for_obj(std::string object);
35
36     bool allowed_to_read(std::string object);
37
38     bool allowed_to_write(std::string object);
39
40     bool allowed_to_execute(std::string object);
41
42     bool allowed_to_delete(std::string object);
43 };
44
45 #endif /* end of include guard: */
```

Schauen wir uns jetzt die Implementierung der oben aufgelisteten Funktionen an (src/user.cpp).

Zuerst werden die wichtigen Direktiven eingebunden, dann die setter- und getter-Funktionen und eine einfache Funktion zur Ausgabe der Benutzerdaten:

```
1 #include "user.h"
2
3 #include <string>
4 #include <iostream>
5 #include <algorithm>
6 #include <map>
```

```
 7 #include <list>
 8 #include <sstream>
 9
10 using namespace std;
11
12 string User::get_name() {
13     return name;
14 }
15
16 void User::set_name(string name_) {
17     name = name_;
18 }
19
20 string User::get_pw() {
21     return pw;
22 }
23
24 void User::set_pw(string pw_) {
25     pw = pw_;
26 }
27
28 void User::print() {
29     cout << "Name: " << name << " Password: " << pw << endl;
30 }
```

Folgende Funktionen dienen dem Überprüfen und dem Ausgeben von Rechten bezüglich eines Objektes.

Da die Ausgabe der Rechte dem Format *rwxd* entsprechen soll müssen diese dementsprechend aus der Liste der Rechte ausgelesen und in das entsprechende Format gebracht werden, das passiert wie folgt.

```
 1 bool User::has_rights_for_obj(string object) {
 2     return (rights.find(object) != rights.end());
 3 }
 4
 5 string User::get_rights(string object) {
 6     stringstream ss;
 7
 8     ss << "Name: " << name << " Object: " << object << " Rights: ";
 9     list<Action>* actions = &rights[object];
10
11     if (find(actions->begin(), actions->end(), Action::Read) != actions->end()) {
12         ss << "r";
13     } else {
14         ss << "-";
15     }
16
17     if (find(actions->begin(), actions->end(), Action::Write) != actions->end()) {
18         ss << "w";
19     } else {
20         ss << "-";
21     }
22
23     if (find(actions->begin(), actions->end(), Action::Execute) != actions->end()) {
24         ss << "x";
25     } else {
26         ss << "-";
27     }
28
29     if (find(actions->begin(), actions->end(), Action::Delete) != actions->end()) {
```

```
30          ss << "d";
31      } else {
32          ss << "-";
33      }
34
35      ss << endl;
36
37      return ss.str();
38  }
```

Die nächsten drei Funktionen dienen der Ausgabe der Rechte, entweder im Bezug auf ein Objekt oder alle Rechte, die ein Benutzer besitzt.

```
1  void User::print_rights(string object) {
2      cout << get_rights(object);
3  }
4
5  string User::get_rights() {
6      stringstream ss;
7
8      for (auto& obj : rights) {
9          ss << get_rights(obj.first);
10     }
11
12     return ss.str();
13 }
14
15 void User::print_rights() {
16     for (auto& obj : rights) {
17         print_rights(obj.first);
18     }
19 }
```

Da die Rechte bei der Eingabe auf der Kommandozeile eben dem Format *rwxd* entsprechen, müssen diese wieder in die Form von einer *list* mit Elementen vom Typ *Action* gebracht werden.

```
1  bool User::set_right(string object, string right) {
2      if (right.length() == 4) {
3          if (right.compare("----") == 0) {
4              rem_right(object);
5              return true;
6          }
7
8          if (right.at(0) == 'r') {
9              rights[object].push_back(Action::Read);
10         } else if (right.at(0) == '-') {
11             rights[object].remove(Action::Read);
12         } else {
13             return false;
14         }
15
16         if (right.at(1) == 'w') {
17             rights[object].push_back(Action::Write);
18         } else if (right.at(1) == '-') {
19             rights[object].remove(Action::Write);
20         } else {
21             return false;
22         }
23
24         if (right.at(2) == 'x') {
25             rights[object].push_back(Action::Execute);
```

```
26          } else if (right.at(2) == '-') {
27              rights[object].remove(Action::Execute);
28          } else {
29              return false;
30          }
31
32          if (right.at(3) == 'd') {
33              rights[object].push_back(Action::Delete);
34          } else if (right.at(3) == '-') {
35              rights[object].remove(Action::Delete);
36          } else {
37              return false;
38          }
39
40          return true;
41      }
42
43      return false;
44  }
```

Die nachfolgende Funktion dient dem Entziehen von Rechten bezüglich eines Objekts von einem Benutzer.

```
1  bool User::rem_right(string object) {
2      if (rights.find(object) != rights.end()) {
3          rights.erase(object);
4
5          return true;
6      }
7
8      return false;
9  }
```

Wenn der Benutzer mit einem Objekt eine bestimmte Aktion durchführen will, sei es Lesen, Schreiben, Ausführen oder Löschen, muss überprüft werden, ob der Benutzer die entsprechenden Rechte dafür besitzt.

```
1  bool User::allowed_to_read(string object) {
2      list<Action>* tmp = &rights[object];
3      return find(tmp->begin(), tmp->end(), Action::Read) != tmp->end();
4  }
5
6  bool User::allowed_to_write(string object) {
7      list<Action>* tmp = &rights[object];
8      return find(tmp->begin(), tmp->end(), Action::Write) != tmp->end();
9  }
10
11 bool User::allowed_to_execute(string object) {
12     list<Action>* tmp = &rights[object];
13     return find(tmp->begin(), tmp->end(), Action::Execute) != tmp->end();
14 }
15
16 bool User::allowed_to_delete(string object) {
17     list<Action>* tmp = &rights[object];
18     return find(tmp->begin(), tmp->end(), Action::Delete) != tmp->end();
19 }
```

3.5.5 Manager

Als nächstes wird die Klasse *Manager* implementiert. Auch hier beginnen wir mit der head-File(include/server/manager.h).

Zuerst, wie immer, der include-Guard und die Direktiven.

```
1 #ifndef MANAGER_H
2 #define MANAGER_H
3
4 #include<list>
5 #include<string>
6
7 #include "user.h"
```

Die Klasse *Manager* beinhaltet eine Liste von den Benutzern, die verwaltet werden und die Funktionen dienen zum verwalten eben dieser.
Außerdem beinhaltet sie einen Standardkonstruktor.
Auf die spezifischen Funktionen wird im nächsten Abschnitt eingegangen.

```
1 class Manager {
2 private:
3     std::list<User> users;
4 public:
5     Manager () {}
6
7     void print();
8
9     bool contains(std::string);
10
11    bool add(std::string, std::string);
12
13    bool mod_pw(std::string, std::string);
14
15    bool mod_name(std::string, std::string);
16
17    bool del(std::string);
18
19    bool set_right(std::string, std::string, std::string);
20
21    bool rem_right(std::string, std::string);
22
23    bool print_rights(std::string, std::string);
24
25    bool login(std::string, std::string);
26
27    User* get_user(std::string);
28 };
29
30 #endif /* end of include guard: */
```

In den Nachfolgenden Code-Abschnitten erstellen wir die C++-Datei von *Manager* (src/-server/manager.cpp).

Auch hier allem voran der include-Guard und die Direktiven:

```
1 #include "manager.h"
2 #include "user.h"
3
4 #include <string>
5 #include <iostream>
6
7 using namespace std;
```

Die folgende Funktion dient der Ausgabe aller Benutzer:

```
1 void Manager::print() {
2     for (auto& user : users) {
3         user.print();
4     }
5 }
```

get_user filtert die Instanz der Klasse $User$ aus der Liste der Benutzer und gibt diese zurück, falls dieser existiert, sonst wird ein $nullptr$ zurückgegeben.

```
1 User* Manager::get_user(string name) {
2     for (auto& user : users) {
3         if (user.get_name().compare(name) == 0) {
4             return &user;
5         }
6     }
7
8     return nullptr;
9 }
```

Die Funktionen im folgenden Codeabschnitt haben folgende Funktionalitäten:

- $contains$ überprüft, ob ein Benutzer mit einem bestimmten Namen existiert.
- add legt, falls nicht bereits einen Benutzer mit diesem Namen existiert, einen neuen Benutzer mit Name und Passwort an und fügt diesen in der Liste der Benutzer ein.
- mod_pw setzt, falls ein Benutzer mit dem übergebenen Namen existiert, ein neues Passwort für eben diesen
- mod_name setzt, falls ein Benutzer mit dem übergebenen Namen existiert, einen neuen Namen für eben diesen
- del löscht, falls ein Benutzer mit dem übergebenen Namen existiert, den Benutzer
- set_rights setzt, falls ein Benutzer mit dem übergebenen Namen existiert, neue Rechte für den Benutzer bezüglich des übergebenen Objekts
- rem_rights löscht, falls ein Benutzer mit dem übergebenen Namen existiert, löscht dessen Rechte für den Benutzer bezüglich des übergebenen Objekts
- $print_rights$ gibt, falls ein Benutzer mit dem übergebenen Namen existiert, dessen Rechte für den Benutzer bezüglich des übergebenen Objekts aus

Alle dieser Funktionen geben mittels eines *bool*-Wertes zurück, ob die Aktion erfolgreich war. $false$ ist dieser Wert, wenn KEIN Benutzer mit dem übergebenen Namen existiert (bei *add*, wenn EIN Benutzer mit dem übergebenen Namen existiert).

```
1  bool Manager::contains(string name) {
2      return (get_user(name) != nullptr);
3  }
4
5  bool Manager::add(string name, string pw) {
6      if (!contains(name)) {
7          users.push_back(User{name, pw});
8          return true;
9      }
10
11     return false;
12 }
13
14 bool Manager::mod_pw(string name, string pw_) {
15     User* user = get_user(name);
16
17     if (user != nullptr) {
18         user->set_pw(pw_);
```

```
19          return true;
20      }
21
22      return false;
23  }
24
25  bool Manager::mod_name(string name, string new_name) {
26      User* user = get_user(name);
27
28      if (user != nullptr) {
29          user->set_name(new_name);
30          return true;
31      }
32
33      return false;
34  }
35
36  bool Manager::del(string name) {
37      User* user = get_user(name);
38
39      if (user != nullptr) {
40          users.remove(*user);
41          return true;
42      }
43
44      return false;
45  }
46
47  bool Manager::set_right(string name, string object, string right) {
48      User* user = get_user(name);
49
50      if (user != nullptr) {
51          return user->set_right(object, right);
52      }
53
54      return false;
55  }
56
57  bool Manager::rem_right(string name, string object) {
58      User* user = get_user(name);
59
60      if (user != nullptr) {
61          return user->rem_right(object);
62      }
63
64      return false;
65  }
66
67  bool Manager::print_rights(string name, string object) {
68      User* user = get_user(name);
69
70      if (user != nullptr) {
71          if (!object.empty()) {
72              user->print_rights(object);
73          } else {
74              user->print_rights();
75          }
76
77          return true;
78      }
79
```

```
80     return false;
81 }
```

Die nächste Funktion get_user gibt einen Pointer auf die Instanz des Typs $User$ aus der Liste der Benutzer zurück, falls ein Benutzer mit diesem Namen existiert, ansonsten wird ein *nullptr* zurückgegeben.

```
1  bool Manager::print_rights(string name, string object) {
2      User* user = get_user(name);
3
4      if (user != nullptr) {
5          if (!object.empty()) {
6              user->print_rights(object);
7          } else {
8              user->print_rights();
9          }
10
11         return true;
12     }
13
14     return false;
15 }
```

Die letzte Funktion dient der Anmeldung eines Clients, der am Anfang einen Benutzernamen und ein Passwort eingeben muss. In *login* wird nun überprüft, ob die Daten stimmen und eben dadurch entschieden, ob sich der Client in weiterer Folge als eben dieser Benutzer anmelden kann.

```
1  bool Manager::login(string name, string pw) {
2      User* user = get_user(name);
3
4      if (user != nullptr) {
5          return (user->get_pw().compare(pw) == 0);
6      }
7
8      return false;
9  }
```

Jetzt müssen wir am Server (src/server/main.cpp) natürlich die header-Dateien einbinden:

```
1 #include "manager.h"
2 #include "user.h"
```

Und wir erstellen am Server eine globale Instanz der Klasse $Manager$, die alle Benutzer beinhaltet, sowie eine Liste der aktuell-verbundenen Benutzer (Es werden hier Zeiger verwendet, sodass nicht immer der Nutzer mit dem gleichen Namen gefiltert werden muss und dass man Änderungen nicht in der Liste um am Manager durchführen muss):

```
1 Manager manager{};
2 list<User*> connections;
```

3.6 Benutzerverwaltung am Server

Anstatt den Server nur warten zu lassen, wird jetzt eine Benutzerverwaltung implementiert (src/server/main.cpp). Dazu ersetzen wir zuerst die letzte Zeile der Run-Funktion durch folgenden Codeabschnitt:

```
1 char *server_name = new char[servername.size()+1];
2 strcpy(server_name, servername.c_str());
3
```

```
4 string input;
5
6 while (true) {
7     cout << servername << "$ " << flush;
8     getline(cin, input);
9
10    if (input.length() > 0) {
11        vector<char *> result;
12        istringstream iss(input);
13
14        result.push_back(server_name);
15
16        for(string s; iss >> s;) {
17            char *c = new char[s.size() + 1];
18            strcpy(c, s.c_str());
19
20            result.push_back(c);
21        }
22
23        clp_server(result.size(), result.data());
24    }
25 }
```

Da beim Verarbeiten der Eingaben wieder CLI11 verwendet wird, müssen wir die passenden Argument dafür erzeugen: *int argc, char * argv[]* Zuerst wandeln wir den Inhalt der globalen Variable *servername* vom Typ *string* zum Typ *char** um, sodass wir diesen bei einer Eingabe an erster Stelle in eine Variable vom Typ *vector* einfügen können.
Danach implementieren wir innerhalb einer while-Schleife die Benutzereingabe des Servers, sodass diese eben ganze Zeit auf eine Eingabe wartet, solange der Server gestartet ist.
In dieser Schleife lesen wir einen *string* ein und wandeln diesen danach in einen Vektor um, der alle *char**-Elemente enthält, die beim Zerteilen der Eingabe bei allen Leerzeichen, entstehen.

Dazu inkludieren wir folgende Direktiven:

```
1 #include <iostream>
2 #include <vector>
```

Danach wird aus dem Vektor ein *char**-Array erzeugt und die Funktion *clp* mit der Deklaration: *int clp_server(int argc, char * argv[])*, die jetzt erstellt wird, die folgende Benutzerverwaltung implementiert, wird aufgerufen:

```
1 Manages users and rights
2 Usage: ./server [OPTIONS]
3
4 Options:
5   -h,--help                   Print this help message and exit
6   -u,--user TEXT Excludes: --list
7                               Select user by name
8                                (list existing users with -l,--list,
9                                modify with -m, --modify
10                               add with -a,--add,
11                               delete with -d,--delete)
12  -p,--password TEXT Needs: --user
13                              Set password for new user
14                               or for existing user
15                               (--modify or --add needs to be set)
16  -n,--new-name TEXT Needs: --user --modify
17                              Set new username for existing user
18  -o,--object TEXT Needs: --user Excludes: --modify --delete
```

```
                              Select object by name
                                (list existing rights with --list-rights,
                                modify existing rights with --modify-rights,
                                add new rights with --add-rights,
                                delete existing rights with --delete-rights)
[Option Group: User-Options]

  [At most 1 of the following options are allowed]
  Options:
    -l,--list Excludes: --user  Prints list of all users
    -a,--add Needs: --user --password
                                Add new user
    -m,--modify Needs: --user Excludes: --object
                                Modify existing user
    -d,--delete Needs: --user Excludes: --object
                                Delete existing user
[Option Group: Right-Options]

  [At most 1 of the following options are allowed]
  Options:
    --list-rights Needs: --user Excludes: --set-rights --remove-rights
                                List existing rights
                                 from user or user and object
    -s,--set-rights TEXT Needs: --user --object Excludes: --list-rights --remove-
    rights
                                Set or modify rights
                                 4 letters in correct order: rwxd
                                 replace letter with - to not set or delete right
    -r,--remove-rights Needs: --user --object Excludes: --list-rights --set-rights
                                Delete existing rights
```

3.6.1 Service-Implementierung

In der Datei src/server/main.cpp werden wir jetzt die Klasse *UmServiceImplementation* implementieren, die die in unserer um.proto festgelegten Fuktionen implementiert. Das funktioniert wie folgt:

```
Status Login(
    ServerContext* context,
    const LoginRequest* request,
    LoginReply* reply
) override {}
```

Jetzt wird die gewünschte Reaktion des Servers auf die Anfrage des Clients implementiert: Den vom Client eingegebenen Benutzernamen und das eingegebene Passwort werden mit den Funktionen der globalen Variable der Klasse *Manager* überprüft und dementsprechend wird die Anmeldung genehmigt oder eben nicht.
Zuerst fügen wir den Benutzer zu den angemeldeten Benutzern hinzu (falls die Anmeldedaten korrekt sind), dann setzen wir die Antwort an den Client dementsprechend, danach wird am Server mittels spdlog ausgegeben, dass sich ein Client angemeldet hat und am Ende wird die Antwort an der Client gesendet.

```
string user = request->user();
string pw = request->pw();

if (!manager->contains(user)) {
    stringstream reply;
    reply << "login failed!\n" << user << " does not exist!";

```

```
8       cout << endl;
9       spdlog::warn("{} tried to connect! (user does not exist)", user);
10      cout << servername << "$ " << flush;
11
12      return Status(StatusCode::CANCELLED, reply.str());
13  } else if (!manager->login(user, pw)){
14      stringstream reply;
15      reply << "login failed!\nwrong password for " << user;
16
17      cout << endl;
18      spdlog::warn("{} tried to connect! (wrong password)", user);
19      cout << servername << "$ " << flush;
20
21      return Status(StatusCode::CANCELLED, reply.str());
22  } else if (manager->login(user, pw)) {
23      connections->push_back(manager->get_user(user));
24
25      reply->set_result("login successful!\nwelcome " + user + "!");
26
27      cout << endl;
28      spdlog::info("{} connected!", user);
29      cout << servername << "$ " << flush;
30
31      return Status::OK;
32  } else {
33      return Status(StatusCode::ABORTED, "");
34  }
```

Da der Client sich nicht nur anmelden können soll, sondern auch eine Abmeldung eines Benutzers, das Löschen, das Bearbeiten, das Lesen und das Schreiben eines Objekts und das Ausgeben der Berechtigungen möglich sein soll erweitern wir unser Service *UM* in unserer Datei *um.proto* um folgendes:

```
1 rpc ListRights (ListRequest) returns (ListReply) {}
2 rpc Delete (DeleteRequest) returns (DeleteReply) {}
3 rpc Read (ReadRequest) returns (ReadReply) {}
4 rpc Write (WriteRequest) returns (WriteReply) {}
5 rpc Execute (ExecuteRequest) returns (ExecuteReply) {}
6 rpc Logout (LogoutRequest) returns (LogoutReply) {}
```

Diese Funktionen müssen in der main.cpp wieder eingebunden und dementsprechend überschrieben werden. In der Datei *um.proto* müssen die Anfragen und Antworten wieder mit Feldern versehen und diese müssen wieder nummeriert werden, wie bei der Funktion *Login*.

3.7 Client

In diesem Kapitel wird nun der Client implementiert.

3.7.1 includes

Als allererster Schritt erfolgt wie immer das Einbinden der benötigten Direktiven:

```
1 #include "um.grpc.pb.h"
2 #include "CLI11.hpp"
3 #include "user.h"
4
5 #include "spdlog/spdlog.h"
6 #include "spdlog/sinks/basic_file_sink.h"
7
```

```
#include <grpcpp/grpcpp.h>

#include <string>
#include <fstream>
#include <limits>
#include <sstream>
#include <signal.h>

using namespace std;

using grpc::Server;
using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;

using um::UM;
using um::LoginRequest;
using um::LoginReply;
using um::ListRequest;
using um::ListReply;
using um::DeleteRequest;
using um::DeleteReply;
using um::ReadRequest;
using um::ReadReply;
using um::WriteRequest;
using um::WriteReply;
using um::ExecuteRequest;
using um::ExecuteReply;
using um::LogoutRequest;
using um::LogoutReply;
```

3.7.2 main

Die Funktion *main* hat den gleichen Aufbau wie am Server.

3.7.3 Run

Je nach Auswahl in der Main wird entsprechend ein Kanal mit dementsprechenden Referenzen initialisiert, also entweder ein Gesicherter oder ein Ungesicherter. Mithilfe dieses Kanals wird dann der globale Pointer auf eine Instanz von *UmClient* initialisert.

```
string address("localhost:8888");

if (security == Communication::Secure) {
    grpc::SslCredentialsOptions sslOpts;
    sslOpts.pem_root_certs = readFile("../openssl/server.crt");

    // Create an SSL ChannelCredentials object.
    auto channel_creds = grpc::SslCredentials(sslOpts);

    // Create a channel using the credentials created in the previous step.
    auto channel = grpc::CreateChannel(address, channel_creds);

    client = new UmClient(channel);
} else {
    // Create a default Insecure ChannelCredentials object.
    auto channel_creds = grpc::InsecureChannelCredentials();

    // Create a channel using the credentials created in the previous step.
```

```
19     auto channel = grpc::CreateChannel(address, channel_creds);
20
21     client = new UmClient(channel);
22 }
```

Wie am Server ist die identische Funktion *readFile* zu definieren.
Natürlich muss auch die globale Variable erstellt werden:

```
1 UmClient* client = nullptr;
```

Danach wird der Client um eine Eingabe von Benutzernamen und Passwort gefragt, solange bis eine gültige Eingabe erfolgt ist und bei jeder Eingabe wird die Login-Funktion am Server aufgerufen.
Außerdem wird der Client beendet, wenn kein Server mit der selben Sicherheitsstufe aktiv ist.

```
1 int response = -1;
2 string user;
3 string pw;
4
5 while (response != 0) {
6   cout << "username: ";
7   cin >> user;
8
9     cout << "password: ";
10  cin >> pw;
11
12    response = client->Login(user, pw);
13
14    if (response == -1) {
15        cout << "no server with your communication type running\n"
16            << "try restarting client --help" << endl;
17        exit(-1);
18    }
19 }
```

Beenden mittels Ctrl+C abfangen

Als nächstes wird ein *signal* implementiert, sodass die Logout am Server aufgerufen wird, sobald der Client mittels Crtl+C beendet wird.

```
1 signal(SIGINT, signal_handler);
```

Der $signal_handler$ ist ebenfalls zu implementieren, der eine Ausgabe am Client tätigt, die Logout-Funktion am Server aufruft und dann den Server beendet:

```
1 void signal_handler(int signum) {
2     cout << "logging off..." << endl;
3     client->Logout();
4     exit(signum);
5 }
```

Benutzerschnittstelle

Da der Client nun gestartet und als Benutzer angemeldet ist, wird auch dieser mittels CLI11 mit folgender Benutzerschnittstelle (in der Funktion *clp*)versehen:

```
1 Manages users and rights
2 Usage: user [OPTIONS]
3
```

```
Options:
  -h,--help                   Print this help message and exit
  -o,--object TEXT Excludes: --logout
                              Select object by name
                               (list existing rights for object with -l,--list,
                               read with -r, --read
                               write with -w,--write,
                               execute with -x,--execute,
                               delete with -d,--delete)
[Option Group: Operations]

  [At most 1 of the following options are allowed]
  Options:
    -l,--list                  list existing rights
                                from user or user and object
    -r,--read Needs: --object  read object
    -w,--write Needs: --object  write object
    -x,--execute Needs: --object
                                execute object
    -d,--delete Needs: --object delete object
    --logout Excludes: --object logout user and exit programm
```

Die *run*-Funktion muss dazu mit der gleichen Funktionalität wie die *run*-Funktion des Servers erweitert werden:

```
char *username = new char[user.size()+1];
strcpy(username, user.c_str());
string input;
cin.ignore();
while (true) {
    cout << user << "$ " << flush;
    getline(cin, input);

    if (input.length() > 0) {
        vector<char *> result;
        istringstream iss(input);

        result.push_back(username);

        for(string s; iss >> s;) {
            char *c = new char[s.size() + 1];
            strcpy(c, s.c_str());

            result.push_back(c);
        }

        clp_user(result.size(), result.data());
    } else {
    }
}
```

Als letzten Punkt muss die Klasse *UmClient* erstellt werden, welche die Anfragen der Funktionen aus *um.proto* in folgender Form implementiert:

```
int Login(string username, string pw) {
        LoginRequest request;

        request.set_user(username);
        request.set_pw(pw);

        LoginReply reply;

```

```
9         ClientContext context;
10
11        grpc::Status status = stub->Login(&context, request, &reply);
12
13        if (status.ok()){
14            user = username;
15            cout << "Answer received: "
16                << reply.result() << endl;
17            return 0;
18        } else if (status.error_code() == 1){
19            cout << "Answer received: "
20                << status.error_code() << ": " << status.error_message()
21                << endl;
22            return 1;
23        } else {
24            return -1;
25        }
26    }
```

Zuerst wird ein Request passend zu der Funktion erzeugt, danach werden diesem die in der *.proto*-Datei festgelegten Felder zugewiesen und die zusätzlich für den Aufruf der Funktion notwendigen Felder vom Typ *ClientContext* und vom Typ der entsprechenden Antwort des Servers werden erstellt.
Die Funktion am Server wird aufgerufen und die Antwort wird verarbeitet.

Die restlichen Funktionen zum Lesen, Schreiben, Ausführen und Löschen eines Objekts und das Abmelden des Benutzers sind gleich zu implementieren.

Kapitel 4

Kompilieren und Ausführen

Nach der Fertigstellung des Projektes (und normalerweise schon zwischendurch zum Testen) kann dieses nun durch das Ausführen folgender Befehle im *build*-Ordner im Terminal erzeugt werden:

```
1  $ meson ..
2  $ ninja
```

Ausgeführt werden die von-*ninja*-gefertigten ausführbaren Programme wie folgt (der Name des ausführbaren Programms ist der, der in der *meson.build*-Datei angegeben wurde):

```
1  $ ./server [-s | -i | -h]
2  $ ./client [-s | -i | -h]
```

Danach ist die Benutzerverwaltung auf beiden so zu handhaben, wie wir diese implementiert haben (in der Funktion *clp*). Am Besten lässt man sich dieser aber nochmal sicherheitshalber mittels der Eingabe von $--help$ anzeigen ;-).