Entwicklung eines Remote Method Invocation Systems

Emil Watz, 5AHIF
Beispielnummer: 44
Matrikelnummer: 19

11. April 2022

Inhaltsverzeichnis

Kapitel 1

Theoretischer Hintergrund

Das Ziel dieses Projekts ist die Entwicklung eines RMI-Systems basierend auf der `asio`-Bibliothek[1] in `C++`. Im Folgenden werden ähnliche Systeme und Konzepte beschrieben. Daraus werden dann die Anforderungen an das System abgeleitet.

1.1 Was ist ein RMI-System

Remote Method Invocation - auch Remote Procedure Call (RPC) genannt - beschreibt den Aufruf von „entfernten Funktionen“, also Methoden die von einem anderen Prozess ausgeführt werden. Dieser Prozess muss jedoch nicht am gleichen Rechner ausgeführt werden. Im Kontext von verteilten Systemen werden die Prozesse meist von Rechnern in unterschiedlichen Netzwerken ausgeführt. Das bedeutet, dass Funktionsaufrufe und Rückgabewerte serialisiert, und über das Netzwerk übertragen werden müssen.

1.2 Zusammenhang mit Client/Server-Modell

Das Konzept basiert auf dem Client/Server-Modell, bei dem der Client Anfragen sendet und der Server antwortet. Jedoch wird eine zusätzliche Abstraktion eingeführt. Der Client ruft eine lokale Funktion auf, die dem Server eine Anfrage sendet. Dieser führt eine Funktion aus und sendet den Rückgabewert serialisiert zurück. Dabei verhält sich der Aufruf am Client (in der Theorie) genau gleich wie ein lokaler Aufruf.

1.3 RMI-Konzept

Es gibt kein allgegenwärtiges RMI-Protokoll oder einen Standard, jedoch haben viele Implementierungen eine ähnliche Struktur.

1.3.1 Allgemeines Konzept

[1] `https://think-async.com/Asio/`
[2] `https://docs.microsoft.com/de-de/previous-versions/dn151205(v=technet.10)`

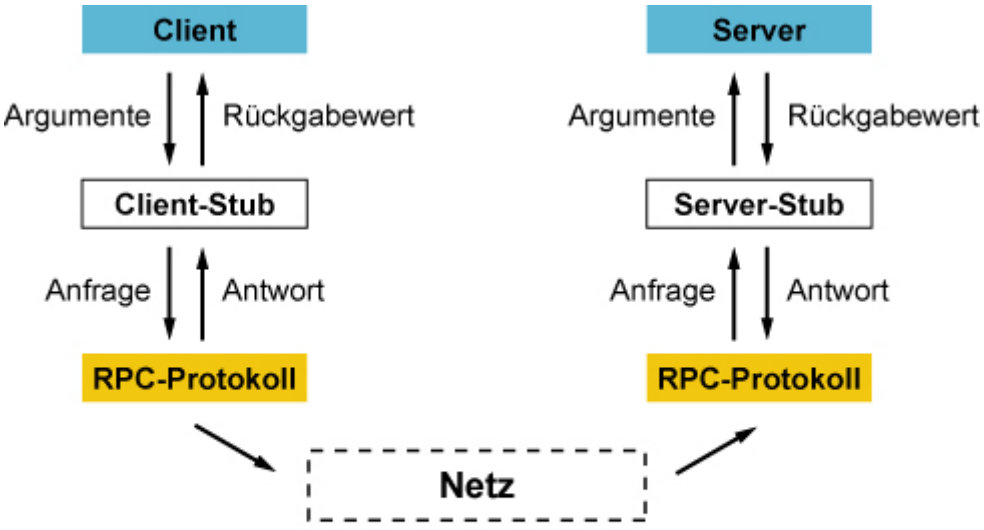

Abbildung 1.1: RPC-Prinzip[2]

Wie in der Abbildung zu sehen ist, gibt es im Grunde 3 „Schichten“.

1. Die oberste ist der Client und der Server. Sie sind die Anwender des Systems. Der Client ruft Funktionen am Server über den Client-Stub auf und gibt Argumente mit. Er kümmert sich aber nicht darum, wie die Aufrufe zum Server kommen. Der Server definiert die Funktionen, die vom Client aufgerufen werden können. Er gibt Rückgabewerte zurück, ist aber nicht dafür zuständig, wie diese zum Client kommen. Je nach Implementierung können auch Exceptions geworfen werden, diese werden in diesem Modell jedoch vernachlässigt

2. Eine Ebene darunter befindet sich der Client- und Server-Stub. Sie sind für die Serialisierung und Deserialisierung der Argumente und Rückgabewerte zuständig.

3. Die serialisierten Daten der Stubs werden dann von dem RPC-Protokoll über das Netzwerk übertragen, sodass sie wieder deserialisiert werden können[3].

1.3.2 Java-RMI

Eine konkrete Implementierung des, im vorherigen Abschnitt beschriebenem, Konzepts ist die Java-proprietäre Remote Method Invocation. Erweitert wird

[3]Prof. Dr. Stephan Euler. *Remote Procedure Call – RPC*. abgerufen am 03.10.2021. URL: `https://docs.microsoft.com/de-de/previous-versions/dn151205(v=technet.10)`.

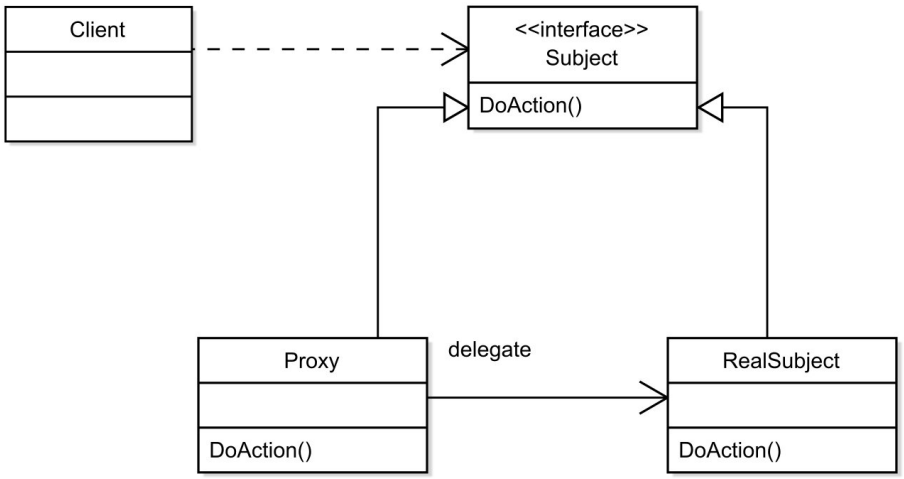

Abbildung 1.2: Simples Klassendiagram des Proxy-Patterns[6]

es durch einen Namensdienst, die sogenannte Registry, bei dem Objekte angemeldet und abgefragt werden können. Außerdem orientiert sich das Java-RMI an dem Proxy-Design-Pattern, welches im Folgenden beschrieben wird.

1.4 Das Proxy-Pattern

Das Proxy-Pattern - auf Deutsch Stellvertreter-Entwurfsmuster - ist ein Design Pattern mit dem Ziel, einen Platzhalter für ein anderes Objekt bereitzustellen. So lässt sich der Zugriff auf das Objekt kontrollieren. Wie in Abbildung 1.2 zu sehen ist, besteht das Pattern grundlegend aus Komponenten. Der Platzhalter und das echte Objekt erben von einem Interface, welches die Methoden vorgibt. Beim Aufruf einer Funktion des Proxys, leitet dieser den Aufruf an das echte Objekt weiter[4].

1.4.1 Proxy-Pattern im Kontext von RMI

In einem RMI-System stellt der Client-Stub den Proxy beziehungsweise Stellvertreter dar. In diesem Fall handelt es sich um einen **remote proxy**, also einen lokalen Stellvertreter für ein Objekt in einem anderen Addressraum[5]. Das echte Objekt wäre in Abbildung 1.1 der Server. Die Weiterleitung eines Funktionsaufrufes ist bei einem RMI-System schwerer als bei einer rein lokalen Implementierung, da das „echte Objekt“ von einem anderen Prozess instanziert wird und der Aufruf über Netzwerk übertragen werden muss.

[4] vgl. Erich Gamma u. a. *Design Patterns. Elements of Reusable Object-Oriented Software.* Englisch. Prentice Hall, Juli 1997, Seite 207.

[5] vgl. Gamma u. a., *Design Patterns. Elements of Reusable Object-Oriented Software.* Seite 208.

[6] `https://www.programmingwithwolfgang.com/proxy-pattern-in-net-core/`

1.5 Anforderungen an ein RMI-System

Auf Basis der bisher beschriebenen Informationen soll ein RMI-System implementiert werden. Die Anforderungen werden nun im Folgenden beschrieben.

1.5.1 Entferntes Aufrufen

Ein Client soll einen **Remote-Proxy** in Form eines Client-Stubs instanzieren können, der die Methoden eines Interfaces implementiert. Beim Aufruf einer Methode wird der Aufruf über das Netzwerk gesendet. Die Methode wird dann am Server, der dasselbe Interface implementiert, aufgerufen. Dabei muss es die Möglichkeit geben, Parameter mitzusenden. Nachdem die Funktion am Server ausgeführt wurde, muss der Rückgabewert wieder zurückgesendet werden.

1.5.2 Exceptions

Funktionen am Server können auch Exception werfen. Daher müssen diese ebenfalls an den Client-Stub zurückgesendet werden und dort geworfen werden.

Kapitel 2

Umsetzung

2.1 Klassendiagram

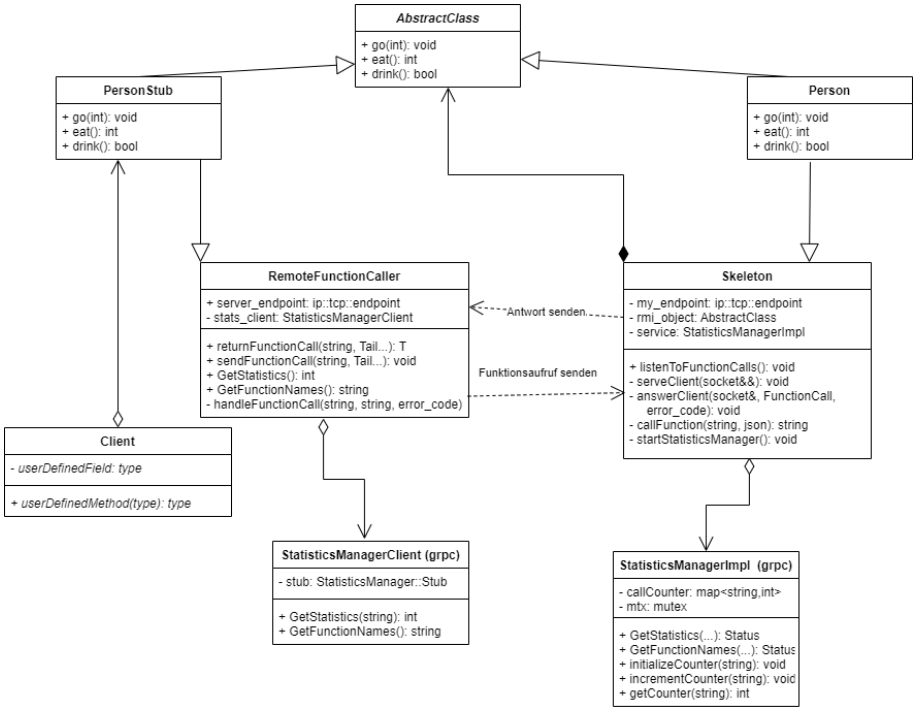

Abbildung 2.1: Klassendiagram des RMI Systems

Auf Basis der gesammelten Informationen aus dem letzten Kapitel wurde ein Klassendiagramm entwickelt. Dieses soll ein RMI-System anhand des Proxy-Patterns darstellen.

Die Klasse `Person` ist dabei ein Beispiel. Diese wird, genauso wie die Methoden der `AbstractClass`, vom Benutzer erstellt und kann auch anders heißen. Genauso ist es beim PersonStub. Die Verwendung dieses Systems wird in einem späteren Kapitel (3) genauer beschrieben.

2.1.1 Abstrakte Klasse (AbstractClass)

Wie im Abschnitt über das Proxy-Pattern beschrieben, implementieren die Klassen ein Interface, welches die Methoden definiert, die über das Netzwerk aufgerufen werden können.

`C++` Implementierung

Da es in `C++` an sich keine Interfaces gibt, wird eine abstrakte Klasse mit dem Namen `AbstractClass` verwendet. Die Methoden der Klasse werden mit dem `virtual`-Schlüsselwort gekennzeichnet. Dadurch können sie von einer abgeleiteten Klasse überschrieben werden. Mit dem setzen auf 0 werden daraus „pure-virtual functions“, die überschrieben werden **müssen**.

```
class AbstractClass {
  public:
  virtual ~AbstractClass() { }
  virtual void go(int i) = 0;
  virtual bool drink() = 0;
  virtual int eat() = 0;
};
```

Die Methodendeklarationen können nach Belieben geändert werden. Der Name der Klasse und der Destruktor müssen jedoch wie vorgegeben verwendet werden.

2.1.2 Server-Objekt / `Person`-Klasse

In dem Klassendiagramm wird für das Server-Objekt als Beispiel die `Person`-Klasse verwendet. Diese Klasse enthält die eigentliche Logik, die über das Netzwerk aufgerufen werden soll. Sie erbt ebenfalls von der abstrakten Klasse, kann aber vom Benutzer frei gestaltet werden, solange sie die vorgegebenen Methoden implementiert.

2.1.3 PersonStub

Die Client-Stub-Klasse erbt von der abstrakten Klasse. Die Implementierungen der virtuellen Funktionen, senden einen Funktionsaufruf an die Skeleton-Klasse des Servers. Den Rückgabewert oder die Exception erhält der Client-Stub in serialisierter Form wieder vom Skeleton.

Der `PersonStub` ist der Stellvertreter für die `Person`-Klasse. Er ist das Equivalent zum Client-Stub.

Da der Client-Stub von `AbstractClass` abgeleitet wird, müssen die Funktionen überschrieben werden. Der `PersonStub` ist dafür da, um den Funktionsaufruf samt Parameter an den `RemoteFunctionCaller` zu übergeben. Um die Verwendung zu vereinfachen, wurden dafür 2 Makros erstellt.

`C++` Implementierung

```
//Zeilenumbrueche nur damit die Zeile nicht zu lange ist
#define __SEND_FUN__(type, ...)
return RemoteFunctionCaller::returnFunctionCall<type>
```

```
    (__func__, __VA_ARGS__);

#define __SEND_VOID_FUN__(...)
RemoteFunctionCaller::sendFunctionCall(__func__, __VA_ARGS__);
```

Es gibt zwei unterschiedliche Makros. Der `__SEND_FUN__`-Makro wird verwendet, wenn es einen Rückgabewert gibt. In dem Fall ist der erste Parameter der Rückgabewert und danach kommen die Parameter. Der Makro ruft das `returnFunctionCall`-Template auf. Dabei wird der Typ als Template-Parameter mitgegeben. Das erste „normale“ Argument ist ein weiterer Makro `__func__`, der den aktuellen Funktionsname als `char[]` einfügt. `__VA_ARGS__` fügt die weiteren Argumente, getrennt mit einem Beistrich, ein.

Bei Funktionen ohne Rückgabewert muss der Makro `__SEND_VOID_FUN__` verwendet werden. Dieser Makro ruft die `sendFunctionCall`-Funktion auf und macht dasselbe wie der andere Makro, jedoch muss kein Typ angegeben werden.

2.1.4 Der `RemoteFunctionCaller`

Diese Klasse ist für die Serialisierung, das Senden des Funktionsaufrufes und die Deserialisierung des Rückgabewertes zuständig. Wie schon im Teil über den `PersonStub` beschrieben, werden, je nachdem ob es einen Rückgabewert gibt oder nicht, entweder die `returnFunctionCall`- oder die `sendFunctionCall`-Funktion aufgerufen.

`returnFunctionCall`

Beide davon sind Funktionstemplates. Im Gegensatz zur `sendFunctionCall`-Methode, muss jedoch bei dieser Funktion, der Rückgabewert angegeben werden. Dieser wird durch den `typename T` bestimmt und wird als Template-Argument in den spitzen Klammern übergeben. Die Parameter bestehen zuerst aus dem Namen der Funktion, die aufgerufen werden soll. Außerdem wird ein `Parameter-Pack` verwendet. Dadurch kann eine variable Anzahl an Argumenten mit unterschiedlichen Typen entgegengenommen werden.

```
template<typename T, typename... Tail>
T returnFunctionCall(std::string name, Tail... tail);
```

`sendFunctionCall`

```
template<typename... Tail>
void sendFunctionCall(std::string name, Tail... tail);
```

`sendFunctionCall` ist sehr ähnlich zur anderen Funktion. Es wird ebenfalls ein `Parameter-Pack` verwendet, jedoch kann kein Return-Typ angegeben werden und es wird nichts zurückgegeben.

Serialisierung

Die Serialisierung der Parameter erfolgt über die `JSON for modern C++`-Bibliothek[1]. Außerdem wird wieder ein variadisches Template verwendet. Das `Parameter-Pack`

[1] `https://github.com/nlohmann/json`

wird an die convertParametersToJson-Methode weitergegeben.

```
template<typename T>
void convertParametersToJson(nlohmann::json& j, int& i, T last) {
    i++;
    j[std::to_string(i)] = last;
}


template<typename T, typename... Tail>
void convertParametersToJson(nlohmann::json& j, int& i,
  T head, Tail... tail) {
    i++;
    j[std::to_string(i)] = head;
    convertParametersToJson(j, i, tail...);
}
```

Der erste Parameter ist eine Referenz auf ein Json-Objekt. Der zweite ist ein Integer, der die Stelle des Parameters bestimmt. Die Methode wird so lange rekursiv aufgerufen, bis nur mehr ein Parameter übrig ist. Dann wird mit einer überladenen Funktionen der letzte Parameter in das Json-Objekt eingefügt.

Der, durch den __func__-Makro serialisierte, Name der Funktion und das Json-Objekt mit den Parametern wird dann an die handleFunctionCall-Funktion weitergegeben. Diese kümmert sich um die Übertragung des Funktionsaufrufes und Empfangen des Rückgabewertes/der Exception. Die Kommunikation wird im Abschnitt 2.2 noch genauer beschrieben.

2.1.5 Skeleton-Klasse

Diese Klasse ist für Deserialisierung des Funktionsaufrufes und Serialisierung des Rückgabewertes oder der Exception zuständig. Sie hat eine Instanz rmi_object einer AbstractClass. Die Skeleton-Klasse ruft die entsprechenden Funktionen an dieser Instanz auf, wenn ein Funktionsaufruf von einem Client ankommt. Dazu müssen die Aufrufe deserialisiert werden. Das geschieht folgendermaßen:

Deserialisierung

Für die Deserialisierung werden wieder Makros eingesetzt. Sie werden in der skeleton.h-Datei definiert.

```
//Zeilenumbrueche damit die Zeile nicht zu lange ist
#define __ARGUMENT__(type, place) par[#place].get<type>()

#define __FUNCTION__(name, ...)
if (functionName == #name) {
  service.incrementCounter(#name);
  j["returnValue"] = rmi_object->name(__VA_ARGS__);
}

#define __VOID_FUNCTION__(name, ...)
if (functionName == #name) {
  service.incrementCounter(#name);
```

```
  rmi_object->name(__VA_ARGS__);
}

#define __END__ return j.dump();
```

Es wird wieder zwischen Funktionen mit und ohne Rückgabewerten unterschieden. Jeder Parameter muss mit dem `__PARAMETER__`-Makro angegeben werden. Dabei muss der Typ und die Position des Parameters angegeben werden. Das ist notwendig, damit der Parameter an der richtigen Stelle und mit dem richtigen Datentypen aus dem Json-Objekt ausgelesen werden kann. Nachdem die Funktion ausgeführt wurde, wird ein neues Json-Objekt mit (oder ohne) dem Rückgabewert als String zurückgegeben.

In der `abstractMethods`-Datei wird die `callFunction` mit den Makros definiert. Eine Beispielangabe ist im Abschnitt 3.1 zu finden.

2.2 Kommunikation mit Protokoll Buffers

Die Kommunikation zwischen dem `RemoteFunctionCaller` und dem `Skeleton` basiert auf Google Protokoll Buffers.

2.2.1 Protokoll Buffer Definitionen

Im Rahmen des Projekts werden 2 unterschiedliche Protokoll Buffer verwendet. Der erste ist für den Funktionsaufruf.

Funktionsaufruf

```
syntax = "proto3";

message FunctionCall {
  string name = 1;
  optional string json_arguments = 2;
}
```

Im der `FunctionCall`-Nachricht wird der Name der Funktion gespeichert und die Argumente als Json-String. Die Argumente sind logischerweise als optional gekennzeichnet, da nicht jede Funktion Parameter hat.

Rückgabewert

Bei der `ReturnValue`-Nachricht wird zuerst gespeichert, ob der Funktionsaufruf erfolgreich ausgeführt werden konnte. Dadurch können auch `void`-Funktionen bestätigt werden. Zusätzlich wird entweder ein String mit dem Rückgabewert als Wert im Json-Format gespeichert. Oder es wird die Nachricht einer Exception gespeichert, die geworfen wurde.

```
syntax = "proto3";

message ReturnValue {
  bool success = 1;
  optional string json_value = 2;
```

```
  optional string exception_text = 3;
}
```

Übertragung

Nachdem die Protokoll Buffer erstellt wurden, müssen sie übertragen werden. Die Übertragung wird von dem `RemoteFunctionCaller` und dem `Skeleton` übernommen. Für das Senden gibt es jeweils eine `sendProtoBuffer`-Funktion. Diese legt einen `asio::streambuf`, der für die Kommunikation verwendet wird. Da das Ende eines Protokoll Buffers nicht markiert wird, muss der Empfänger die Länge wissen. Darum werden zuerst 4 Bytes im `streambuf` reserviert, welche die Länge (`ByteSizeLong`) speichern. Nachdem diese gesendet werden, wird der Protokoll Buffer zu einem `ostream` serialisiert und mit dem Socket versendet.

```
//Verkuerzte Version der Serialisierung im RemoteFunctionCaller
asio::streambuf buf;
buf.prepare(4);
std::ostream os(&buf);
uint32_t protobufLength = f->ByteSizeLong();
os << protobufLength;
buf.commit(4 - buf.size());
size_t serializeSuccessful = f->SerializeToOstream(&os);
asio::write(sock, buf.data(), ec);
```

Am anderen Ende der Verbindung werden dann zuerst die 4 Bytes mit der Länge n gelesen. Danach werden die n Bytes gelesen, und von einem `istream` in ein Objekt deserialisiert.

```
//Deserialisierung im Skeleton
read(sock, buf.prepare(4), ec);
...
asio::read(sock, buf.prepare(protobufLength), ec);
...
functionCall->ParseFromIstream(&is)
```

2.3 Exceptions

Es kann natürlich auch sein, dass die Funktion am Server gar keinen Rückgabewert zurückliefert, sondern eine Exception wirft. In diesem Fall muss diese auch über das Netzwerk zurückgesendet werden. Wie schon im Abschnitt 2.2.1 beschrieben, gibt es ein optionales Feld für die Exception-Nachricht. Die `callFunction`-Methode wird in einem `try/catch`-Block ausgeführt. Wenn eine Exception geworfen wird, wird das `exception_text`-Feld mit der Exception-Nachricht gesetzt.

```
try {
  std::string s{callFunction(d->name(), j)};
  returnValue->set_json_value(s);
} catch (const std::exception& ex) {
  spdlog::info("Funktion " + d->name() +
```

```
    "␣hat␣eine␣Exception␣geworfen:␣" + ex.what());
  returnValue->set_exception_text(ex.what());
}
```

Wenn der Protokoll Buffer am Client ankommt, wird mit der `has_exception_text()`-Methode überprüft, ob das Exception-Feld gesetzt ist. Wenn ja, wird eine `rmi_user_error` geworfen, welche die Nachricht des Feldes enthält.

```
//r => ReturnValue
if (r->has_exception_text()) {
  spdlog::info("Ein␣Fehler␣wurde␣geworfen:␣" +
    r->exception_text());
  const std::string s{r->exception_text()};
  delete r;
  throw rmi_user_error(s);
}
```

2.4 Fehlerbehandlung

Die Fehlerbehandlung basiert größtenteils auf Fehlercodes (`asio::error_code`). Nach einer Operation, die einen Fehler erzeugen kann, wird überprüft, ob der Wert des Fehlercodes 0 ist. Wenn nicht wird eine Fehlermeldung mit `spdlog` ausgegeben und die Funktion bricht ab. Im Folgenden Codestück wird eine beispielhafte Fehlerbehandlung in der `handleFunctionCall`-Funktion gezeigt:

```
sock.connect(server_endpoint, ec);
if (eclog::error("Verbindung␣zu␣" +
  server_endpoint.address().to_string()\
  + "␣konnte␣nicht␣aufgebaut␣werden", sock, ec))
return nullptr;
```

Im `error_handler.h` sind (im Namespace `eclog`) Funktionen definiert, welche die Fehlerbehandlung vereinfachen. Wenn der Fehlercode gesetzt ist, wird die Fehlermeldung ausgegeben, und es wird `true` zurückgeliefert. In diesem Fall schließt die Überladung der Funktion sogar den Socket. Danach wird die Funktion abgebrochen indem der `nullptr` zurückgegeben wird. Der `error_code ec` ist ein Referenzparameter, dadurch weiß die aufrufende Funktion ebenfalls, dass ein Fehler aufgetreten ist.

Wenn ein Fehler bei der Übertragung auftritt, wird der Fehlercode solange propagiert, bis er bei der `returnFunctionCall/sendFunctionCall`-Funktion ankommt. Dort wird die `rmi_system_error`-Exception geworfen. Dadurch wird dem Benutzer mitgeteilt, dass der Aufruf fehlgeschlagen ist.

2.5 gPRC

Das RMI-System wurde durch einen Server erweitert, über den Statistiken abgefragt werden. Dieser wurde mittels gRPC implementiert.

2.5.1 Proto-Definition

Der RPC-Service wird via Protokoll Buffer in der `statistics_server.proto`-Datei folgendermaßen definiert.

```
service StatisticsManager {
  rpc GetStatistics (StatsRequest) returns (StatsReply) {}
  rpc GetFunctionNames (google.protobuf.Empty)
        returns (FunctionNamesReply) {}
}

message StatsRequest {
  string function_name = 1;
}

message StatsReply {
  int32 counter = 1;
}

message FunctionNamesReply {
  string names = 1;
}
```

Es wird ein StatisticsManager-Service definiert, der zwei Funktionen hat. GetStatistics hat den Parameter StatsRequest, der den Namen einer Funktion enthält. Die Funktion liefert die Anzahl der Aufrufe dieser Funktion zurück. Die Funktion GetFunctionNames hat keinen Parameter. In gRPC muss jede Funktion jedoch einen Parameter haben, weswegen hierfür der zur Verfügung gestellte Protokoll Buffer `google.protobuf.Empty` verwendet wird. Der Rückgabewert enthält die Namen aller Funktionen, die über das RMI-System aufgerufen werden können.

2.5.2 Umsetzung

Die durch gRPC generierten Klassen werden durch die Klassen **StatisticsManagerImpl** und **StatisticsManagerClient** erweitert.

StatisticsManagerImpl

Diese Klasse stellt den gRPC-Server dar. Sie erbt von dem gRPC-generierten StatisticsManager::Service und implementiert die Methoden.

```
class StatisticsManagerImpl final : public StatisticsManager::Service {
...
  public:
    grpc::Status GetStatistics(grpc::ServerContext* context,
      const StatsRequest* request,
      StatsReply* reply) override;
    grpc::Status GetFunctionNames(grpc::ServerContext* context,
      const ::google::protobuf::Empty* request,
      FunctionNamesReply* reply) override;
...
```

```
}
```

Die Skeleton-Klasse hat eine Instanz des StatisticsManagerImpl. In einem eigenen Thread wird in der Funktion `startStatisticsManager` ein Server mit dieser Instanz gestartet. Dabei wird der Port 50051 verwendet.

StatisticsManagerClient

Die Klasse ist ein Wrapper für den StatisticsManager::Stub. Sie hat ebenfalls die zwei gRPC-Funktionen, welche den Aufruf an den Stub weiterleiten. Dabei werden die Parameter in den entsprechenden Protokoll Buffer umgewandelt beziehungsweise der Returnwert aus dem Protokoll Buffer ausgelesen und zurückgegeben.

```
class StatisticsManagerClient {
  public:
    StatisticsManagerClient(std::shared_ptr<grpc::Channel> channel)
      : stub_(StatisticsManager::NewStub(channel)) {}
    int GetStatistics(const std::string&);
    std::string GetFunctionNames();
  private:
    std::unique_ptr<StatisticsManager::Stub> stub_;
};
```

Der Client kann diese über den RemoteFunctionCaller (und dadurch auch über den Stub) aufrufen. Der RemoteFunctionCaller hat eine Instanz des StatisticsManagerClient und ebenfalls die Funktionen `GetStatistics` und `GetFunctionNames`, die den Aufruf an die Instanz weiterleiten.

Kapitel 3

Verwendung

In diesem Kapitel wird beschrieben, wie das RMI-System verwendet wird und wie es konfiguriert werden kann.

Die Ordnerstruktur ist folgendermaßen aufgebaut: Im `src`-Ordner befinden sich drei Ordner. Der erste ist der `client`-Ordner. In diesem kann der Client vom Anwender implementiert werden. Der `rmi`-Ordner ist für die `skeleton.cpp` vorgesehen, dieser darf nicht verändert werden. Im `server`-Ordner ist die `abstractMethods`-Datei, welche die serverseitigen Funktionsdeklarationen enthält. In dem Ordner kann außerdem der Server entwickelt werden.

Der `include`-Ordner enthält den Ordner `rmi_user`. In diesem ist die `abstractClass` und die `.h`-Dateien für das Server-Objekt (Person) und den Stub (PersonStub). Der `rmi_system`-Ordner enthält Dateien, die für das System notwendig sind, aber nicht verändert werden dürfen.

3.1 Konfiguration der Methoden

Damit die Funktionen aufgerufen werden können, müssen diese in verschiedenen Orten angegeben werden.

AbstractClass Diese muss in der `abstractClass.h`-Datei „pure-virtual“ Funktionsdefinition enthalten. Beispiel: 2.1.1

Server-Objekt Das Server-Objekt muss von der AbstractClass und dem Skeleton erben und die Funktionen der AbstractClass überschreiben. Es kann sonst frei gestaltet werden. Wichtig ist, dass im Konstruktor der Skeleton-Konstruktor aufgerufen wird mit dem Pointer auf das aktuelle Objekt (`this`). Damit Anfragen des Clients vom Server verarbeitet werden, muss die Funktion `listenToFunctionCalls` des Skeletons aufgerufen werden. Diese Funktion horcht auf Anfragen auf dem **Port 50113**. Im folgenden Beispiel wird sie im Konstruktor von `Person` aufgerufen.

```
class Person : public AbstractClass, public Skeleton {
  public:
    Person() : Skeleton(this) {
      listenToFunctionCalls();
```

```
    }
    ~Person() { };
    void go(int i);
    int eat();
    bool drink();
};
```

Client-Stub Dieser sollte im `rmi_user`-Ordner definiert werden. Die Klasse muss von der AbstractClass und dem RemoteFunctionCaller erben. Dabei muss jede Funktion definiert werden mit dem `__SEND_FUN__` oder `__SEND_VOID_FUN__`-Makro. Bei beiden müssen die Parameter angegeben werden, wobei bei dem ersten Makro noch der Rückgabewert als erstes Argument angegeben werden muss.

```
class PersonStub : public AbstractClass, RemoteFunctionCaller {
  public:
    void go(int i) {
        __SEND_VOID_FUN__(i)
    }

    int eat() {
        __SEND_FUN__(int)
    }

    bool drink() {
        __SEND_FUN__(bool)
    }
};
```

Beim Aufruf senden diese Funktionen eine Anfrage auf den **(TCP-)Port 50113 des lokalen Hosts**.

Außerdem müssen dafür die `abstractClass.h` und `remoteFunctionCaller.hpp`-Dateien inkludiert werden.

Skeleton-Definitionen In der `abstractMethods`-Datei müssen die `__FUNCTION__` beziehungsweise `__VOID_FUNCTION__`-Makros angegeben werden. Das erste Argument ist dabei der Name der Funktion und die restlichen Argumente müssen mit dem `__PARAMETER__`-Makro definiert werden. Dabei ist das erste Argument des Makros der Typ und der zweite die Position. Am Schluss muss noch der `__END__`-Makro aufgerufen werden. Eine beispielhafte Angabe sieht folgendermaßen aus:

```
std::string Skeleton::callFunction(const std::string functionName,
  nlohmann::json par) {
    nlohmann::json j;
    __VOID_FUNCTION__(go, __ARGUMENT__(int, 1))
    __FUNCTION__(drink)
    __FUNCTION__(eat)
    __END__
}
```

3.2 CLI

Das RMI-System selbst kann nicht per CLI gesteuert werden, da der Einstiegspunkt in das Programm vom Anwender bestimmt wird. In der beispielhaften Anwendung wurde jedoch eine Kommandozeilenschnittstelle erstellt. Im der `main.cpp` des Server-Prozesses können folgende Einstellungen getroffen werden:

```
RMI Server
Usage: ./server [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  --loglevel UINT             Log level: 0 = fatal, 1 = error,
                              2 = warning, 3 = info
```

Mit dem `loglevel` wird bestimmt, bis zu welcher Stufe Meldungen ausgegeben werden. Der Standardwert ist dabei 3.

```
RMI Client
Usage: ./client [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  --loglevel UINT             Log level: 0 = fatal, 1 = error,
                              2 = warning, 3 = info
```

Die Schnittstelle des Clients ist im Grunde ident mit der des Servers.