Praktische Übung 1 zur Vorlesung
“Verteilte Systeme”
ETH Zürich, WS 2002/2003

Prof. Dr. F. Mattern

Ausgabedatum: 8. November 2002
Abgabedatum: 29. November 2002

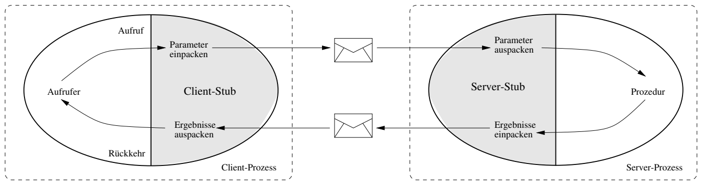
In dieser Aufgabe geht es darum, ein RPC (*Remote Procedure Call*)-System in Java zu konzipieren und zu implementieren.

Einführung

Beim lokalen Prozeduraufruf befinden sich der Aufrufer und die aufgerufene Prozedur im gleichen Prozess. Im Gegensatz dazu versteht man unter Remote Procedure Call (RPC) den Aufruf einer Prozedur über Prozessgrenzen hinweg, d.h. der Aufrufer und die aufgerufene Prozedur befinden sich in verschiedenen Prozessen (dem Client- bzw. Server-Prozess), die entweder auf dem gleichen Rechner oder sogar auf verschiedenen Rechnern ausgeführt werden.

Dabei wird durch die Verwendung eines sogenannten *Client-Stubs* (Stellvertreter im Client) im Prozess des Aufrufers der Anschein erweckt, dass es sich um einen lokalen Prozeduraufruf handelt. Beim Client-Stub handelt es sich im wesentlichen um eine Prozedur mit gleichem Namen und Parametern wie die eigentliche Prozedur, die durch Verschicken geeigneter Nachrichten die Ausführung der entfernten Prozedur veranlasst und Parameter bzw. Ergebnisse des Aufrufs zwischen den beiden Prozessen transferiert.

Ebenso wird durch einen Server-Stub (Stellvertreter im Server) im Prozess der aufgerufenen Prozedur der Anschein erweckt, als befinde sich der Aufrufer im gleichen Prozess. Der Server-Stub ist ein Programm-Stück, das Netzwerknachrichten vom Client-Stub empfängt, die Prozedur mit den so übermittelten Parametern aufruft und die Ergebnisse an den Client-Stub schickt.

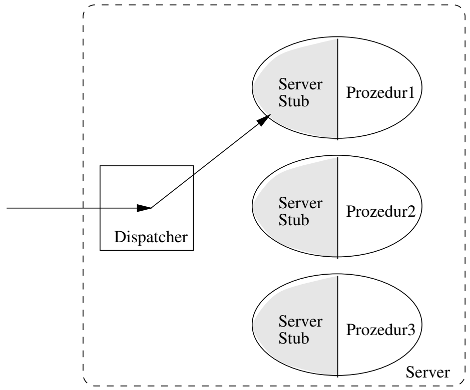


Obenstehende Abbildung verdeutlicht die Abläufe bei einem RPC. Hier im Einzelnen die Schritte:

1. Der Aufrufer ruft den Client-Stub in gewohnter Weise auf
2. Der Client-Stub erzeugt eine Nachricht, die den Namen der aufzurufenden Prozedur und die Parameter enthält
3. Die Nachricht wird zum Server-Prozess übertragen

4. Die Nachricht wird empfangen und dem Server-Stub übergeben
5. Der Server-Stub dekodiert den Prozedur-Namen und die Parameter und ruft die Prozedur auf
6. Der Server-Stub verpackt die Ergebnisse des Prozeduraufrufes in eine Nachricht
7. Die Nachricht wird zum Client-Prozess übertragen
8. Die Nachricht wird empfangen und dem Client-Stub übergeben
9. Der Client-Stub dekodiert die Ergebnisse und übergibt sie dem Aufrufer

Es ist durchaus üblich, dass auf diese Art und Weise viele verschiedene Prozeduren im Server-Prozess aufgerufen werden können. Für jede dieser Prozeduren existiert dann je ein Client- und Server-Stub. Im Server wird dann eine zusätzliche Komponente notwending, ein sogenannter *Dispatcher* (Verteiler), der anhand des mitübertragenen Prozedurnamens den richtigen Server-Stub auswählt:



Eine Abart des RPC ist die *Remote Method Ivocation* (RMI), wobei anstelle einer Prozedur eine Methode eines Objektes aufgerufen wird. Der Client-Stub ist in diesem Fall ein Objekt mit der gleichen Schnittstelle wie das Server-Objekt, wobei jede Methode ein Client-Stub im Sinne des RPC ist. Da es mehrere Instanzen eines Objekttyps geben kann, muss der Client zusätzlich zum Methodennamen auch noch eine Objekt-ID mitschicken, so dass der Dispatcher im Server das richtige Objekt auswählen kann. Da Objekte im Server dynamisch erzeugt und gelöscht werden können, muss der Dispatcher eine Schnittstelle zum An- bzw. Abmelden solcher neu erzeugter bzw. gelöschter Objekte anbieten.

Aufgabenstellung

Sie sollen zunächst ein einfaches Banken-Szenario lokal implementieren und in einem zweiten Schritt durch Bereitstellung von Stubs und Dispatchern den entfernten Zugang dazu ermöglichen.

Das Szenario besteht im wesentlichen aus einem Objekttyp *Konto*. Konto-Objekte verfügen über Methoden zum Einzahlen, Abheben und Abfragen des Kontostandes. Die Konto-Klasse ist als Java-Interface deklariert:

```
// Konto.java

public interface Konto {
    // zahle gegebenen Betrag auf das Konto ein
```

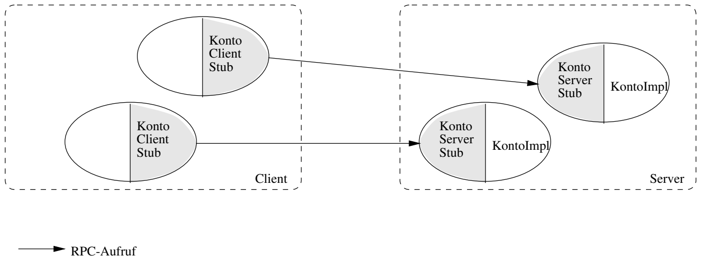
```
    public void einzahlen (long betrag);
    // hebe gegebenen Betrag vom Konto ab, wirft Exception,
    // wenn nicht genügend Geld auf dem Konto
    public void abheben (long betrag)
        throws KontoException;
    public long kontostand ();
    public String kontonummer ();
};
```

Die Methode `Konto.abheben()` löst eine Exception aus, falls nicht genügend Geld auf dem Konto ist:

```
// KontoException.java

public class KontoException extends Throwable {
    KontoException (String msg)
    {
        super (msg);
    }
};
```

Ziel der Aufgabe ist die Enwicklung eines Server-Programms, das mehrere Konto-Objekte enthält, sowie eines Client-Programms, das unter Verwendung des RPC-Mechanismus Methoden der Konten im Server aufruft, um Geld zu überweisen etc.



Beachten Sie insbesondere, dass `KontoImpl`-Objekte nur im Server existieren; im Client wird nur ein passender Konto-Client-Stub (nicht jedoch ein `KontoImpl`-Objekt) erzeugt[1].

Die Abbildung ist insofern vereinfacht, als dass zwischen Client und Server nur eine einzige Socket-Verbindung bestehen soll und der Dispatcher, der im Server die Methodenaufrufe an den richtigen Server-Stub weiterleitet, hier nicht dargestellt ist.

Gehen Sie zur Lösung der Aufgabe folgendermassen vor:

1. Implementieren Sie Konto, indem Sie eine Klasse `KontoImpl` von `Konto` ableiten. Entwickeln Sie ein Testprogramm, um die soeben implementierte Funktionalität zu überprüfen. Das Testprogramm sollte (mittels `new`) mehrere Konton erzeugen, um dann Geld einzuzahlen und abzuheben etc.

2. Überlegen Sie sich ein Protokoll für den Nachrichtenaustausch zwischen Client und Server und welche Informationen die ausgetauschten Nachrichten enthalten müssen. Bedenken Sie dabei insbesondere, dass:

   - der Dispatcher auf der Serverseite anhand der Nachricht entscheiden können muss, welche Methode welches Objektes[2] aufgerufen werden soll.
   - einige Methoden Exceptions auslösen können und der Client-Stub zwischen "normaler" Beendigung eines Methodenaufrufes und Exception unterscheiden können muss.

[1] Es ist also möglich, dass mehrere Client-Stubs auf ein und den selben Server-Stub verweisen.

[2] Dafür müssen Sie jedes Objekt mit einem eindeutigen Namen versehen!

3. Überlegen Sie sich, welche Klassen Sie zur Implementierung des RPC-Mechanismus benötigen und entwerfen Sie die Schnittstellen dieser Klassen. Im wesentlichen benötigen Sie drei grössere Baugruppen: Die Client-Stubs, die Server-Stubs und den Dispatcher. Hier ein paar Hinweise:

   - Bedenken Sie, dass zwischen Client- und Server-Prozess nur eine einzige Socket-Verbindung bestehen soll, die die verschiedenen Stubs gemeinsam benutzen.
   - Es bietet sich an, die gemeinsame Funktionalität aller Stubs in zwei Basisklassen `ClientStubBase` bzw. `ServerStubBase` auszulagern, von der die Client- bzw. Server-Stubs dann erben.
   - Die Client-Stubs erben also von `ClientStubBase` und von der eigentlichen Objekt-Basisklasse (`Konto`) und stellen Stub-Implementierungen für die in `Konto` deklarierten Methoden bereit.
   - Beim Erzeugen eines neuen Client-Stubs sollten Sie die Adresse und Portnummer des zugehörigen Servers und den Namen des zugehörigen Objektes im Server angeben können, d.h. Sie müssen einen entsprechenden Konstruktor bereitstellen.
   - Die Server-Stubs erben von der Klasse `ServerStubBase`, die im wesentlichen eine Methode `dispatch()` mit entsprechenden Parametern deklariert, die der Dispatcher aufruft, um eine Methode auf dem zum Stub gehörenden Objekt auszuführen. Die Server-Stubs stellen dann eine entsprechende Implementierung für `dispatch()` bereit.
   - Damit der Dispatcher ankommende Nachrichten entgegennehmen kann, sollte er eine Methode `run()` bereitstellen, die in einer Endlosschleife auf ankommende Nachrichten wartet und diese dann bearbeitet.
   - Der Dispatcher muss eine geeignete Schnittstelle zum An- bzw. Abmelden von Server-Stubs der neu erzeugten bzw. gelöschten Objekte bereitstellen.
   - Der Dispatcher sollte darauf vorbereitet sein, nacheinander (aber nicht gleichzeitig) mit verschiedenen Clients zu kommunizieren.

4. Implementieren Sie die im vorangegangenen Schritt entworfenen Klassen. Java stellt eine Vielzahl von Klassen bereit, die Ihnen die Aufgabe sehr erleichtern können[3]:

   - `java.util.HashMap` zur Verwaltung der Server-Stubs im Dispatcher.
   - `java.net.Socket` und `java.net.ServerSocket` für die Kommunikation.
   - `java.io.DataInputStream` zum Dekodieren der Nachrichten. Bei Verwendung von `DataInputStream` können Sie das Ende einer Verbindung daran erkennen, dass die entsprechenden `read*()` Methoden eine `java.io.EOFException` auslösen.
   - `java.io.DataOutputStream` zum Dekodieren der Nachrichten. Beachten Sie bei Verwendung von `DataOutputStream`, dass die Daten erst nach einem Aufruf von `flush()` wirklich geschrieben werden.

5. Spalten Sie das bereits existierende Testprogramm in einen Client und Server auf, um die verteilte Version des Bank-Szenarios zu testen. Der Server sollte mehrere Konto-Objekt instanziieren, beim Dispatcher anmelden und mittels `Dispatcher.run()` auf eingehende Methodenaufrufe warten. Der Client sollte je einen Client-Stub für die Konten erzeugen und dann mittels Methodenaufrufen auf diesen Stubs Geld einzahlen etc.

[3] Details zu diesen Klassen finden Sie in der Java API Dokumentation unter `http://java.sun.com/products/jdk/1.2/docs/api/overview-summary.html`.