****

Michael Müller

Aufgabe 4 – Middleware

**Referat**

Referat eingereicht im Rahmen der Vorlesung Verteilte Systeme

im Studiengang Angewandte Informatik (AI)  
am Department Informatik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. C. Klauck

Abgegeben am Datum

**Michael Müller**

Aufgabe 4 - Middleware

# Inhaltsverzeichnis

[Inhaltsverzeichnis 3](#_Toc516578647)

[1. Einleitung 4](#_Toc516578648)

[2. Überblick über die Aufgabenstellung 5](#_Toc516578649)

[3. Die Komponenten 6](#_Toc516578650)

[3.1. mware\_lib 6](#_Toc516578651)

[3.1.1. ObjectBroker 6](#_Toc516578652)

[3.1.2. SocketCommunicator 6](#_Toc516578653)

[3.1.3. SocketHolder 6](#_Toc516578654)

[3.1.4. SkeletonServer 7](#_Toc516578655)

[3.1.5. SkeletonThread 7](#_Toc516578656)

[3.1.6. ObjectReference 7](#_Toc516578657)

[3.2. Compiler 7](#_Toc516578658)

[3.3. Namensdient 8](#_Toc516578659)

[4. Kommunikation 9](#_Toc516578660)

[5. Einbindung neuer Services 10](#_Toc516578661)

[6. (Zukunfts-) Aussichten 11](#_Toc516578662)

[6.1. Compiler 11](#_Toc516578663)

[7. Resümee 12](#_Toc516578664)

[7.1. Vorgehen 12](#_Toc516578665)

[7.2. Designentscheidungen 12](#_Toc516578666)

[8. Quellenverzeichnis 13](#_Toc516578667)

[9. Anhangsverzeichnis 14](#_Toc516578668)

[*10.* Erklärung zur schriftlichen Ausarbeitung des Referates 15](#_Toc516578669)

# Einleitung

Im Sommersemester in Verteilte Systeme ist die Klausur kein extra schriftlicher Test. Stattdessen gibt es ein Referat inklusiver Ausarbeitung (diese hier). Die PVL erstreckt sich dann von Aufgabe 1 bis Aufgabe 3 und Aufgabe 4 wird als Präsentation (Klausurersatzleistung) vorgetragen.

Dabei ist jedoch zu beachten, dass die Präsentation nur begrenzt auf Inhalte eingehen kann, da sie auf 15 Minuten (Maximal) begrenzt ist. Deswegen ist dieses Dokument die komplette Ausarbeitung für Aufgabe 4. Enthalten ist die Dokumentation der Planung, Spezifikation bis hin zur Implementierung von Aufgabe 4.

Zudem zum Abschluss noch einen Blick in die Zukunft und ein Resümee über die erledigte Arbeit mit Aufgabe 4.

# Überblick über die Aufgabenstellung

In dieser Aufgabe soll eine einfache objektorientierte Middleware konzipiert und realisiert werden, mit deren Hilfe Methodenaufrufe eines entfernten Objektes möglich sind.

Abgegeben werden 3 Komponenten:

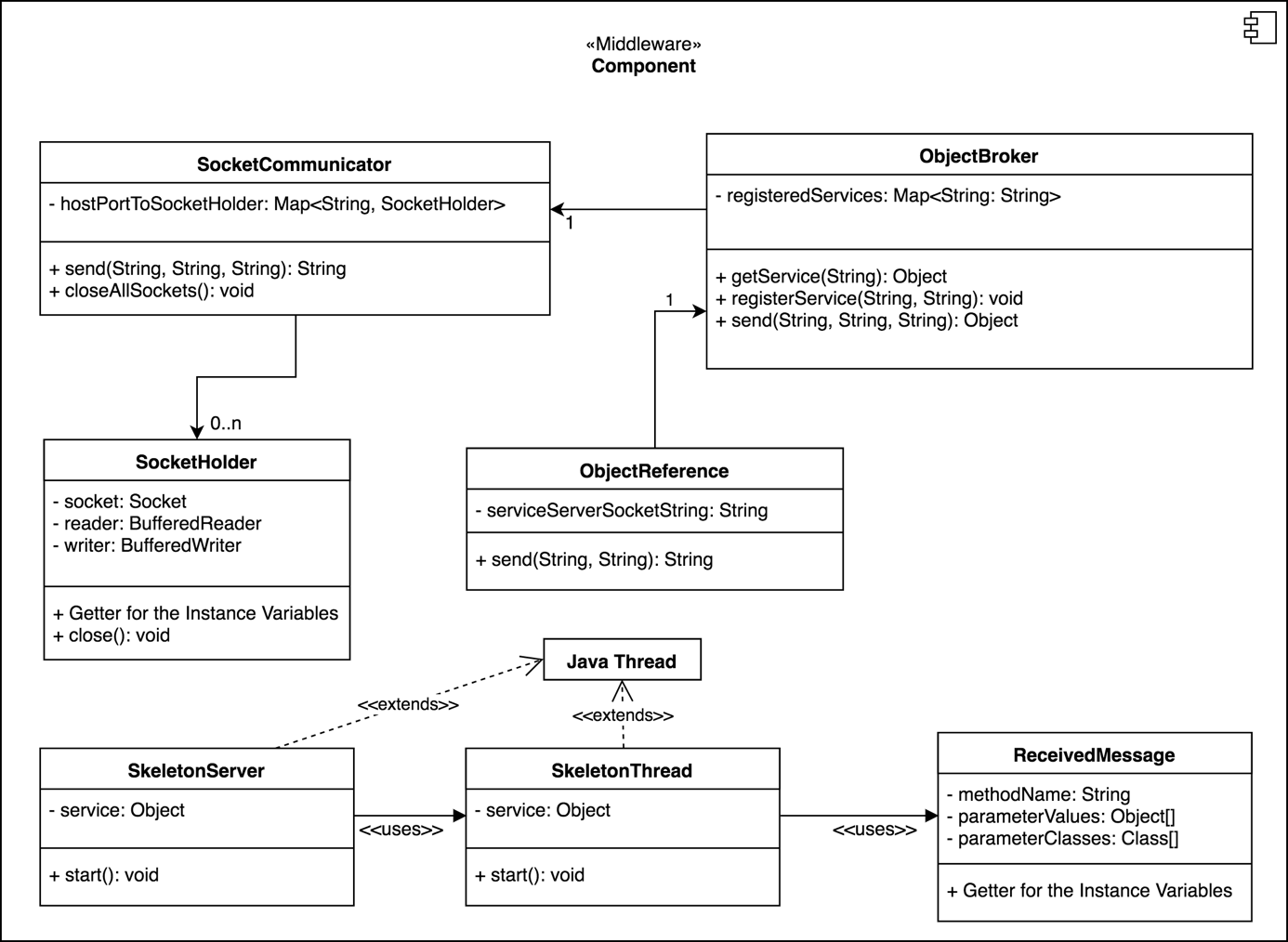
* mware\_lib (Java Package, das eine gewisse Interatkion / Kommunikation zu anderen Middlewares dieser Implementation ermöglicht)
* Compiler (Compiled .idl Dateien zu .java Sourccode)
* Namensdienst (Service für Servicenamen (zu Servicesocket) Auflösung)

Eine Applikation bindet zur Benutzung Ihrer Middleware *mware\_lib* und den mit Ihrem IDL­ Compiler generierten Code ein. Wirft eine Serverapplikation beim Remoteaufruf eine RuntimeException, soll diese an den Aufrufer weitergeleitet werden, d.h. gleicher Exceptiontyp und gleicher Meldungstext

# Die Komponenten

## mware\_lib

Das Java package mware\_lib besteht aus mehreren Klassen:



### ObjectBroker

Ist der Kern der lokalen Middleware. Über sie läuft die Lokale Anmeldung und Lokale Suche nach Services. Zudem läuft über sie die gesamte Beziehung der Middleware auf dem Gerät. Der ObjectBroker hilft zu dem beim Starten eines neuen Services.

### SocketCommunicator

Die wichtigste Hilfsklasse für den ObjectBroker stell der SocketCommunicator dar. Diesen gibt es pro ObjectBroker einmal und er sendet letztendlich die Pakete (Zum Beispiel Anfragen an einen Service Socket). Da Lokal alle Services und Clients über den ObjectBroker indirekt auf einen SocketCommunicator zugreifen gibt es genau ein Socket für einen gewissen Service. So stellen wir sicher, dass es zu einem Service hin (Zum Beispiel NamensDienst) nicht beliebig viele Connections gibt. Zudem können wir alle Sockets bei Herunterfahren korrekt schließen.

### SocketHolder

Ein SocketHolder hilft dem SocketCommunicator in dem sie den Socket und die dazugehörigen Writer und Reader hält. So muss für einen Socket nicht beliebig viele Writer und Reader erstellt werden sondern es gibt genau je einen pro Socket.

### SkeletonServer

Es wird im ObjectBroker (Lokal) hinterlegt zu welchem Servicenamen man welchen Socket benutzen muss. Dabei ist der Socket ein String mit IP, „:“ und Port. Zum Beispiel: „127.0.0.1:15000“. Dieser Socket führt uns dann zu einem SkeletonServer. Dieser hält den entsprechenden Service. Für jeden neuen Socket wird ein SkeletonThread erstellt. Ein SkeletonThread ist solang aktiv wie sein Socket offen ist, dagegen ist der SkeletonServer solang aktiv wie der Prozess noch läuft. Sowohl Server als auch Thread sind extra Java Threads.

### SkeletonThread

SkeletonThreads verwendet die ReceivedMessage Klasse und deren Instanzen. Hierzu wird die eingehende String Nachricht in ReceivedMessage speziell geparst, so dass dann dem Zugreifenden (hier der SkeletonThread) bequem gewisse Daten geliefert werden können. Zum Beispiel der Name der Methode die ausgeführt werden soll, oder die Parameterklassen. So hilft sie dabei im SkeletonThread angenehmer Reflection zu betreiben.

### ObjectReference

Wird über den ObjectBroker ein neuer Service registriert wird gemerkt wie dieser Service heißt und welchen ServerSocketString (IP + „:“ + Port) er hat. Wird nun beim ObjectBroker nach einem Service (anhand eines Namens) gesucht so wird im nicht direkt der ServerSocketString sondern eine Instanz von ObjectReference zurück gegeben. Diese enthält den ServerSocketString und kennt den dazugehörigen ObjectBroker (als Referenz). Diese Instanz wird dann dazu verwendet dann im Service Interface in die „narrowCast“ Methode mitgegeben zu werden.

## Compiler

Hierbei wurde auf den vorgegebenen Dateien aufgebaut. Vielen Dank dafür an Herrn Schulz.

Ich habe hierzu baute ich zusätzlich eine FileCreator Klasse um die Java Dateien und den Sourcecode zu generieren.

Doch zunächst zum Parsen. Dabei werden folgenden Zeichenketten wie folgt erkannt:

|  |  |
| --- | --- |
| IDL­Typ | Entspricht in Java |
| module (keine Schachtelung, 1 Modul pro Datei)  class (nicht als Parameter oder Returnwert, keine Schachtelung)  int  double  string  void  object | package  class  int  double  String  void  Object |

(Erweitert habe ich diese Tabelle um die Returntypen „void“ und „Object“)

Beispiel für eine IDL­Datei und dem daraus vom Compiler generierten Java­Quellcodeschema:

|  |  |
| --- | --- |
| IDL­Datei | Zu generierender Java­Code |
| module math\_ops {  class Calculator {  double add(double a, double b);  string getStr(double a);  };  }; | package math\_ops;  public abstract class **\_**Calculator**ImplBase** ... {  public abstract double add(double a, double b);  public abstract String getStr(double a);  public static **\_**Calculator**ImplBase narrowCast**(Object rawObjectRef)  {...}  ... } |

Es handelt sich also um eine Generierung von Abstrakten Klassen mit Abstrakten Methoden. Ausgenommen der narrowCast Methode die static ist.

Der Parser, Compiler und FileCreator arbeitet so wie im Beispiel zu sehen. Einzige Anmerkung ist, dass der Body der narrowCast Method nicht leer ist, sondern das eingegebene Object verwendet um ein neues zu generieren. Dieses neu generierte Objekt ist von der Klasse <Klassenname> + „Stub“. Es wird davon ausgegangen dass sich der Input hierbei um eine Instanz der oben beschriebenen ObjectReference Klasse handelt.

Zum Beispiel (um an das Beispiel oben anzuknüpfen) kann es wie folgt aussehen:

public static \_CalculatorImplBase narrowCast(Object rawObjectReference){

return new \_CalculatorImplBaseStub(rawObjectReference);

}

Sollte man einen weiteren Typ hinzufügen wollen muss man dies an zwei Stellen berücksichtigen. Zum einen in den Enums von idl\_compiler.IDLCompiler und in der Klasse mware\_lib.ReceviedMessage.

## Namensdient

Der Namensdienst wird verwendet wie ein normaler Service in unserer Middleware Umgebung.

Deswegen wird hier nicht nochmal auf die Einbindung / Implementation eines Services eingegangen.

Sondern kümmern uns um die Spezifika des Namensdienstes.

Dir Arbeit des Namensdienstes ist schnell geklärt. Wie der ObjectBroker lokal, mappt der Namensdienst Servicenamen auf ServiceServerSocketStrings. Jedoch kann diese Zuweisung und Auflsung global genutzt werden.

Wenn man also die IP und Port des SkeletonServers (quasi den ServiceServerSocketString) hat in dem der NameService läuft, so kann man global prüfen ob ein besimmter Service (anhand seines Namens) registriert ist oder nicht.

Man kann auch neue Services dort anmelden. Gleicht der neue Servicename einem bekannten Servicenamen, so wird der alte ServiceServerSocketString mit dem neuen überschrieben. Mehr Verantwortung hat der Namensdienst nicht.

Um diese Arbeit zu verrichten hat der Server des Namensdienstes eine Map, diese bildet wie beschrieben String (Servicenamen) auf String (ServiceServerSocketString) ab.

Und bietet wie in der .idl Datei spezifiziert 2 Methoden:

* rebind(String servicename, String serviceServerSocketString): void
* resolve(String servicename): String serviceServerSocketString

(ist null wenn Name nicht bekannt)

# Kommunikation

So läuft die Kommunikation im Normallfall (wie in 3.1 Beschrieben):

Clientprogramm -> ServiceStub -> ObjectReference -> ObjectBroker -> SocketCommunicator -> SkeletonServer -> SkeletonThread (und wieder zurück für die Antwort).

Wobei über die Reader und Writer der Sockets gelesen bzw. geschrieben wird.

Vor allem der vordere Teil inklusive SocketCommunicator sieht sehr umständlich aus und in den Klassen ObjectReference und ObjectBroker wird die gewünschte Nachricht auch einfach nur weitergegeben.

Jedoch stellen wir so sicher, dass alle ServiceStubs eines ObjectBrokers über eine Stelle / einen SocketCommunicator zu ihren Servern kommunizieren. Der Vorteil davon ist wie in 3.1. Beschrieben, dass wir alle Sockets eines ObjectBrokers an einem Ort (SPOC) haben.

Nachteil des Ganzen ist, dass man die Methode zum Senden und Empfangen der Antworten als „synchronized“ kennzeichnen muss. Sonst könnte es sein, dass sich zwei Prozesse (die auf denselben Socket zu greifen) sich gegenseitig die Antwort wegfischen. Dies könnte unter Umständen zu einem Flaschenhals führen, sollte ggf. ein Service sehr viel Zeit zu Antwort brauchen.

Ein Socket pro Lokaler Middleware hinzu einem Service heißt auch nur 1 SkeletonThread.

Synchronized der Methoden in den Services?

Sockets gibt es somit nur in den Klassen:

* SocketCommunicator (für alle ServiceStubs einer Middleware (ObjectBroker Instanz))
* SkeletonServer (nur um neue Connection anzunehmen)
* SkeletonThread (um durch sie Nachricht zu bekommen und zu senden)

Die gesendeten Nachrichten haben folgende Unterscheidung und sehen wie folgt aus:

* REQUESTS (String):
  1. <methodenname> +
     + „(„ +
       - <ParameterEinsKlasse> + „ „ + <ParameterEinsWert> + „,“
       - <ParameterZweiKlasse> + „ „ + <ParameterZweiWert> + „,“
       - ….. +
     + „)“
* RESPONSE (String: <Antwort>

Beispiele (anhand des Namensdiestes, zusammenhängend):

* Requests:
  1. „rebind(String calculator,String localhost:15000)“
  2. „resolve(String calculator)“
* Response:
  1. null (Methode hat keinen Rückgabewert)
  2. „localhost:15000“

Auf dieses Format kam ich, da ich Java Reflection verwendet und hierfür Methodenname und Parameterklassen benötigte. Zudem weil dies die Standard Notation für Java Funktionen ist. Also erschien mir das natürlichste und intuitivste für zukünftige Entwickler für die Einbindung Ihrer Services und deren Nachrichten. Die hier beschriebenen Formate sind Service-unabhängig und es wird empfohlen diese einzuhalten. Jedoch könnte man auch in einem Rahmen veränderte Formate senden um spezielleren Aufgaben gerecht zu werden. Dies ist kein für die middleware, da nur die ServiceStubs und Server die Nachrichten schreiben bzw. auswerten.

# Einbindung neuer Services

# (Zukunfts-) Aussichten

## Compiler

Für die Zukunft könnte hier noch überlegt werden will ob man allen kompilierten Klassen direkt ein Feld erzeugt in dem man Die IP und den Port des Servers angeben kann. Diese könnten dann per Default so aussehen:

public static final int SERVER\_PORT = 0;

public static final String SERVER\_IP = „localhost“;

Ist dann bei der Einbindung IP und Port bekannt kann man diese dort eintragen. Der Vorteil ist das, man dann für diese Infos einen passenden Single Point of Control hätte.

Eine weitere Idee wäre die ImplBase Klassen serverseitig ein spezielles Service Interface implementieren zu lassen. Über diese könnte eine Schnittstelle angeboten werden die ermöglicht der Java Reflection aus dem weg zu gehen. In dem die Instanz der ReceivedMessage Klasse dann innerhalb des Services ausgewertet wird (da der seine eigenen Methoden und Inputparameter kennt könnte man so zu den auszuführenden Methoden kommen).

Den Vorteil Reflection nicht zu nutzen zu müssen wird allerdings damit erkauft, dass eine Erweiterung der Funktionalität in einem Service hierbei berücksichtigt werden muss. Inklusive natürlich bei einer Neu Implementierung eines Services. Dies erschwert die Erweiterung ein wenig, schränkt sie aber immerhin nicht ein.

# Resümee

An dieser Stelle blicke ich auf die Arbeit (Planung / Implementierung) zurück und analysiere was Gut & was Schlecht lief.

## Vorgehen

Bevor spezifiziert und vor allem bevor implementiert wurde, wurde allein überlegt. Es wurde viel Überlegt, und ein nicht unerheblicher Teil des initialen Entwurfes ist davon unverändert in diesem Dokument gelandet.

Da ich mit Java noch nicht so viel zu tun hatte, konnte ich gewisse Details (z.B. Wie sende / empfange ich Nachrichten?) nicht abschätzen. Für diese speziellen kleinen Fälle hatte ich dann kleine (<10 Zeilen) Prototypen geschrieben. Den initialen Entwurf hat das nicht sonderlich getroffen, da er wie im Praktikum gelernt, Sprachenunabhängig war.

Aufgrund der starken Verzögerungen mit Aufgabe 3 kam ich erst sehr spät dazu mich um Aufgabe 4 zu kümmern. So blieben mir nur ungefähr 2 Wochen. Ich hatte etwas mehr als eine Woche für den Entwurf geplant und etwas weniger als eine Woche für die Implementation. Gerne hätte ich noch länger geplant, aber dies lies die Zeit nicht zu. Dann hätte ich mich noch mehr um Themen wie Erweiterbarkeit, Wartbarkeit, Tests und Deadlocks gekümmert.

Vor allem bei den Test schmerzt es mir ein wenig, da ich hierfür kaum Zeit hatte. Dafür hatte ich jedoch sehr vorsichtig (ständiges Starten der Applikation) und mit vielen kleinen Prototypen (wie oben erwähnt) implementiert. So dass ich gemachte Fehler schnell lokalisieren konnte.

Die Entwurfsphase lief also in kurzer Zeit ganz gut, kann jedoch nicht wie in den Aufgaben davor, mir sichern sein, dass ich nicht doch vielleicht irgendwo ein kleines Detail übersehen habe. Die Implementation lief wegen des vorsichtigen Vorgehens auch den Umständen entsprechend gut.

Jetzt hoffe ich, dass die Aufregung die Präsentation nicht allzu stark beeinträchtigt.

## Designentscheidungen

Aufgabe ist, doch reduziert eine solche Aufteilung in Module die Komplexität des eigentlichen

konnte.

# Quellenverzeichnis

Hauptsächlich wurde die Aufgabenstellung von Aufgabe 4 verwendet.

|  |  |  |
| --- | --- | --- |
| **Dokumentenreferenz** | **Bezeichnung** | **Verweis** |
| Vorl01 | Vorlesung | -/- |
| Vorl02 | Vorlesungsunterlagen | <https://users.informatik.haw-hamburg.de/~klauck/VerteilteSysteme/VSFolien.zip> (Zugriff : 2016-10-21) |
| ErlDoc | Erlang OTP Dokumentation | <http://erlang.org/doc/search/> |
| Prak01 | Praktikumsaufgabe 2 | <http://users.informatik.haw-hamburg.de/~klauck/VerteilteSysteme/aufg2.html> (Zugriff 2016-10-19 bis 2016-11-13 |

# Anhangsverzeichnis

|  |  |  |
| --- | --- | --- |
| **Dokumentenreferenz** | **Bezeichnung** | **Dateipfad** |
| Abb1 | Lebenszyklus, Sequenzdiagramm | .\img\zustandsdiagramm\_system.png |
| -/- | zuAbb1 Rohdatei | .\etc\zustandsdiagramm\_system.vsdx |
| Abb2 | Initialisierung Kommunikationsdiagramm | .\etc\initialisierungs\_phase.png |
| -/- | Abb2 Rohdatei  (https://www.draw.io) | .\etc\initialisierungs\_phase.xml |

# Erklärung zur schriftlichen Ausarbeitung des Referates

*Hiermit erkläre ich, dass ich diese schriftliche Ausarbeitung meines Referates selbstständig und ohne fremde Hilfe verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe sowie die aus fremden Quellen (dazu zählen auch Internetquellen) direkt oder indirekt übernommenen Gedanken oder Wortlaute als solche kenntlich gemacht habe. Zudem erkläre ich, dass der zugehörige Programmcode von mir selbständig implementiert wurde ohne diesen oder Teile davon von Dritten im Wortlaut oder dem Sinn nach übernommen zu haben. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht..*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, den \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Michael Müller