Team: TEAM 01, Falco Winkler (FW), Daniel Schruhl (DS)

Aufgabenteilung:

- IDL Compiler (FW)
- Namensdienst (DS)
- mware_lib Library (FW,DS)

Quellenangaben:

- Aufgabe 4, 11.06.2017, C. Klauck & H. Schulz:
  http://users.informatik.haw-hamburg.de/ schulz/pub/Verteilte-Systeme/AI5-VSP/Aufgabe4/

Bearbeitungszeitraum:

- 11.06.2017 4 Stunden (DS,FW)
- 12.06.2017 2 Stunden (DS)
- 12.06.2017 3 Stunden (FW)
- 14.06.2017 4 Stunden (DS)
- 14.06.2017 5 Stunden (FW)
- 15.06.2017 5 Stunden (DS)
- 16.06.2017 5 Stunden (DS)

Aktueller Stand:

- IDL Compiler fertig und getestet
- Namensdienst fertig und getestet
- mware_lib Library fertig

Änderung des Entwurfs:

- Keine Änderungen

1 Einführung und Ziele

Es soll eine einfache objektorientierte Middleware entworfen werden, die Methodenaufrufe eines entfernten Objektes ermöglicht.

Zur Orientierung gilt hierbei die CORBA Architektur. Genauer soll hier ein ORB zur Verfügung gestellt werden, der es ermöglicht Methoden von entfernten Objekten aufzurufen.

Zur Abstraktion und Beschreibung der Schnittstellen der Objekte soll eine IDL verwendet werden. Diese IDL wird dann zur Erzeugung von Klassen- und Methodenrümpfen verwendet.

Außerdem beinhaltet der ORB einen Namensdienst, der Objektreferenzen in einem Netz mit Namen finden kann.

Die Middleware an sich soll durch eine Library abstrahiert und verwendbar sein.

1.1 Randbedingungen

Der Namensdienst soll auf einem entfernten Rechner unabhängig von der Middleware Library lauffähig sein. Der Port muss zur Laufzeit einstellbar sein.

Der IDL-Compiler soll in einem Package oder einer `.jar` Datei zur Verfügung gestellt werden. Der Compiler soll folgende IDL Typen unterstützen:

- module (keine Schachtelung, 1 Modul pro Datei)
- class (nicht als Parameter oder Returnwert, keine Schachtelung)
- int
- double
- string

Ein Beispiel:

```
module math_ops {
  class Calculator {
   double add(double a, double b);
   string getStr(double a);
 };
};
```

Die Middleware Library soll in einem Package `mware_lib` zusammengefasst werden.

Wenn eine Serverapplikation während eines entfernten Methodenaufrufes eine RuntimeException wirft, soll diese an den Aufrufer weitergeleitet werden.

Es soll möglich sein, dass zwei oder mehrere Klienten die selbe Objektreferenz zeitgleich nutzen wollen. Das soll innerhalb der Middleware nicht zu Deadlocks führen.

1.2 Kontextbegrenzung

Die Implementierung soll in Java vorliegen.
Die Behebung von Deadlocks in den Anwendungen ist nicht Aufgabe der Middleware.

2 Gesamtsystem

Das Gesamtsystem besteht aus drei Subsystemen, die unabhängig voneinander lauffähig sind. Dabei sind das NameService System und das IDL-Compiler System Applikationen, die über die Kommandozeile gestartet werden können.
Die Middleware Library (mware_lib) ist eine Bibliothek, die in der jeweiligen Applikation, die die Middleware verwenden soll eingebunden werden muss. Die Middleware Library ist also nicht direkt über die Kommandozeile startbar.

2.1 Bausteinsicht

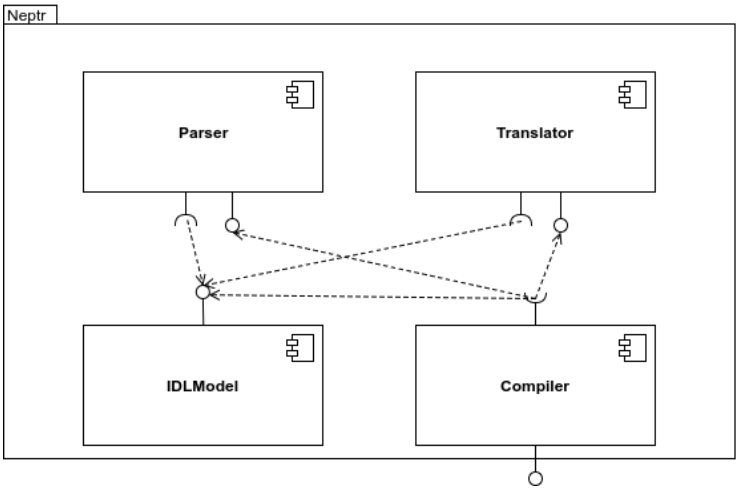

Abbildung 1: Komponentendiagramm des IDL-Compilers (Neptr)

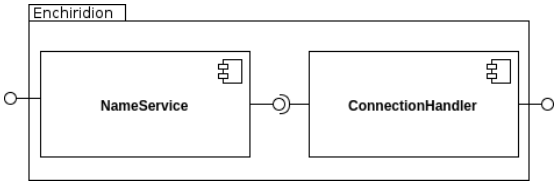

Abbildung 2: Komponentendiagramm des NameServices Server (Enchiridion)

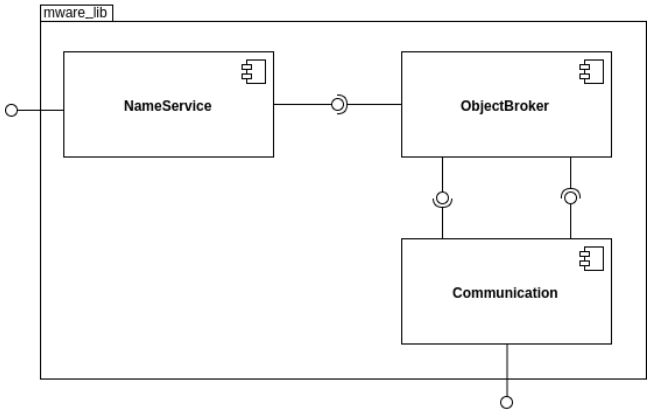

Abbildung 3: Komponentendiagramm der Middleware Library (mware_lib)

2.2 Laufzeitsicht

3 Subsysteme und Komponenten

3.1 NameService (Enchiridion)

3.1.1 Aufgabe und Verantwortung

Der NameService bildet Namen auf Objektreferenzen ab. Er wird verwendet, um Objekte anhand ihres Namens zu finden und anzusprechen und um Objekte anzumelden. Er besteht dabei aus einem Client und einem Server.

3.1.2 Schnittstelle

Die Schnittstellen zwischen dem Client und dem Server tauschen Nachrichten per TCP mittels u.g. Nachrichtenformat (Entwurfsentscheidungen) an gegebenem Port aus.

```
public abstract class NameService {

    /**
     * Registers an Object (servant) at the NameService
     *
     * @param servant Object to register
     * @param name    String representation of the object
     */
    public abstract void rebind(Object servant, String name);

    /**
     * Returns the Object reference from the given servant
     *
     * @param name String of servant
     * @return general Object reference
     */
    public abstract Object resolve(String name);
}

public class NameServiceStarter {
    /**
     * Ueber die main - Methode der Klasse NameServiceStarter
     * kann der NameService (Server) gestartet werden.
     */
    public static void main(String[] args);
}
```

Zu übergebene Argumente sind:

- args[0]: Port auf dem der NameService Server laufen soll

Die erzeugten Binaries können wie folgt ausgeführt werden:

```
./bin/enchiridion <Port>
```

3.1.3 Entwurfsentscheidungen

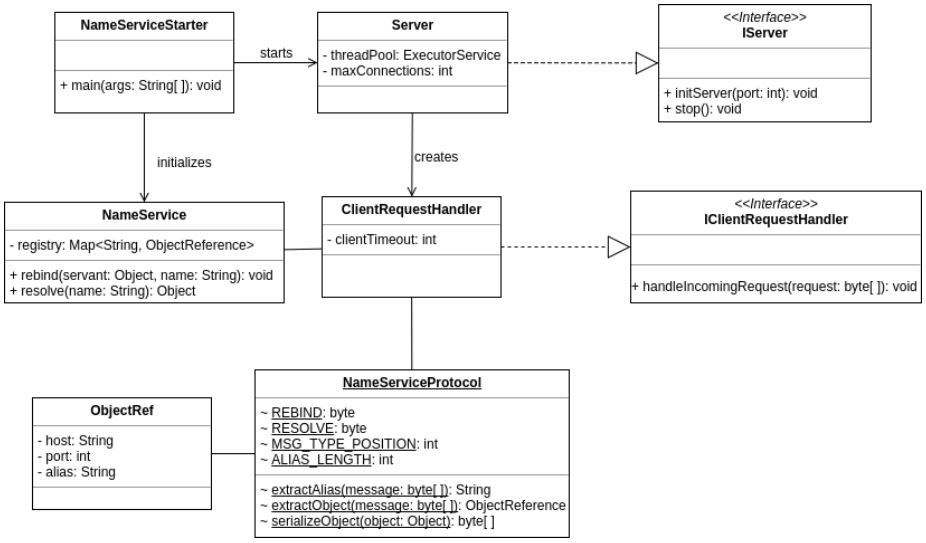

Abbildung 4: Fachliches Daten Modell des NameServices

Der NameService an sich besteht aus einem Server, der Anfragen entgegennehmen kann. Der Client des NameServices ist in der Middleware enthalten.
Der Port, an dem der NameService läuft ist zur Laufzeit einstellbar. Das geschieht über den Startparameter.

Der Nameservice Empfängt Nachrichten im folgenden Format:

- Byte 0: Art des Befehls. (0 = Rebind, 1 = Resolve)
- Byte 1 - 11: Alias für rebind / resolve
- Byte 12 - n: Serialisiertes Referenz Objekt, nur für rebind benötigt

Der NameService besteht aus der Hauptkomponente und der ConnectionHandler Komponente (siehe Abbildung 2).

Die Hauptkomponente enthält den eigentlichen NameService. Der hat eine Registry, die eine Map darstellt, in der Namen auf Objektreferenzen abgebildet werden. Diese Registry ist Threadsafe, da mehrere Clients parallel auf diese zugreifen müssen und diese eventuell verändern können.
Der NameService an sich ist ein Singleton, wodurch sichergestellt wird, dass nur eine zentrale Stelle existiert, die die benötigten Informationen (Registry) enthält.

In der Hauptkomponente ist außerdem ein NameServiceStarter enthalten, der den NameService Server mit dem übergebenen Port startet.

Die ConnectionHandler Komponente ist für die einkommenden Verbindungen zuständig. In der Komponente ist der Server, der durch einen Prozess realisiert wird. Der Prozess lauscht auf dem Socket und nimmt eventuelle Client Anfragen entgegen.
Jede einkommende Client Anfrage startet einen neuen ClientRequestHandler Prozess.

Der ClientRequestHandler Prozess dekodiert die einkommende Nachricht des Clients und versucht diese zu bearbeiten. Mit Hilfe des Protokolls wird entweder ein Resolve oder ein Rebind durchgeführt.
Bei einem Resolve wird die passende Objektreferenz aus der Registry des NameServices übergeben. Wenn keine gefunden wird, wird als Antwort das leere Element an den Client übergeben.
Bei einem Rebind wird die mit übermittelte Objektreferenz mit dem gegebenen Namen in der Registry des NameServices hinterlegt.

Wenn Fehler beim Bearbeiten der an den NameService Server übermittelten Nachrichten auftreten, wird die Nachricht an sich verworfen und die Verbindung des Clients serverseitig abgebaut. Wenn Client Anfragen zu lange brauchen, werden diese abbgebrochen und verworfen (timeout).

3.1.4 Konfigurationsparameter

- Port des NameServices
- Timeout für einkommende Nachrichten in Sekunden (10s)

3.2 IDL Compiler (Neptr)

3.2.1 Aufgabe und Verantwortung

Der IDL Compiler hat die Aufgabe, eine gegebene Modulbeschreibung aus einer `.idl-Datei` im IDL Format einzulesen und daraus Java-Code zu generieren.

Es wird eine Abstrakte Java Klasse generiert (ImplBase bzw. Stub). Diese wird auf serverseitig eingebunden und zu einer konkreten Klasse abgeleitet, die die Methodenrümpfe implementiert.

Darüber hinaus soll der Compiler eine Proxy - Klasse für die Clientseite generieren, welche die Methodenaufrufe auf dem entfernten Server Objekt weiterleitet und ausführt. Das Proxy Objekt wird mit der Abstrakten Klasse (Stub) im Client per narrowCast Methode erstellt.

3.2.2 Schnittstelle

```
public class Compiler {
    /**
     * Ueber die main - Methode der Klasse Compiler kann der Compiler ausgefuehrt werden.
     */
    public static void main(String[] args);
}
```

Zu übergebene Argumente sind:

- args[0]: Pfad zur IDL - Datei
- args[1]: Pfad zum Ordner für die Ausgabedateien

Die erzeugten Binaries können wie folgt ausgeführt werden:

```
./bin/neptr <IDL-Dateipfad> <Ausgabe-Ordner>
```

3.2.3 Entwurfsentscheidungen

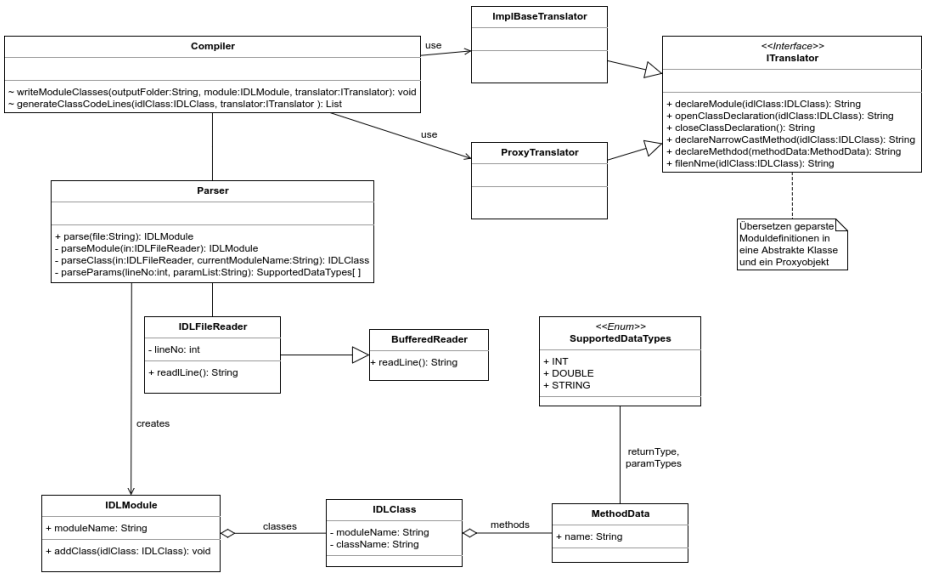

Abbildung 5: Fachliches Daten Modell des IDL-Compilers

Der Compiler besteht aus vier Komponenten (siehe Abbildung 1). In der Hauptkomponente (Compiler) ist der eigentliche Compiler mit seiner Mainmethode. Diese wird nach außen frei gegeben.
Der Compiler benutzt den Parser um eine IDL-Datei zu parsen und daraus ein IDLModule zu erstellen (siehe Abbildung 5). Diese IDLModule beinhaltet alle Klassen. Jede Klasse hat eine Ansammlung von Methoden. Der Compiler schreibt nachdem die

IDL-Datei geparsed wurde alle erkannten Klassen im Modul der IDL-Datei mit Hilfe der Translator an den angegeben Ausgabepfad.

Die Translator haben ein fest definiertes Interface. Dadurch können leichter neue Translator erstellt werden und Translator ausgetauscht werden. Außerdem ist dadurch der Übersetzungsprozess logisch von der Syntax der zu übersetzenden Klasse getrennt. Es gibt insgesamt zwei Translators. Einen um die Stubs (ImplBase) zu erstellen und einen um die Proxy Klassen zu erstellen. Die Translators sind also dafür zuständig die IDLModules zu kompilieren in die gewünschten Java Klassen.

Die Parser Komponente und die IDLModel Komponente wurden mit Hilfe der vorgegeben Klassen aus der Aufgabe erstellt und erweitert.

3.3 mware_lib

3.3.1 Aufgabe und Verantwortung

Die Bibiothek mware_lib stellt die Middleware für entfernte Methodenaufrufe dar. Sie ist vergleichbar mit einem ORB in einer CORBA Umgebung. Sie ist dafür zuständig, über den NameService Objekte über einen Namen anzusprechen und die Methoden der Objekte auf einem entfernten Server aufzurufen. Außerdem müssen Server mit Hilfe der Middleware Objekte im NameService anmelden können.
Die Middleware muss also vom Client und dem Server eingebunden werden.

3.3.2 Schnittstelle

```
public class ObjectBroker {
    /**
     * Main entry point to the middleware.
     *
     * @param serviceHost String host of NameService
     * @param listenPort  int Port of NameService
     * @param debug       boolean indicates extra logging
     * @return ObjectBroker
     */
    public static ObjectBroker init(String serviceHost, int listenPort, boolean debug);

    /**
     * Returns the NameService (proxy Object / Client).
     *
     * @return NameService
     */
    public NameService getNameService();

    /**
     * Initiates the shutting down sequence for the middleware.
```

```
     */
    public void shutDown();
}

/**
 * Interface for communication between ObjectBroker
 */
public class Communication {
    /**
     * Invokes a method for a given Object and returns the result
     *
     * @param ref        ObjectReference of the method
     * @param method     String method name
     * @param args       Object.. arguments of the method
     * @return Object result of the called method
     */
    public Object invoke(ObjectReference ref, String method, Object.. args);

    /**
     * Starts the receiver
     */
    public void startReceiver();

    /**
     * Handles incoming requests, calls function locally and answers request
     *
     * @param clientSocket     Socket from incoming client
     */
    public Runnable handleIncomingRequest(Socket clientSocket);
}
```

3.3.3 Entwurfsentscheidungen

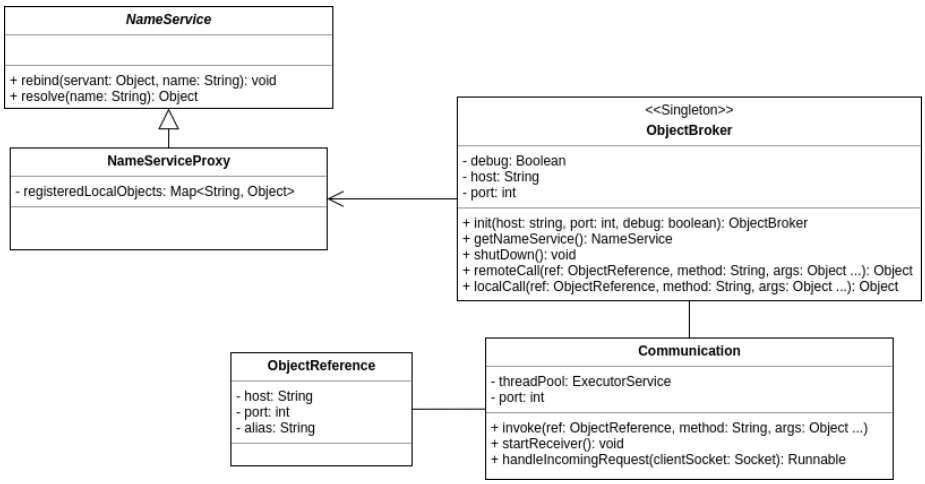

Abbildung 6: Fachliches Daten Modell des Middleware

Der ObjectBroker wird gemäß Vorgabe in einer statischen Methode initialisiert. Außerdem wird er mit dem Singleton Pattern realisiert. Server und Client Anwendungen interagieren über den ObjectBroker mit der Middleware. Der ObjectBroker kann also zum Einen im Server Kontext und zum Anderen im Client Kontext benutzt werden.

In der Communication Komponente der Middleware ist das Communication Modul. Dieses Modul ist für die Kommunikation zwischen verschiedenen Instanzen der Middleware zuständig (GIOP). Die Kommunikation wird über den Port 9002 via TCP geführt.
Dazu wird die Anfrage des entfernten Methodenaufrufes eines Objektes an das Communication Modul der Middleware des Servers gesendet. Das Protokoll schickt serialisierte Objekte. Anfragen für entfernte Methodenaufrufe werden durch ein RemoteCall Objekt realisiert. Die Antworten werden als Objekte an den anfragenden Client zurückgeschickt.
Wenn eine Methode aufgerufen wird, die nicht im entfernten Objekt exisitert, sendet das Communication Modul des Servers als Antwort eine RuntimeException an das CommunicationModul der Middleware des aufrufenden Clients.

Das Communication Modul ist ein nebenläufiger Prozess, der startet, sobald der ObjektBroker initalisiert wird (`startReceiver`). Pro eintreffende Client Anfrage wird ein neuer Prozess gestartet (`handleIncomingRequest`), der die Anfrage bearbeitet und beantwortet. Dazu wird das ObjectBroker Singleton aufgerufen, um die Methode für das ObjectReference lokal auszuführen (`localCall` im ObjectBroker).

Ausgehende Anfragen vom Communication Modul werden durch die Schnittstellenmethode `invoke` ausgeführt. Dabei wird die Anfrage an den Server gesendet, der aus der ObjectReference ersichtlich ist. Die Antwort der Anfrage wird dann als Objekt gelesen und weitergeleitet. Die Anfrage kann dabei mit einer RuntimeException beantwortet werden. Diese wird durch das Communication Modul dann einfach an die aufrufende Komponente weitergeleitet.
Falls keine Antwort kommt (clientTimeout - 10s), wird auch eine RuntimeException an die aufrufende Komponente weitergeleitet.

Der ObjectBroker beinhaltet zusätlich einen Client für den NameService in der NameService Komponente. Dieser sendet Nachrichten über das NameServiceProtocol (siehe Abschnitt 3.1.3). Er empfängt serialisierte ObjectReference Objekte. Er implementiert die clientseitigen Methoden für `resolve` und `rebind`.
Den Hostnamen und den Port des NameService Servers bekommt er durch den ObjectBroker weitergereicht (`init` des ObjectBrokers). Der NameService wird durch den ObjectBroker also erstellt und ist in diesem referenziert. Der NameService beinhaltet zusätzlich eine Map, in der lokale Objekte mit einem Namen hinterlegt sind. Diese lokale Registry wird für einkommende Anfragen eines entfernten Methodenaufrufes verwendet. Mit dieser lokalen Registry können dann die Namen in lokale Objekte aufgelöst werden, die dann zum Ausführen der Methoden verwendet werden. Die lokale Registry ist threadsafe, damit parallele Anfragen bearbeitet werden können.

Der ObjectBroker an sich ist ein Singleton, das eine Referenz zum Communication Modul und zum NameService (Proxy) hält. Ein entfernter Methodenaufruf wird über die vom Compiler erstellten Klassen ausgeführt.
Dafür wird zunächst mit Hilfe des NameService des ObjectBrokers per `resolve` die ObjectReference geholt. Diese wird per `narrowCast` in ihr Stellvertreterobjekt transformiert (Proxy). Dieses Stellvertreterobjekt leitet alle Methodenaufrufe an die `remoteCall` Methode des ObjectBrokers weiter. Dieser delegiert das an sein Communication Modul, das die Antwort an den ObjectBroker zurückgibt. Der ObjectBroker gibt die Antwort dann an das Stellvertreterobjekt weiter. Diese Antwort kann eine RuntimeException sein.
Ein lokaler Methodenaufruf wird vom Communication Modul an den ObjectBroker gesendet. Der ObjectBroker löst den Namen mit seinem NameService in ein lokales Objekt auf und führt die Methode auf diesem aus, wenn sie vorhanden ist. Das Ergebnis der Methode wird an das Communication Modul weitergeleitet. Ansonsten wird eine RuntimeException an das Communication Modul weitergeleitet.

3.3.4 Konfigurationsparameter

- Host, der Hostname des entfernten NameService
- Port, der Port des entfernten NameService
- Debug, boolean konfiguriert, ob debug Nachrichten geloggt werden sollen