Team: 9, David Asmuth, Vladimir Malikov

Aufgabenaufteilung:

1. Vladimir Malikov:
   Entwurf
2. David Asmuth:
   Entwurf

Quellenangaben: http://users.informatik.haw-hamburg.de/~scotty/pub/Verteilte-Systeme/AI5-VSP/Aufgabe3/VSP_Aufgabe3.pdf

Begründung für Codeübernahme: Es wurde kein Code übernommen.

Bearbeitungszeitraum:
19.05.2015: 3 Stunden
20.05.2015: 3 Stunden
21.05.2015: 3 Stunden

Aktueller Stand: Entwurf fertig.

Änderungen im Entwurf:

1. Ziel

In diesem Dokument wird ein einfaches RMI-Framework in Form einer Middleware detailliert entworfen. Diese soll die Aufrufe eines entfernten Objektes ermöglichen. Die Middleware soll nebenläufige Aufrufe erlauben.

2. Rahmenbedingungen

Die Middleware soll unabhängig von der Anwendungsimplementation sein. Es dürfen keine Anwendungsspezifischen Funktionalitäten enthalten sein. Die Fehler der Objekte sollen ohne weiteres an den Aufrufenden geleitet werden. Die Middleware soll aber sinnvolle Fehlermeldungen geben, falls ein Fehler in der Middleware passiert. Das Gesamtsystem wird in einem Netzwerk aus mehreren Computern ausgeführt.

2.1 Verwendete Bibliotheken/externe Module

Für unseren Entwurf verwenden wir keine Externen Module. Grundsätzlich bestehen unter den Packages keine Abhängigkeiten mit Ausnahme von der mware_lib mit den Accessor Paketen.

2.2 Beschreibung des Startvorgangs

Zu Beginn muss der Namensdienst gestartet werden. Anschließend wird die Serversoftware gestartet, welche die Middleware verwendet. In diesem Schritt sollten die Remote-Methoden beim Namensdienst registriert werden.

Anschließend wird eine Client-Software gestartet die von der Middleware Gebrauch macht. Diese Verbindet sich mit dem Namensdienst und erstellt die Stub-Objekte die den Entfernten Methodenaufruf auslösen.

3. Modularisierung

Namensdienst

Verantwortung der Komponente:

Der Namensdienst verwaltet die Referenzen und Verbindungsinformationen auf entfernte Objekte. Man kann Methoden bei ihm registrieren und die entsprechenden Informationen dazu wieder abrufen.

Außensicht:

Die Komponente bietet nach außen hin 2 Schnittstellen an

- Rebind um neue Remote-Methoden zu registrieren.
- Resolve um Remote-Object informationen abzurufen, mit denen im Anschluss Stub-Objekte erzeugt werden können.

Innensicht:

Der Namensdienst besteht intern aus einem WelcomeSocket NameServiceServer.jar, der neue Verbindungen entgegen nimmt und einem NameServiceWorker.jar welcher mit den Clients kommuniziert und ihre Anfragen bearbeitet. Der NamesService verfügt über eine HashMap in der die Informationen über die Remote-Objekte verarbeitet werden.

Entwurfsentscheidungen:

Die Vorgaben der Aufgabenstellung wurden hier berücksichtigt und werden als Entwurfsentscheidung betrachtet.

Schnittstellen zu Nachbarsystemen:

Keine.

Middleware

Verantwortung der Komponente:

Die Middleware ist das Herzstück des RMI-Frameworks. Er wird sowohl auf dem Server als auch auf dem Client durch die Benutzer-Software ausgeführt. Intern verwaltet sich die Middleware über eine HashMap, in der die vom Nutzer vegebenen Namen den entsprechenden Remote-Objekten zugeteilt sind. Er kommuniziert mit:

- Dem Namensdienst um Informationen zu Remote-Objekten zu speichern (Serverseite) bzw. zu lesen (Clientseite).
- Mit sich selber, wobei das Modul auf der Clientseite sich auf die Serverseite verbindet, um Remote-Methodenaufrufe auszulösen.

Außensicht:

Die Komponente bietet nach außen hin Schnittstellen für den Benutzer an, der es ihm erlaubt neue Objekte zu registrieren und somit die Fassade für den NameService darstellt. Der Aufruf erfolgt über die Komponente accessor für den eine Schnittstelle in Form eines TcpSockets bereitgestellt wird der auf das InvokationServer Modul zugreift (siehe dazu auch Accessors).

Innensicht:

Die Komponente besteht aus dem Objectbroker Modul, welches das NameService-Modul sowie den InvokationServer-Modul startet, welcher wiederrum die RMI-Anfragen an den Skelleton weiter leitet.

Entwurfsentscheidungen:

Um zu gewährleisten das auch zwei verschiedene Programme auf dem selben Computer mit der Middleware arbeiten können, wird der Port des InvokationServer zur Laufzeit festgelegt. Die entsprechenden Verbindungsdaten müssen somit dem Client mitgeteilt werden. Dies geschieht über den NameService. Der Invokationserver dient nur als WelcomeSocket. Alle Anfragen werden in einem eigenen Skelleton-Thread bearbeitet um die Nebenläufigkeit zu gewährleisten.

Schnittstellen zu Nachbarsystemen:

Keine.

Accessors

Verantwortung der Komponente:

Die Accessors sind die Stummel der eigentlichen Remote-Objekte.

Die Basisklassen dienen als Interface um die wahren Objekte abzubilden. Im Hintergrund findet jedoch ein Netzwerkzugriff statt. Dennoch sollen sich alle Objekte wie lokal verhalten. So müssen auch Exceptions weiter geleitet werden.

Außensicht:

Für den Benutzer bietet diese Komponente nur die Abstrakten Objekte und deren Methoden, sowie dazu jeweils eine statische Methode, die es erlaubt mit den Daten aus dem Namensdienst neue Stummel zu erzeugen.

Innensicht:

Die Komponente besteht aus den Abstrakten Objekten die als Interface für den Benutzer dienen, sowie für den Benutzer nicht sichtbaren Implementierungen, welche die eigentliche Datenübertragung abwickeln und die Ergebnisse zurück liefern.

Entwurfsentscheidungen:

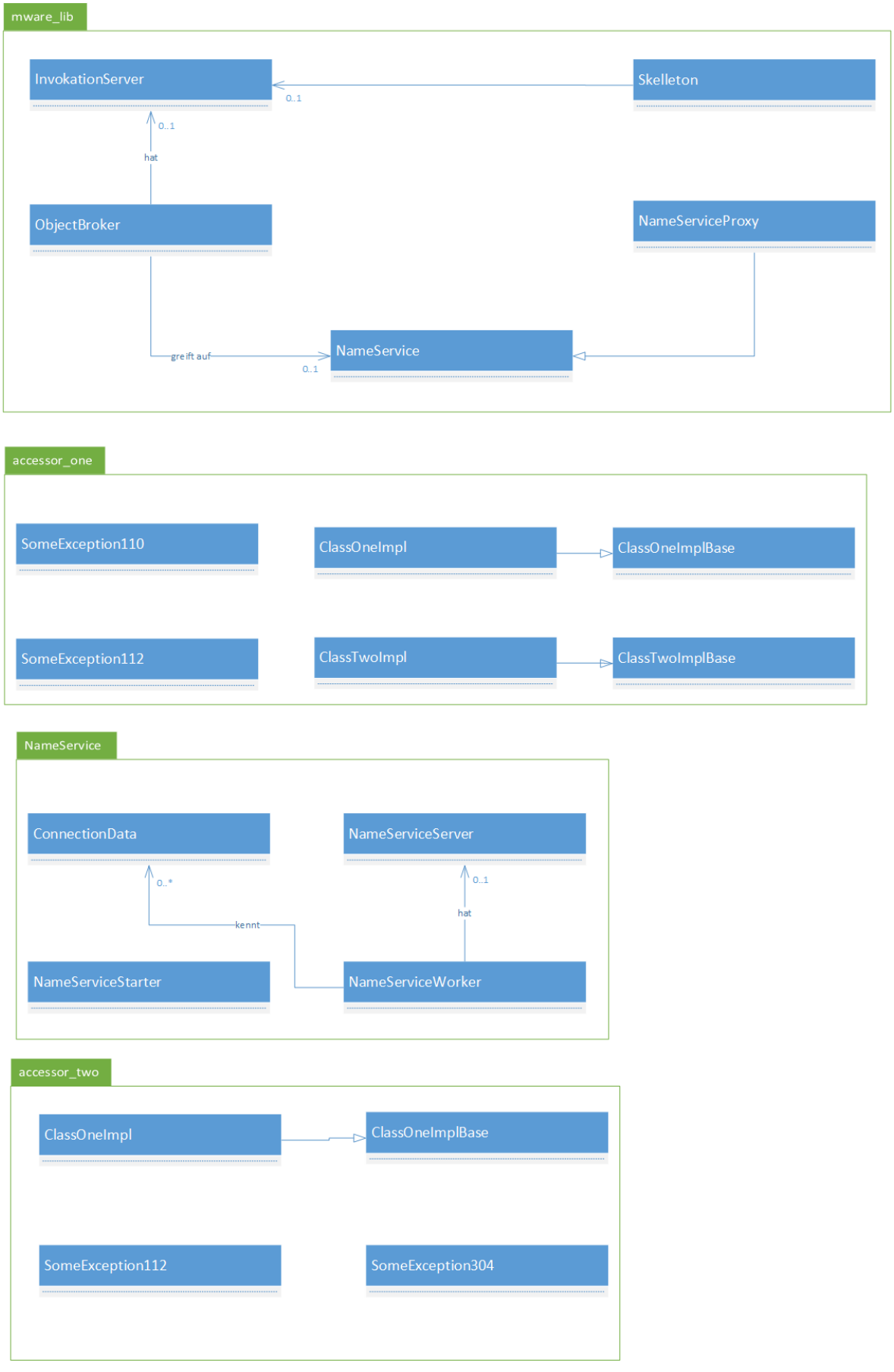
Die Vorgaben der Aufgabenstellung wurden hier berücksichtigt und werden als Entwurfsentscheidung betrachtet.

Schnittstellen zu Nachbarsystemen:

Keine.

4. Klassenbeschreibung

4.1 Klassendiagram



4.2 Modulbeschreibungen:

Im Folgenden werden wichtige Methoden der einzelnen Module kurz beschrieben:

NameService

NameServiceStarter.java

Main():

Definition: Verarbeitet die Start-Parameter und startet den NameServer WelcomeSocket.
Precondition: Der Parameter „Port“ wurde korrekt angegeben.
Postcondition: WelcomeSocketThread erzeugt
Return: void.

NameServiceServer.java

Run():

Definition: Ist ein eigener Thread. Nimmt eingehende Verbindungen entgegen und erzeugt Worker Threads, Regelt die maximalen gleichzeitigen Verbindungen. Initialisiert eine HashMap für die RMI-Einträge, und gibt jedem Worker-Thread eine Referenz darauf.
Precondition: Freier Port.
Postcondition: Keine.
Return: void.

NameServiceWorker.java

Run():

Definition: Ist ein eigener Thread. Bei eingehender „rebind“-Anfrage (siehe Netzwerk Protokoll), wird ein Eintrag in der HashMap geschrieben oder aktualisiert, wobei der Key der übergebene Name ist. Bei einer „resolve“-Anfrage wird aus der HashMap der entsprechende Eintrag ausgegeben. Ist keiner vorhanden, wird „null“ Als String zurückgeliefert.
Precondition: keine.
Postcondition: keine.
Return: void.

mware_lib

ObjectBroker.java

_ctor()

Definition: Erzeugt eine neue Objectbroker Instanz. Hierbei sollte eine HashMap initialisiert werden, welche im NameServiceProxy den Namen dem „Servant“ zuordnet. Initialisiert ebenfalls den InvokationServer (WelcomeSocket für Skeletons), dieser bekommt die HashMap. Initialisiert abschließend den NameServiceProxy, dieser bekommt ebenfalls die HashMap.
Precondition: NameService läuft bereits.
Postcondition: keine.
Return: ObjectBroker Instanz.

NameService():

Definition: Gibt den NameServiceProxy zurück.
Precondition: kein.

Postcondition: keine.
Return: NameService-Interface

ShutDown():
Entfernt alle Einträge aus der Hasmap. Beendet den InvokationServer.
Precondition: ObjectBroker initialisiert.
Postcondition: HashMap geleert. InvokationServer-Thread beendet.
Return: void.

NameServiceProxy.java

Rebind():
Definition: Sendet eine rebind Nachricht (siehe Netzwerkprotokoll) an den NameService und vermerkt „servant“-Referenz in der HashMap mit “name” als Key.
Precondition: Gültige NameService Adresse.
Postcondition: HashMap Eintrag aktualisiert.
Return: void.

Resolve():
Definition: Sendet eine Resolve Anfrage an den NameServer gibt die Antwort als „object“ zurück.
Precondition Gültige NameService Adresse.
Postcondtion: keine.
Return: Die Antwort des NameServers.

InvokationServer.java

_ctor():
Definition: Initialisiert einen neuen WelcomeSocket. Um mehrere Middlewares auf einem Rechner zu erlauben, sollte dieser per Zufall gesetzt werden. Ist er belegt, wird ein anderer Verwendet. Welcher Port verwendet wird, kann über ein getter abgerufen werden.
Precondition: HashMap initialisiert.
Postcondition: keine.
Return: void.

Run():
Definition: Nimmt alle Verbindungen von den Clients entgegen und startet einen eigenen Thread für diese. Limitiert die Anzahl der gleichzeitigen Verbindungen. Alle SkeletonThreads erhalten eine Referenz auf die HashMap mit den Objektreferenzen.
Precondition: Port frei.
Postcondition: keine.
Return: void.

Skelleton.java

Run():
Definition: Eigener Thread. Führt den eigentlichen Methoden-Aufruf aus. Die eingehenden Anfragen werden Zunächst geparsed (siehe Netzwerkprotokoll). Anschließend wird mittels dem Namen die entsprechende Servant-Referenz aus der HashMap rausgesucht und die Parameter in Ihre echten Datentypen konvertiert. Die Aufrufe sollten mit try-catch

eingeschlossen werden um als Antwort auch Exceptions zurückgeben zu können.
Precondition: Gültige Anfrage/Paket.
Postcondtion: Die Methode wurde aufgerufen.
Return: void.

5 Netzwerk Protokoll

Im Folgenden ist der Aufbau und der Zweck der einzelnen Nachricht die in dem System verschickt werden beschrieben. Grundsätzlich sind Parameter durch einen Doppelpunkt getrennt.

NameService ←→ NameServiceProxy

Definition: „rebind:skellton_host:skelleton_port:name“
Verwendung: Anfrage von NameServiceProxy an NameService. Setzt einen neuen Eintrag „name“
Antwort: Keine.

Definition: „resolve:name“
Verwendung: Anfrage vom NameServerProxy an NameService: Fragt die Verbindungsdaten zu einem Name ab.
Antwort: „name:skellton_host:skelleton_port“

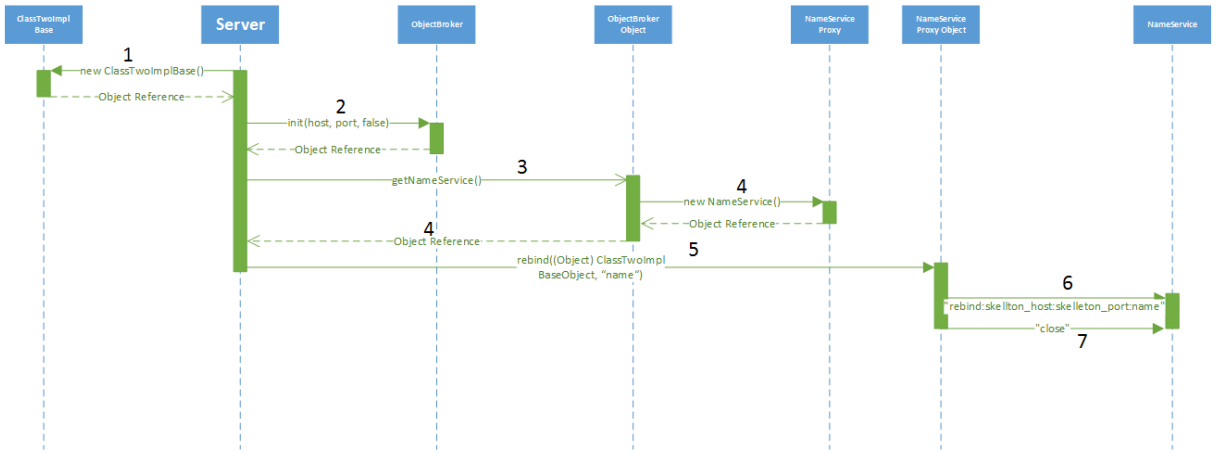
Definition: „close“
Verwendung: Schließt eine Verbindung ordnungsgemäß.
Antwort: Keine.

Acessor(Stub) ←→ Skelleton

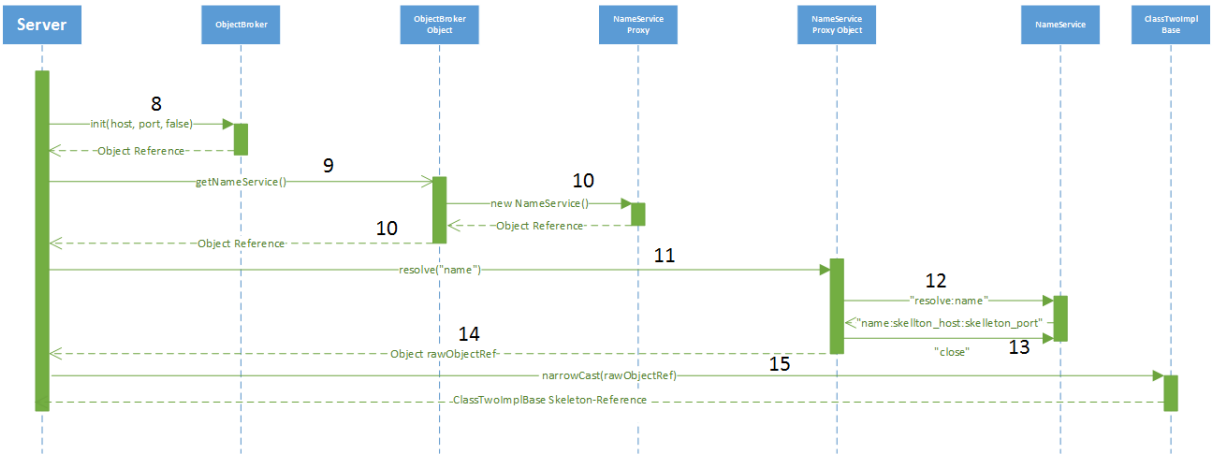
Definition: „:….“
Verwendung: Übertragen von Methodenaufrufen. Anzahl der Paramter abhängig von stub.
Antwort: „result:ergebniss“ / „exception_name:message“

6 Sequenzdiagramm

6.1 Sequenzdiagramm rebind



6.2 Sequenzdiagramm resolve



6.3 Sequenzdiagramm remote call