ENTWURF

ENTWURF zur aufgabe 4 aus der vorlesungsreihe “VERTEILTE SYSTEME”

TEAM 1: MICHAEL STRUTZKE, iGOR arkhipov

Aufgabenaufteilung

Alle Aufgaben wurden gemeinsam entworfen und bearbeitet.

## Quellenangaben

Aufgabenstellung, Vorstellung der Aufgabe in der Vorlesung.

## Bearbeitungszeitraum

|  |  |
| --- | --- |
| Datum | Dauer in Stunden |
| 08.12.16 | 5 |
| 10.12.16 | 4 |
| 11.12.16 | 5 |
| 14.12.16 | 1 |
| 17.12.16 | 10 |
| Gesamt | 25 |

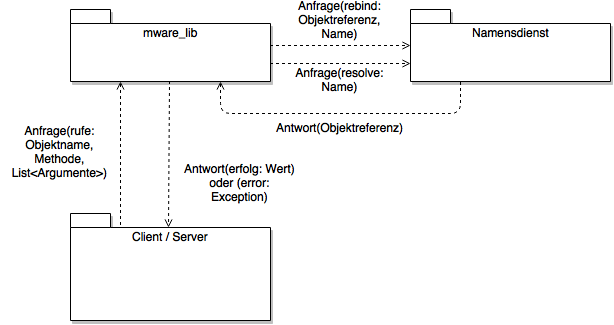
## Aktueller Stand

Der Entwurf ist fertig. Die Implementation ist umgesetzt.

## Änderungen im Entwurf

Der Entwurf wurde erweitert und enthält detaillierte Komponentenbeschreibung.

## Details zum Entwurf:



Es soll eine Middleware mit dazugehörigem Namensdienst implementiert werden, die es ermöglichen, Methoden auf entfernten Objekten auszuführen (sogenannte „Remote Methode Invocation“ Technique). Die Umsetzung erfolgt in der Programmiersprache Java.

Dem Client stehen die Original-Klassen der entfernten Methoden nicht zur Verfügung. Lediglich die Interface-Klassen, die mit einem mitgelieferten IDL-Compiler erzeugt werden, müssen auf dem Client-Rechner vorhanden sein.

Der IDL-Compiler erzeugt aus „.idl“-Dateien Java-Klassen, die die Methoden der Server-Klassen simulieren (d.h. als Schnittstellen dienen, da die Klassen in der Anwendung von diesen generierten Klassen abgeleitet werden). Die Return-Values dieser Methoden sind die Return-Values eines Referenz-Objekts auf einem entfernten Server.

Die Middleware wird von einem „Calculator“ Programm (als Beispielanwendung) gebraucht.

Das gesamte Software-Paket beinhaltet folgende Komponenten:

* einen IDL-Compiler („idl\_compiler.jar“)
* einen Namensdienst („name\_service.jar“)
* eine Middleware („mware\_lib.jar”)
* einen Beispiel-Server (AppServer) mit einer Applikation „Calculator“ („math\_ops.jar“)
* ein Package mit Interfaces der entfernten Klassen („accessor\_one.jar“)
* einen Beispiel-Client (Client), der einen entfernten Methoden-Aufruf mit dem Calculator-Objekt macht („client.jar“)

Alle 3 Anwendungen (Namensdienst, AppServer und Client) können auf unterschiedlichen Computern laufen.

Auf den AppServern und Clients müssen die Dateien mware\_lib.jar und accessor\_one.jar vorhanden sein.

Mit der Datei "idl\_compiler.jar" werden die Interfaces für die in der Datei "idl.files" angegebenen idl-Dateien erzeugt.

**Nameservice**

Der Namensdienst bietet einen Socket an, an dem er die Anfragen „rebind“ und „resolve“ entgegennimmt.

Mit jeder Anfrage wird ein neuer Thread (Connection-Handler) gestartet, der die eingehende Anfrage bearbeitet.

Ist die Anfrage abgearbeitet oder sollte eine vordefinierte Zeit (Timeout) vergangen sein, beendet sich dieser Thread.

Bei einer „rebind“-Anfrage speichert der Namensdienst sich die Kombination aus Namen und Server. Wobei ein Server durch die Kombination aus IP-Adresse und Port definiert ist. Den Port, über den das Objekt erreichbar ist, übermittelt der Server in der Nachricht.

War eine Bindung unter dem angegebenen Namen schon gespeichert, wird die alte Referenz durch die neue ersetzt (überschrieben).

Der Namensdienst setzt voraus, dass der AppServer einen Socket auf dem angegebenen Port bereitstellt und das Objekt selbst wieder über den Namen auflösen kann, da ansonsten das Referenzobjekt nutzlos ist.

Erhält der Nameservice eine „resolve“-Anfrage, antwortet er mit der Kombination aus Objektnamen, IP und Port des AppServers.

Klassen:

* **interface** NameServiceServer **extends** Runnable
  + **void** addReference(String name, AppHost appHost);
  + **void** removeReference(String name);
  + AppHost getReferenceObject(String name);
* **class** NameServiceServerImpl **implements** NameServiceServer, Runnable
  + **public static void** main(String[] args)
  + … (weitere Methoden ergeben sich aus “implements …”
* **class** AppHost
  + InetAddress getRemoteHost()
  + **int** getRemotePort()
* **interface** ConnectionHandler **extends** Runnable
  + **void** sendStream(String msg)
* **class** ConnectionHandlerImpl **implements** ConnectionHandler, Runnable
* **class** TimeOut **implements** Runnable

**Middleware**

Diese Komponente ist eine objektorientierte Middleware, die Anwendern die Interaktion mit den entfernten Objekten ermöglicht.

Die Middleware beinhaltet eine andere Implementierung des Namensdienstes. Diese Instanz (NameServiceOB) vermittelt die Anfragen zwischen dem Frontend (Middleware) und dem entfernten Namensdienst.

Außerdem erstellt dieser Namensdienst mit dem ersten „rebind“-Befehl einen Thread (NameServiceStorage), der die Objekte abspeichert und die Methoden-Aufrufe für die gebundenen Objekte abarbeitet. Dieser Thread erzeugt für jede eingehende Anfrage (Methodenaufruf) einen weiteren Thread (Connection-Handler), der ausschließlich diese eine Anfrage bearbeitet.

Bei einem Methodenaufruf holt der Connection-Handler das entsprechende Objekt vom NameServiceStorage und führt mittels „invoke“-Befehl die Methode aus.

Das Ergebnis wird als „Object“ über TCP dem Client übermittelt.

Sollte beim Aufruf der Methode eine Exception auftreten, wird diese ebenfalls übermittelt. Beim Client wird dann durch die Interface-Klasse (z.B. \_*Calculator*ImplBase) eine RuntimeException erzeugt.

Die Kommunikation vom Client mit dem AppServer geschieht mit einer abstrakten Klasse, die vorab mit dem IDL-Compiler erzeugt wird.

In dieser abstrakten Klasse ist eine Implementierung enthalten (inner Class).

Die Klasse „ObjectBroker“ dient dem Client als Einstiegspunkt zur Middleware.

Der Client erzeugt mit „init“ eine Instanz, wodurch ein „NameServiceOB“-Objekt erstellt wird.

Beispiel-Aufruf Server:

ObjectBroker objBroker = ObjectBroker.init(host, port, true);

NameService nameSvc = objBroker.getNameService();

nameSvc.rebind((Object) new Calculator(), "zumsel");

Beispiel-Aufruf Client:

ObjectBroker objBroker = ObjectBroker.init(host, port, true);

NameService nameSvc = objBroker.getNameService();

Object rawObjRef = nameSvc.resolve("zumsel");

\_CalculatorImplBase remoteObj = \_CalculatorImplBase.narrowCast(rawObjRef);

try {

double s = remoteObj.add(1, 567);

}catch(RuntimeException e){...}

Klassen:

* **class** ObjectBroker
  + **static** ObjectBroker init(String serviceHost,  
     **int** listenPort, **boolean** debug)
  + NameService getNameService()
  + **void** shutDown()
* **abstract class** NameService
  + **abstract void** rebind(Object servant, String name);
  + **abstract** Object resolve(String name);
* **class** NameServiceOB **extends** NameService
* **interface** NameServiceStorage **extends** Runnable
  + **void** addObject(String name, Object object);
  + **void** removeObject(String name);
  + Object getObject(String name);
* **class** NameServiceStorageImpl **implements** NameServiceStorage, Runnable
  + **public static void** main(String[] args)
  + …
* **interface** ConnectionHandler **extends** Runnable
  + **void** sendStream(String msg);
* **class** ConnectionHandlerImpl **implements** ConnectionHandler, Runnable
* **class** TimeOut **implements** Runnable
* **class** RemoteMethodCallObject **implements** Serializable

**IDL-Compiler**

Mit dem IDL-Compiler können von lokalen Klassen (die in den „idl“-Dateien beschrieben sind) Remote-Klassen (also Schnittstellen) erzeugt werden, die es ermöglichen, entfernte Aufrufe in der gleichen Art und Weise wie lokale Aufrufe zu tätigen.

Eine „idl“-Datei wird auf einer spezifizierten „Interface Definition Sprache“ geschrieben und erstellt den Entwicklern eine formale Spezifikation der Schnittstellen (Methodensignaturen und Datentypen). Diese Beschreibung wird mit dem vorliegenden IDL-Compiler in äquivalente Schnittstellenbeschreibung der verwendeten Programmiersprache Java umgesetzt. Außerdem wird ein Quellcode generiert, der zu der benutzten ObjectBroker-Implementierung passt.

Notwendig sind lediglich „idl“-Dateien, die den Klassen-Namen, die Methoden-Köpfe mit ihren Parametern und dem Rückgabetyp enthalten.

Wird der Compiler mit einem Parameter (Klassennamen) gestartet, erzeugt er aus dieser Klasse eine Interface-Klasse, die beim AppServer und beim Client vorhanden sein müssen. Wird kein Parameter angegeben, sucht der Parser die Datei „idl.files“ und erzeugt für alle darin enthaltenen Dateinamen die Interface-Klassen.

Der Parser liest „idl“-Dateien ein, die folgende Schlüsselworte enthalten können:

* module
* class
* int
* double
* string

Beispiel:

module math\_ops {

class Calculator {

double add(double a, double b);

string getStr(double a);

};

};

Neben der reinen Übersetzung (hier in Java) des folgenden Beispiels:

**package** math\_ops;  
  
**public abstract class** \_CalculatorImplBase {  
  
 **public abstract double** add(**double** a, **double** b);  
  
 **public abstract** String getStr(**double** a);  
}

muss der Parser zusätzlich eine Methode „narrowCast“ erzeugen, mit der der Client ein Objekt erzeugen kann.

Der Name der abstrakten Klasse setzt sich zusammen aus:

„\_“ + Original-Klassenname + „ImplBase“

Beispiel: \_*Calculator*ImplBase

In der narrowCast-Methode ist eine Implementierung der abstrakten Klasse enthalten, die alle Methoden der Original-Klasse enthält.

Der Unterschied besteht darin, dass die Methoden-Aufrufe nicht die eigentliche Funktion erfüllen, sondern stattdessen den „AppServer“ ansprechen, ihn den Methodenaufruf durchführen zu lassen und das Ergebnis vom Server an den Client zurückgeben. Somit erhält der Client das gleiche Ergebnis, als würde er die Original-Klasse benutzen.

Nachdem der Client den Nameservice mit „resolve“ nach der Referenz eines entfernten Objekt gefragt hat, lässt er sich mit „narrowCast“ und dieser Referenz ein Objekt erzeugen.

In dieser Referenz sind der Name, unter dem das entfernte Objekt registriert ist, die IP-Adresse und der Port des Servers enthalten.

Damit ist es diesem Objekt möglich, die Methode auf dem AppServer aufzurufen.

Wird also z.B. die Methode add(1.0,2.0) aufgerufen, muss die abstrakte Klasse eine Verbindung zum Server aufbauen und den Methodenaufruf (mit Parametern) übergeben. Anschließend wird das Ergebnis an den Client zurückgegeben.

Diese Funktionalität muss der IDL-Compiler als Java-Code in die Interface-Klassen mit „einbauen“.

**Sequenzdiagramm**

**Programmablauf**



Server:

* ObjectBroker initialisieren
* NameService vom ObjectBroker erfragen
* Objekt instanziieren, welches beim Namensdienst registriert werden soll
* Referenz beim NameServiceStorage speichern
* „rebind“ an NameService senden
* beim entfernten Methodenaufruf Objekt vom NameServiceStorage holen
* Methode ausführen
* Ergebnis an Client übermitteln

Client:

* ObjectBroker initialisieren
* NameService vom ObjectBroker erfragen
* Referenz beim NameService erfragen („resolve“)
* Objekt instanziieren
* Methode aufrufen