Hanzehogeschool Groningen
Instituut voor Informatie
en Communicatietechnologie

Thema 4.2I Gedistribueerde applicaties
Practicumopgaven week 1

Inleiding

De opgaven van deze eerste week staan in het teken van het werken met TCP/IP sockets, het gebruiken van TCP/IP sockets voor het op afstand aanroepen van methoden van objecten met behulp van een eigen framework en het bestuderen van bestaande frameworks op het gebied van RPC. Kortom, deze week staat in het teken van de klassieke distributie van applicaties.

Opgave 1 – TCP/IP Sockets in Java

In deze opgave ga je zelf een eenvoudig client/server systeem bouwen. Je gaat een eenvoudige vertaalservice implementeren waarbij een client de betekenis van een woord bij de server kan opvragen en later een zin kan vertalen. De vertaling moet wel in beide richtingen kunnen gaan: de client bepaalt welke taal wordt aangeboden.

a. Maak een lijst van tien woorden in twee talen.

b. Tijdens de theorieles is het protocol van de KnockKnock-applicatie gedemonstreerd. Ontwerp een vergelijkbaar protocol tussen de vertaalclient en -server. Hou ook rekening met het feit dat woorden niet kunnen voorkomen.

c. Implementeer je ontwerp uit de vorige vraag in een client/server oplossing met sockets.
Maak deze multithreaded door bij elk verzoek een aparte thread te starten. Hint: bekijk de java.util package voor handige datastructuren die je kunt gebruiken voor je woordenboek.

d. Breid je protocol uit vraag b) uit door hele zinnen te laten “vertalen”. Deze vertaling moet gewoon woord voor woord gebeuren.

e. Breid de implementatie van vraag c) uit met het protocol uit vraag d).

Opgave 2 – Sockets, RMI en remote methodes

in deze opgave ga je werken aan een Remote Procedure Call framework dat in de verte gebaseerd is op Java RMI. De bedoeling is dat je leert hoe een dergelijk RPC framework conceptueel werkt door code toe te voegen aan een eigen homegrown RPC framework.

Download RMIRPC.zip van Blackboard en pak het uit. Open `HomeGrownRPCBasic`. Start achtereenvolgens `NameServer`, `HelloServer` en `HelloClient`.

a. Hoeveel JVM processen zijn er nu? Laat zien met een screenshot. Licht het aantal toe.

b. Licht de werking van de applicatie toe door de functie van de klassen en interfaces in dit project uit te leggen.

c. Er wordt een bepaald protocol tussen de `HelloStub` en de `HelloSkel` gebruikt. Geef een beschrijving van dit protocol.

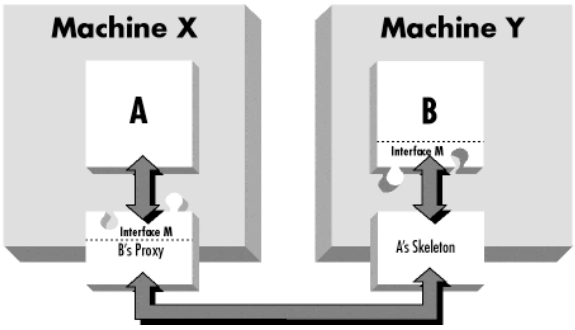
d. Leg uit waarom Base64 *de-* en *encoding* wordt gebruikt bij het versturen van Strings in het protocol van de vorige vraag.

e. Implementeer de drie methoden in `HelloStub` die een `UnsupportedOperationException` gooien.

f. Implementeer de afhandeling voor `sayHello#1#int` in de methode `listen()` van `HelloSkel`.

g. Haal de drie laatste method calls in `HelloClient` uit commentaar en run het project.

Open nu het `RMIHelloStubSkel` project. In dit project zit exact dezelfde functionaliteit als het `HomeGrownRPCBasic` project. Compileer het project. Ga met een (DOS) terminal naar de build/classes directory van dit project. Je laat nu via een speciale RMI compiler (`rmic`) de stubs en skeletons genereren. Voer `rmic -keep -v1.1 nl.hanze.web.rmi.HelloImpl` uit. De compiler genereert nu vier bestanden: een stub en een skel in Java en class formaat. Kopieer de Java bestanden naar de source directory. Blijf in de build/classes directory staan in een terminal. Voer `rmiregistry` uit en start `HelloServer` en `HelloClient`.



h. Wat zijn de overeenkomsten en verschillen tussen dit RMI voorbeeld en het eerdere `HomegrownRPCBasic` voorbeeld?

i. Demonstreer de werking aan de practicumdocent en lever je code en antwoorden op de vragen in via Blackboard.

Opgave 3 – Slimmer en compacter

We willen uiteindelijk ook een soort RMI compiler maken die op basis van een interface definitie (in plaats van een interfaceimplementatie zoals de RMI compiler) stub- en skeletoncode kan genereren. Het blijkt dat het voldoende is om alleen een stub te laten genereren. We willen eerst laten zien dat de standaardinstelling van de RMI compiler het alleen genereren van een stub is.

Open het project `RMIHelloStub` en compileer het project. Ga met een (DOS) terminal naar de build/classes directory van dit project. Voer `rmic -keep nl.hanze.web.rmi.HelloImpl` uit. De compiler genereert nu twee bestanden: een stub in Java en class formaat. Kopieer het Java bestand naar de source directory. Blijf in de build/classes directory staan in een terminal.

a. Voer `rmiregistry` uit. Start `HelloServer` en `HelloClient`. Demonstreer de werking aan de practicumdocent

Blijkbaar heb je geen apart te genereren skeleton nodig. Als je goed kijkt naar de implementatie van `HelloStub` en `HelloSkel` in het `HomegrownRPCBasic` voorbeeld valt je misschien op dat:

- ✓ veel code te maken heeft met het (omslachtig!) inpakken en uitpakken van de aan te roepen methoden en de bijbehorende parameters. Dat moet handiger kunnen.
- ✓ in `HelloSkel` na het uitpakken een specifieke methode van `HelloImpl` wordt aangeroepen. Als er een mogelijkheid zou bestaan om *runtime* op een object een methode aan te kunnen roepen op basis van de naam van de methode als String kun je `HelloSkel` achterwege laten en vervangen door een algemene Skel klasse.

b. Open `Hello_Stub` en `Hello_Skel` in `RMIHelloStubSkel`. Je ziet hier dat gebruik wordt gemaakt van een `ObjectInputStream` en een `ObjectOutputStream`. Leg uit waarom dit een oplossing is voor het eerste punt.

c. Zoek uit wat Java zoal kan versturen via zo'n stream.

d. Open `HomeGrownRPCFull` en bestudeer de code van achtereenvolgens `Stub`, `Skel` en `HelloStub`. Welke java-techniek wordt hierin gebruikt om in runtime aan de hand van een methode-naam als String en eventuele parameters een methode aan te roepen? Waarom is dat een oplossing voor het tweede probleem?

e. Implementeer `AddressBookStub`. Gebruik hierbij het volgende schema (deels af te leiden vanuit de code in `HelloStub`):

- ✓ Bij een void methode hoef je alleen maar invokeSkel aan te roepen.
- ✓ Bij een methode die een object teruggeeft moet je o casten naar het returntype van de functie

(zie bijvoorbeeld sayHello in `HelloStub`)

- ✓ Bij een methode die een primitief type teruggeeft moet je eerst casten naar de bijbehorende wrapperclass en daarna de *Value() methode aanroepen (zie bv ageNextYear in `HelloStub`)

f. De code zal een runtime fout geven. Wat moet je aanvullen aan de klasse Student om het wel te laten werken en waarom? Voer nu de applicatie uit en demonstreer het resultaat aan je practicumdocent.

Opgave 4 - Zelf een StubGenerator maken

Het voorgaande recept voor de implementatie van een stub kun je automatiseren. Open `StubGenerator` en bestudeer de code.

a. Vul wrapperClass in de constructor aan met de acht primitieve types.

b. Vul generateClassLine aan.

c. Vul generatePackageName aan.

d. Implementeer searchMethods.

e. Vul generateMethod aan.

f. Vul saveGeneratedClass aan.

g. Laat AddressBookStub dynamisch genereren door StubGenerator en vergelijk dit met je eigen implementatie. Voer de applicatie uit en demonstreer het resultaat aan je practicumdocent.

Opgave 5 - RMI en WCF

Net als Java heeft ook het .NET framework een eigen implementatie voor het gebruik van remote functionaliteit: Windows Communication Foundation. Schrijf een klein verslagje waarin je uitlegt hoe dit werkt en wat de verschillen en overeenkomsten tussen WCF en RMI zijn.