

# Entwicklung eines Remote Method Invocation Systems

Emil Watz

28. Februar 2022

# Inhaltsverzeichnis

<b>1</b>	<b>Anforderungen an das System</b>	<b>2</b>
1.1	Verbindung des Server-Objekts mit dem Client-Objekt . . . . .	2
1.2	Entferntes Aufrufen . . . . .	2
1.2.1	Benutzerdefinierte Typen . . . . .	2
1.3	gRPC-Einsatz . . . . .	2
1.4	Optionale Anforderungen . . . . .	3
<b>2</b>	<b>Umsetzung</b>	<b>4</b>
2.1	Klassendiagramm . . . . .	4
2.1.1	Abstrakte Klasse (AbstractClass) . . . . .	4
2.1.2	Client Stub . . . . .	5
2.1.3	Remote Object . . . . .	5
2.1.4	Skeleton-Klasse am Server . . . . .	5
2.1.5	Serverklasse . . . . .	5
2.1.6	Client . . . . .	5
2.1.7	Naming-Service . . . . .	5
2.2	Implementierung des Klassendiagramms . . . . .	6
2.2.1	Senden des Funktionsaufrufs . . . . .	6

# Kapitel 1

## Anforderungen an das System

### 1.1 Verbindung des Server-Objekts mit dem Client-Objekt

Es können mehrere Server-Objekte erstellt werden, deren Funktionen über das Netzwerk aufrufbar sind. Der Client kann sich mit einem der Server-Objekte verbinden. Es muss daher ein Mechanismus entwickelt werden, der die Auswahl eines Server-Objekts ermöglicht. Diese Auswahl geschieht über einen Namensdienst (siehe ??).

### 1.2 Entferntes Aufrufen

Ein Client soll Funktionen am Server aufrufen können, dieser Funktionsaufruf soll sich aber gleich Verhalten, wie ein lokaler Aufruf. Der Rückgabewert muss also vom Server zurück an den Client gesendet werden. Es können auch Parameter vom Client an den Server mitgegeben werden. Es muss auch eine Lösung für Exceptions geben. Außerdem muss ein Verhalten für den Verbindungsabbruch definiert werden. Weiters muss auch beachtet werden, dass Funktionen überladen werden können.

#### 1.2.1 Benutzerdefinierte Typen

Es können auch, sofern möglich, benutzerdefinierte Typen als Rückgabe- und Parameter verwendet werden, sowohl der Server, als auch der Client müssen natürlich über diesen Typen verfügen. Hierbei muss wieder eine Fehlerbehandlung erfolgen. Die Definition von benutzerdefinierten Typen muss als Protocol Buffer erfolgen!

### 1.3 gRPC-Einsatz

Zusätzlich zu dem RMI-System soll auch noch gRPC eingesetzt werden. Dieser Einsatz wird im Rahmen eines Namensdienstes implementiert. Mit dem Service

können Server-Objekte unter einem bestimmten Namen angemeldet werden. Clients können dann über den Namen bestimmte Objekte ansprechen und deren Funktionen aufrufen.

## **1.4 Optionale Anforderungen**

Eine optionale Anforderung an das System, welche die Bedienung wesentlich erleichtern würde, ist die (teilweise) automatische Generierung des Client-Stubs und des Server-Skeletons.

# Kapitel 2

## Umsetzung

### 2.1 Klassendiagramm

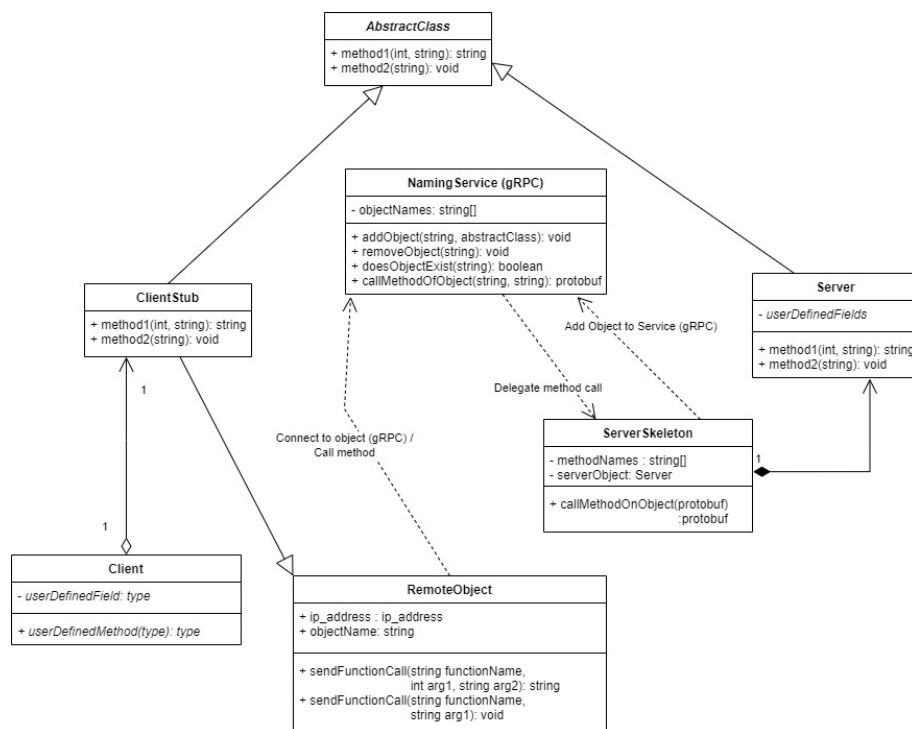


Abbildung 2.1: Klassendiagramm des RMI Systems

#### 2.1.1 Abstrakte Klasse (AbstractClass)

Die meisten Remote Method Invocation Systeme verwenden ein Interface, um die Methoden zu definieren. Da es in C++ keine Interfaces gibt, wird daher eine abstrakte Klasse verwendet, welche eine Anzahl an virtuellen („virtual“)

Funktionen deklariert.

Diese müssen vom Benutzer erstellt werden. Er definiert die Methoden, welche über das Netzwerk aufgerufen werden können. Sowohl der Client, als auch der Server müssen logischerweise die gleiche abstrakte Klasse erweitern. Diese Bedingung muss vom Benutzer selbst erfüllt und überprüft werden, da es keine (einfache) Möglichkeit gibt, die zwei Klassen (übers Netzwerk) in C++ zu vergleichen. Im Klassendiagramm werden die Beispielfunktionen „method1“, „method2“ verwendet

### **2.1.2 Client Stub**

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 am Server.

### **2.1.3 Remote Object**

Das Remote Object enthält ein Template für die Funktion „sendFunctionCall“, welches die Parameter in einen Protokoll Buffer umwandelt und an den Naming-Service sendet.

### **2.1.4 Skeleton-Klasse am Server**

Die Skeleton-Klasse am Server implementiert die benutzerdefinierte abstrakte Klasse. Die Implementierungen der virtuellen Funktionen wandeln die „Funktionsaufruf-Nachrichten“, die vom Client Stub gesendet werden, in „echte“ Funktionsaufrufe der Serverklasse um. Die Rückgabewerte und Exceptions, welche die Funktionen der Serverklasse zurück

### **2.1.5 Serverklasse**

Auch die Serverklasse implementiert die abstrakte Klasse. Die Implementierung der virtuellen Funktionen enthält in dieser Klasse die wirkliche Funktionslogik.

### **2.1.6 Client**

Der Client ist eine Klasse die komplett vom Benutzer konfiguriert werden kann. Zur Verwendung des Remote Method Invocation System benötigt er eine Instanz des Client Stubs.

### **2.1.7 Naming-Service**

Der Naming-Service speichert alle Namen der Objekte, die bei dem Service registriert wurden. Er basiert auf gRPC und erweitert dadurch eine gRPC-Klasse. Objekte können mit einem Namen registriert werden, unter dem sie Funktionen bereitstellen können.

## 2.2 Implementierung des Klassendiagramms

Nun fehlen aber noch viele wichtige Implementationsdetails. In den nächsten Abschnitten wird ein Konzept wichtiger Teile des Systems näher beschrieben.

### 2.2.1 Senden des Funktionsaufrufs

Der Funktionsaufruf muss in irgendeiner Form serialisiert werden. Als Basis bietet sich ein Protokoll Buffer an. Dieser speichert:

1. Den Namen des Objekts, dessen Methode aufgerufen wird
2. Den Namen der Methode, die aufgerufen wird
3. Die Parameter, welche mitgeliefert werden

#### Serialisierung

Der letzte Punkt ist bei der Implementierung der schwierigste, da die Anzahl und die Typen der Parameter nicht fix sind. Die Parameter werden daher als JSON-String gespeichert. Die .proto Datei sind danach folgendermaßen aus:

---

```
syntax = "proto3";

package examplePkg;

message functionCall {
    required string objectName = 1;
    required string functionName = 1;

    optional string parametersInJson = 1;
}
```

---

#### Senden der Daten

Das Erstellen einer Funktion zum Senden der Daten ist wiederum kompliziert, weil die Parameteranzahl variabel ist und der Funktionsname in einen String verwandelt werden muss. Umgesetzt wird das mit einer Superklasse, die jeder Client-Stub erweitern muss. Diese Klasse hat eine Template-Funktion, deren Parametertypen variabel sind und dessen Parameteranzahl beliebig wählbar ist. Der Funktionsname kann durch den Makro `__func__` als „const char array“ übergeben werden. Der Code einer Funktion im Client-Stub könnte folgendermaßen aussehen:

---

```
class ClientStub : public RemoteObject {
public:

    void method1(string s, int i) {
        RemoteObject::sendFunctionCall(__func__, s, i);
    }

    ...
};
```

---

**Vereinfachung mithilfe von Makros** Der Funktionsaufruf der `sendFunctionCall` Methode kann weiter mit der Definition eines Makros vereinfacht werden. Ein Beispielmakro ist Folgender:

---

```
// Definition des Makros
#define __SEND_FUNCTION_CALL__(...) return sendFunctionCall(__func__,
    __VA_ARGS__);

...

//Einsatz des Makros in der Funktion
int method1(int arg1, string arg2) {
    __SEND_FUNCTION_CALL__(arg1, arg2)
}
```

---

### Empfangen des Returnwerts

Der Rückgabewert, der vom Server wieder an den Client gesendet wird, muss am Client wieder deserialisiert werden. TODO: Konzept entwickeln...