

# Client/Server-Programmierung I

## CORBA: Schritt-für-Schritt Anleitung (Mini HOWTO)

Version 1.3, 09.11.2009

### Schritt 1: Erstellung der IDL

Zuerst muss eine IDL (Interface Definition Language)-Datei erstellt werden. Die IDL ist unabhängig von der später zu verwendeten Programmiersprache des Servers bzw. Clients. Für das HelloWorld Beispiel sieht die IDL-Datei wie folgt aus:

Datei `Hello.idl`:

```
module HelloWorld
{
    interface Hello
    {
        string sayHello ( in string Name );
    };
};
```

Beachten Sie bitte die speziellen Datentyp-Bezeichnungen der IDL, die sich von der späteren Entwicklungssprache unterscheiden können. Außerdem sind die Möglichkeiten der IDL sehr vielfältig, wie z.B. Vererbung, geschachtelte Module und Definition von Ausnahmen. Darauf wird hier aber nicht näher eingegangen (siehe dazu u.a. [1], [2], [3], [4], [5]). Im Interface `Hello` haben wir eine Methode `sayHello` definiert. Sie hat als Parameter eine `string` Variable `Name` und gibt einen `String` als Antwort zurück.

### Schritt 2: Übersetzung der IDL

Bevor mit der Umsetzung des Clients und des Servers begonnen wird, muss zuerst die Entwicklungssprache für die Client- und für die Serverseite festgelegt werden. Denn davon abhängig ist der zu verwendende IDL-Compiler. Für unser Beispiel verwenden wir `idlj` der CORBA-Implementierung Java IDL, die im JDK enthalten ist (oder alternativ `idl` des JacORB)<sup>1</sup>. Mit dem Befehl

```
idlj -fall Hello.idl
```

werden sowohl die Server- als auch die Clientdateien für unser Beispiel erzeugt. Dies bewirkt der Parameter `-fall`. Alternativ zum Parameter `-fall` gibt es noch die zwei Parameter `-fclient` und `-fserver`, die beim Aufruf nur die jeweiligen Dateien erzeugen. Standardmäßig erzeugt der `idlj` POA (Portable Object Adapter) Server-Skeletons. Der POA ermöglicht Entwicklern die Konstruktion von Servants, die unabhängig von den verwendeten ORBs eingesetzt werden können<sup>2</sup>.

---

<sup>1</sup> Falls das Kommando nicht gefunden wird, erweitern Sie bitte Ihren Kommandosuchpfad um das Verzeichnis `$JAVA_BINDIR: export PATH=$JAVA_BINDIR:$PATH`.

<sup>2</sup> Hinweis: Die Verwendung des Parameters `-oldImplBase` generiert sog. `oldImplBase` Skeletons. Diese Vorgehensweise wird in der gängigen Literatur beschrieben.

Abb. 1 zeigt, wie die Kommunikation zwischen Client und den Servants abläuft. Eine gute Beschreibung zum POA findet sich in [6].

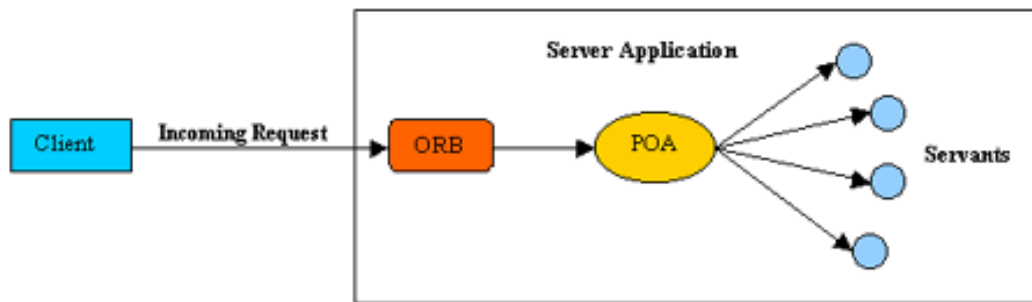


Abbildung 1: Kommunikation ORB-POA-Servants (aus [6])

Der idlj-Übersetzer erzeugt für unser Beispiel ein Verzeichnis HelloWorld mit insgesamt sechs Dateien. Die wichtigste Datei dabei ist HelloPOA.java. Dies ist die abstrakte Basisklasse, von der die zu erstellende Servant-Klasse (HelloServant) erben wird (siehe [4], S.134).

### Schritt 3: Erstellung der Server-Klasse (siehe [4], S.136-140)

Für das Beispiel werden die beiden Klassen HelloServer und HelloServant implementiert.

Die Klasse HelloServer übernimmt die Initialisierung des Brokers über

```
ORB orb = ORB.init ( args, null );
```

Die Klasse HelloServant erbt von der abstrakten Klasse HelloPOA und implementiert die abstrakten Methoden dieser Elternklasse. In der Klasse HelloServer wird ein Objekt der Klasse HelloServant erzeugt:

```
HelloServant helloRef = new HelloServant();
```

Um helloRef bei dem Broker zu registrieren, muss erst der Root-POA bestimmt werden, der vom ORB verwaltet wird. Danach muss dieser explizit aktiviert werden, weil er sich standardmäßig in einem Wartezustand befindet. Ohne diese Aktivierung würden die Anfragen an die Servants in eine Warteschlange geschrieben, aber nicht ausgeführt.

```
POA rootpoa =  
    POAHelper.narrow(orb.resolve_initial_references("RootPOA"));  
rootpoa.the_POAManager().activate();
```

Die folgende Anweisung registriert helloRef als Servant beim Root-POA und gibt eine Objektreferenz für das entstandene CORBA-Objekt zurück:

```
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloRef);
```

Die zurückgegebene Referenz ist aber keine Java-Objektreferenz (Typ: java.lang.Object), sondern eine CORBA-Objektreferenz (Typ: org.omg.CORBA.Object). Daher müssen wir die Referenz über folgenden Methodenaufruf umwandeln:

```
Hello href = HelloHelper.narrow(ref);
```

Die Objektreferenz muss nun noch für den Client erreichbar sein. Das können wir erreichen, indem wir die Referenz beim Namensdienst registrieren. Dazu benötigen wir zunächst eine Referenz auf den Namensdienst, die uns der ORB liefert. Der Name „NameService“ ist dabei für alle CORBA-Implementierungen gleich. Die zurückgelieferte Referenz ist wieder eine CORBA-Objektreferenz, die vor der Verwendung mit `narrow()` in eine Java-Referenz umgewandelt werden muss:

```
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```

Damit das Objekt beim Namensdienst angemeldet werden kann, muss zunächst eine Namenskomponente erzeugt werden. Der Server soll unter dem Namen „HelloWorld“ registriert werden, der nur aus einer einzigen Namenskomponente besteht. `rebind` bindet das Objekt `hRef` im Namensdienst an den gegebenen Namen.

```
NameComponent path[] = ncRef.to_name("HelloWorld");
ncRef.rebind(path, hRef);
```

Als letztes übergeben wir die Kontrolle an den ORB, der nun auf eingehende Anfragen wartet und diese über den POA an den entsprechenden Servant weitergibt:

```
orb.run();
```

## Schritt 4: Erstellung des Clients

Damit ein Client die Dienste eines Servers in Anspruch nehmen kann, muss zuerst eine Verbindung zwischen Client und Servant hergestellt werden. Hierbei erfolgt aber die Kontaktaufnahme nicht direkt mit dem Server, sondern mit dem Broker, der wiederum über den POA mit dem Servant kommuniziert. Hier sind die vom `idlj`-Compiler generierten Clientdateien von Bedeutung, insb. `Hello` und `HelloHelper`.<sup>3</sup>

Zunächst muss eine Referenz auf das `Hello`-Objekt vom Namensdienst geholt werden:

```
NameComponent path[] = ncRef.to_name("HelloWorld")
Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));
```

Über diese Referenz können nun entfernte Methodenaufrufe durchgeführt werden:

```
System.out.println(helloRef.sayHello("Heidi"));
```

## Schritt 5: Compilieren und Starten

Der `idlj`-Compiler erzeugt für die Datei `Hello.idl` ein neues Verzeichnis `HelloWorld`, das die erzeugten Java-Dateien beheimatet. Bevor der Server und der Client compiliert werden können, müssen diese Dateien compiliert werden.

```
#> javac HelloWorld/*.java
```

Danach werden die Java-Dateien des Servers (`HelloServer.java`) und des Clients (`HelloClient.java`) compiliert.

---

<sup>3</sup> Tipp: Um zu erkennen welche Dateien für den Server und den Client nötig sind, kann man den `idlj`-Compiler jeweils mit dem Parametern `-client` und `-server` starten.

Nun muss zuerst der ORB-Daemon gestartet werden. Der ORB-Daemon ist Bestandteil des JDK und sollte sich im Kommandosuchpfad befinden<sup>4</sup>.

```
#> orbd -ORBInitialPort 1050 -port 1049
```

Der Parameter `-ORBInitialPort` gibt den Port an, auf dem der ORB-Daemon auf Namensdienst-Anfragen horcht, `-port` legt fest, auf welchem Port er für Anfragen zu persistenten Objekt-Referenzen erreichbar ist. Standardmäßig werden die Nummern 1050 und 1049 verwendet. Damit es nicht zu Konflikten mit anderen Nutzern kommt, sollten Sie andere Port-Nummern verwenden. Die Nummern müssen über 1024 liegen, weil die Ports bis 1023 reserviert sind und Linux normalen Nutzern grundsätzlich die Verwendung dieser Ports verbietet. Sollte der von Ihnen gewählte Port bereits anderweitig benutzt werden, erhalten Sie eine Fehlermeldung. Versuchen Sie dann einfach eine andere Port-Nummer. Am besten verwenden Sie z.B. 10XX0 und 10XX9, wobei XX die letzten zwei Ziffern ihres csp-Accountnamens sind.

Hinweis: falls Sie eine Datei `orb.properties` des JacORB in Ihrem Home-Verzeichnis haben, sollten Sie diese vor dem Start des JDK ORB-Daemons umbenennen, da es sonst zu Problemen (Exceptions) kommt!

Nach dem Start des ORB muß der `HelloServer` gestartet werden:

```
#> java HelloServer -ORBInitialPort 1050
```

Auch hier muss der Parameter `-ORBInitialPort` angegeben werden, damit der Server mit dem ORB-Daemon kommunizieren kann. Sollten sich `HelloServer` und ORB-Daemon nicht auf dem gleichen Rechner befinden, so ist folgendes Kommando zu verwenden:

```
#> java HelloServer -ORBInitialPort 1050 -ORBInitialHost <Adr_ORB>
```

`<Adr_ORB>` dient hier als Platzhalter für den Rechnernamen bzw. die IP-Adresse des Rechners, auf dem der ORB-Daemon läuft.

Nun kann der Client mit

```
#> java HelloClient -ORBInitialPort 1050
```

bzw.

```
#> java HelloClient -ORBInitialPort 1050 -ORBInitialHost <Adr_ORB>
```

gestartet werden.

---

<sup>4</sup> Falls das Kommando nicht gefunden wird, erweitern Sie bitte Ihren Kommandosuchpfad um das Verzeichnis `$JAVA_BINDIR`: `export PATH=$JAVA_BINDIR:$PATH`.

# Zusammenfassung

Zusammenfassend lässt sich die Entwicklung wie folgt darstellen:

1. IDL-Datei erzeugen
2. Mit IDL-Compiler (evtl. sprachabhängig!) Server- und Clientdateien erzeugen
3. Server implementieren (Servants und Hauptprogramm)
4. Client implementieren
5. Compilierung der vom IDL-Compiler generierten Dateien.
6. Server compilieren
7. Client compilieren
8. Evtl. die entsprechenden Dateien auf Client- und Server-Rechner übertragen

Die Ausführung der CORBA-Anwendung erfolgt mit:

1. ORB-Daemon starten auf Port x ( $x \geq 1024$ )
2. Server starten auf Port x (evtl. Hostangabe des ORB-Daemons)
3. Client starten auf Port x (evtl. Hostangabe des ORB-Daemons)

## Literatur und Verweise

- [1] Farley, Jim; et al.: „*Java Enterprise in a nutshell*“, 2. Aufl., O'Reilly Verlag, Sebastopol, 2002
- [2] Hofmann, Johann, et al.: „*Programmieren mit COM und CORBA*“, Carl Hanser Verlag, München (u.a.), 2001
- [3] Boger, Marko: „*Java in verteilten Systemen*“, dpunkt Verlag, Heidelberg, 1999
- [4] Langner, Torsten: „*Verteilte Anwendungen in Java*“, Markt+Technik Verlag, München, 2002
- [5] Orfali, Robert; Harkey, Dan: „*Client/ Server Programming with JAVA and CORBA*“, 2<sup>nd</sup> Edition, John Wiley & Sons Verlag, New York, 1998
- [6] SUN Microsystems: „*CORBA Programming with J2SE 1.4*“, <http://www.oracle.com/technetwork/articles/javase/corba-137639.html>, aufgerufen am 15.11.13
- [7] SUN Microsystems: CORBA Links, <http://docs.oracle.com/javase/7/docs/technotes/guides/idl/index.html>, aufgerufen am 15.11.13