JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

DezSys-07 “Verteilte Objekte mit Corba”

Systemtechnik Labor
4CHIT 2015/16, Gruppe A

Gabriel Frassl

Version 1.0

Note:

Betreuer: M.Borko

Begonnen am 15. April 2016

Beendet am 22. April 2016

Inhalt

1 Einführung

Verteilte Objekte haben bestimmte Grunderfordernisse, die mittels implementierten Middlewares leicht verwendet werden können. Das Verständnis hinter diesen Mechanismen ist aber notwendig, um funktionale Anforderungen entsprechend sicher und stabil implementieren zu können.

1.1 Ziele

Diese Übung gibt eine einfache Einführung in die Verwendung von verteilten Objekten mittels CORBA. Es wird speziell Augenmerk auf die Referenzverwaltung sowie Serialisierung von Objekten gelegt. Es soll dabei eine einfache verteilte Applikation in zwei unterschiedlichen Programmiersprachen implementiert werden.

1.2 Voraussetzungen

- Grundlagen Java, C++ oder anderen objektorientierten Programmiersprachen
- Grundlagen zu verteilten Systemen und Netzwerkverbindungen
- Grundlegendes Verständnis von nebenläufigen Prozessen

1.3 Aufgabenstellung

Verwenden Sie das Paket ORBacus oder omniORB bzw. JacORB um Java und C++ ORB-Implementationen zum Laufen zu bringen.

Passen Sie eines der Demoprogramme (nicht Echo/HalloWelt) so an, dass Sie einen Namingservice verwenden, welches ein Objekt anbietet, das von jeweils einer anderen Sprache (Java/C++) verteilt angesprochen wird. Beachten Sie dabei, dass eine IDL-Implementierung vorhanden ist um die unterschiedlichen Sprachen abgleichen zu können.

Vorschlag: Verwenden Sie für die Implementierungsumgebung eine Linux-Distribution, da eine optionale Kompilierung einfacher zu konfigurieren ist.

2 Ergebnisse

2.1 CORBA Theorie [1]

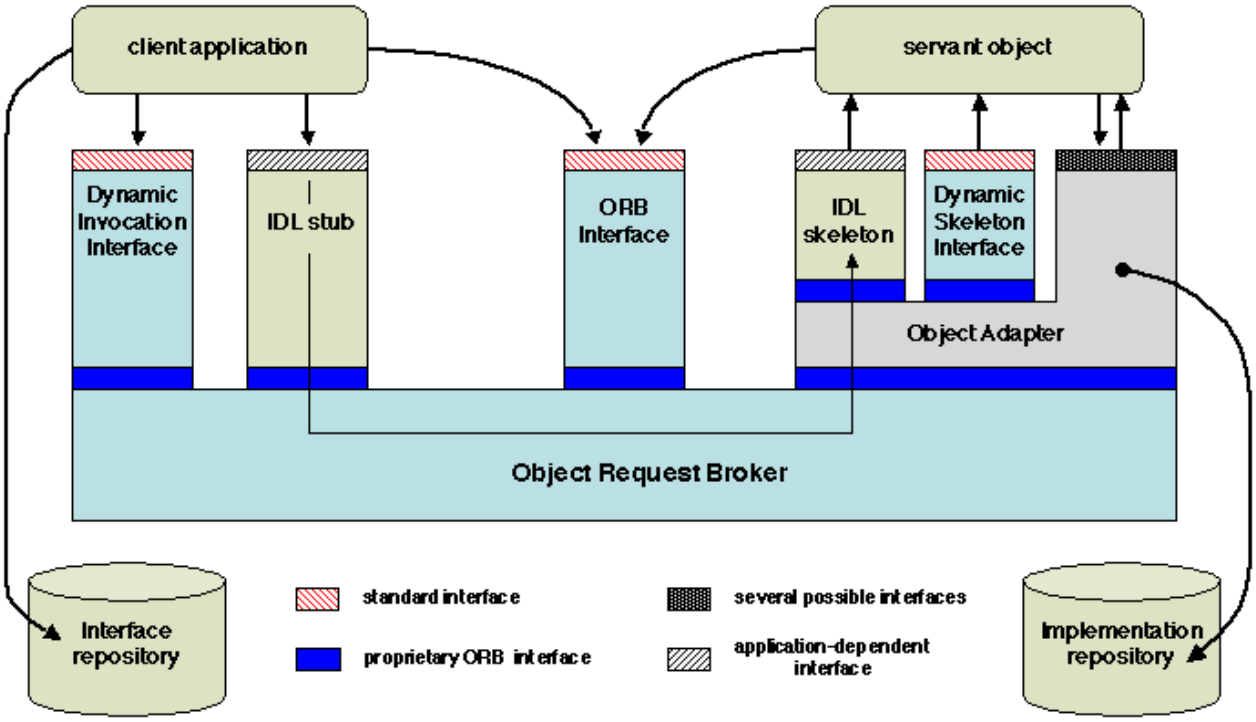
Einführung

CORBA (COMMON OBJEKT REQUEST BROKER ARCHITECTURE) ist eine Architektur die für das Erstellen von verteilten Anwendungen, und dem Aufruf eben dieser dient.

Wie das Wort COMMON im Namen schon sagt ist CORBA eine plattformübergreifende Lösung. Das heißt, dass damit eine Anwendung erstellt werden kann die mit mehreren Programmiersprachen funktioniert. So kann zum Beispiel ein Java Programm die Methode eines verteilten C++ Programmes aufrufen.

Dabei liefert CORBA selbst nur die Spezifikationen aufgrund derer andere Unternehmen oder Communitys dann eine eigene Implementation in Form eines ORB’s für eine bestimmte Sprache erstellen. Aufgrund der gemeinsamen CORBA Schnittstelle können diese ORB’s miteinander kommunizieren.

Aufbau



Object Request Broker:

Wie bereits erwähnt sind die ORB’s Implementierungen des von CORBA spezifizierten ORB Interfaces.

Sowohl Client als auch der Server haben Zugriff auf diesen ORB da dieser für jegliche Kommunikation zwischen den beiden zuständig ist. Die Funktionsweise eines ORB ähnelt dem eines Datenbus.

Sämtliche Services wie zum Beispiel der Naming service werden ebenfalls vom ORB gesteuert und angesprochen.

(Portable) Objekt Adapter (P)OA

Der POA hat die Aufgabe alle benötigten Objekte und Interfaces auf Serverseite zu verwalten, und bei einem Aufruf eines Clients für die Bereitstellung des Objektes und möglicherweise des Skeletons zu sorgen.

- Generieren der Objekt Referenz
- Methodenaufruf
- Security
- Mappen von Objekt Referenz mit Objekt Implementierung
- Implementierung registrieren

Interface Definition Language / IDL Compiler

Die IDL ist ein kritischer Bestandteil der Plattformunabhängigkeit von CORBA, da bei der Method Invokation sowohl auf Clientseite als auch auf Serverseite Interfaces (Stub, Skeleton) vorhanden sein müssen, die den Zugriff auf das verteilte Objekt beschreiben.

Mit Hilfe des IDL Compilers können nun bestimmte Interfaces zu einem idl File kompiliert werden. Nun kann der andere Teil der verteilten Anwendung, welcher möglicherweise in einer anderen Sprache geschrieben ist, einfach dieses IDL File zu einem Interface seiner eigenen Sprache (rück)kompilieren.

Dies erfolgt durch ein Mapping welches angibt wie ein Interface in IDL in der eigenen Programmiersprache aussieht.

Dabei gillt:

- Descriptive Sprache
- Ohne Logik
- Nur Syntax

Interfaces

Wie auch bei RMI gibt es zwei Möglichkeiten des verteilten Methodenaufrufes. Die statische Möglichkeit so wie die dynamische Möglichkeit.

Bei einem statischen Aufruf werden ausschließlich die Interfaces die durch den IDL Compiler erstellt werden verwendet.

Für dynamischen Aufrufe gibt zusätzliche spezielle **Dynamic Invocation Interfaces.**

Dabei muss mein Methodenaufruf folgendes übergeben werden:

- Objekt Reference
- Operation welche ausgeführt werden soll
- Parameter für die Operation

Usage:

- getInterface(); //objekt.getInterface um InterfaceDef. zu erhalten
- describe_interface(); //um erhaltene InterfaceDef zu beschreiben
- create_request(); //object.create_request(objref,operation,param)
- invoke(),send()

2.2 CORBA Setup

Für unsere Übungen wollen wir zwei ORB’s auf einer virtuellen DEBIAN Maschine installieren.

Dafür verwenden wir für den Java Teil der Übung(Client) **JACORB** und für den C++ Teil der Übung(Server) **OmniOrb.**

OmniOrb Installation/Setup

1)

Um OmniOrb herunterzuladen führen wir im gewünschten Verzeichnis folgendes aus:

apt-get install https://sourceforge.net/projects/omniorb/files/latest/download?source=files

2)

Nun gehen wir in das heruntergeladene Verzeichnis und erstellen mit *mkdir build* einen Ordner in dem wir OmniOrb builden wollen.

3)

Jedoch wird das nicht funktionieren bevor wir nicht noch einige Komponenten die dazu notwendig sind installiert haben.

Also laden wir zunächst wieder mit *apt-get install folgende Pakete runter:*

- *gcc (+suggested packages)*
- *g++ (+suggested packages)*
- *libpython-2.7-dev*

4)

Nun können wir das OmniOrb builden. Dazu führen wir im Ordner build folgende Befehle aus:

../configure, make, make install

5)*Nun muss noch ein* ***Naming Service*** *erstellt werden:*

mkdir /var/omninames welcher später beim Testen wie folgt gestartet wird

omniNames -start -always

JacORB Installation/Setup

Bei jacORB haben wir uns dazu entschieden nur die Binary’s runterzuladen. Daher fällt das Setup deutlich leichter aus.

Im Verzeichniss in welchen JacORB installiert werden soll führen wir

apt-get install http://www.jacorb.org/releases/3.7/jacorb-3.7-binary.zip

aus.

Ordner /opt

Für einen leichteren Zugriff auf die gebuildeten ORB’s wollen wir diese nochmal extra in einem eigenen Verzeichnis speichern. Dies vereinfacht außerdem den Zugriff der Anwendungen auf die ORB’s.

Also kopieren wir mit dem Befehl cp den Inhalt des des JaCORB Verzeichnisses sowie den Inhalt des /build Verzeichnisses in den Ordner opt.

cp /SYT/omniorb/build omniorb.versionnmr

cp /SYT/jacorb jacorb.versionnmr

Desweitern erstellen wir in /opt nochmal zwei Ordner die auf unsere beiden ORB’s verlinken, nur diesmal ohne Versionsnummer.

ln -s omniorb.version omniorb

ln -s jacorb.version omniorb

```
root@debian:~/opt# ls
jacorb  jacorb-3.7  omniORB  omniORB-4.2.1
```

Zusätzliche Pakete

Für die spätere Ausführung der Programme und die Arbeit mit Git laden wir noch folgende Pakete runter.

apt-get install git ant openjdk-8-jdk

2.3 Callback Anwendung

Nachdem beide ORB’s erfolgreich installiert wurden, und dies mit dem von Herr Professor Borko zur Verfügung gestellten Hello World Programm [2] getestet wurde, soll nun eine eigene Anwendung verteilt werden.

Dazu nehmen wir die callback-server Anwendung aus den Examples von OmniOrb und schreiben unsere eigene Client-java Anwendung dazu.

Server Seite

Auf der Server Seite unseres verteilten Systems brauchen wir die die c++ Anwendung und ein Makefile in welchem der Pfad zum IDL Compiler und dem ORB angegeben werden.

Da der C++ Code der Anwendung ja vorhanden ist, muss hier nur noch das MakeFile erstellt werden. Als Basis dafür nehmen wir das MakeFile welches wir beim HelloWorld[2] Programm benutz haben.

Zusatz: Sollte man das IDL file anders benennen als in den examples muss man den Import am Anfang des C++ Files natürlich ändern.

```
CXX             = /usr/bin/g++
CPPFLAGS        = -g -c
LDFLAGS         = -g
OMNI_HOME       = ~/opt/omniORB
OMNIIDL         = $(OMNI_HOME)/bin/omniidl
LIBS            = -lomniORB4 -lomnithread -lomniDynamic4
OBJECTS         = echoSK.o server.o
IDL_DIR         = ../idl
IDL_FILE        = $(IDL_DIR)/echo.idl

all server: $(OBJECTS)
        $(CXX) $(LDFLAGS) -o server server.o echoSK.o $(LIBS)

server.o: server.cc
        $(CXX) $(CPPFLAGS) server.cc -I.

echoSK.o: echoSK.cc echo.hh
        $(CXX) $(CPPFLAGS) echoSK.cc -I.

echoSK.cc: $(IDL_FILE)
        $(OMNIIDL) -bcxx $(IDL_FILE)

run: server
        # Start Naming service with command 'omniNames -start -always' as root
        ./server -ORBInitRef NameService=corbaname::localhost

clean clean-up:
        rm -rf *.o
        rm -rf *.hh
        rm -rf *SK.cc
        rm -rf server
```

1 Makefile ServerSeite

IDL File

Natürlich muss auch hier ein IDL-Interface vorhanden sein damit unsere beiden Anwendungen miteinander kommunizieren können.

Hier das benötigte IDL File <echo.idl>:

```
#ifndef __ECHO_CALLBACK_IDL__
#define __ECHO_CALLBACK_IDL__


module cb {

  interface CallBack {

    void call_back(in string mesg);

  };


  interface Server {

    // Server calls back to client just once in a
    // recursive call before returning.
    void one_time(in CallBack cb, in string mesg);

    // Server remembers the client's reference, and
    // will call the call-back periodically.  It stops
    // only when shutdown, or a call to the client fails.
    void register(in CallBack cb, in string mesg,
                  in unsigned short period_secs);

    // Shuts down the server.
    void shutdown();

  };

};


#endif
```

Client Seite

Nun müssen wir eine passende Java Anwendung und das dazugehörige build.xml erstellen die die gewünschten Methoden des Server Objektes aufruft.

JAVA Anwendung:

```
public class Client extends CallBackPOA {
    public static void main(String[] args)  {
        Server echo;
        try {

            /* Erstellen und intialisieren des ORB */
            ORB orb = ORB.init(args, null);

            /* Erhalten des RootContext des angegebenen Namingservices */
            Object o = orb.resolve_initial_references("NameService");

            /* Verwenden von NamingContextExt */
            NamingContextExt rootContext = NamingContextExtHelper.narrow(o);

            /* Angeben des Pfades zum Echo Objekt */
            NameComponent[] name = new NameComponent[2];
            name[0] = new NameComponent("test","my_context");
            name[1] = new NameComponent("Echo", "Object");

            //Aufloesen der Objektreferenzen  */
            echo = ServerHelper.narrow(rootContext.resolve(name));
            POA root_poa = (POA) orb.resolve_initial_references ("RootPOA");
            root_poa.the_POAManager().activate();
            CallBack cbClient = CallBackHelper.narrow (root_poa.servant_to_reference (new Client()));


            short period = 5;
            echo.one_time(cbClient, "Aufruf der einmaligen Callback Methode");
            echo.register(cbClient, "Aufruf der regelmäßigen Callback Methode", period);

            try {
                Thread.sleep(5000);
            } catch (Exception e) {
                e.printStackTrace();
            }


        }   catch (Exception e) {
            System.err.println("Es ist ein Fehler aufgetreten: " + e.getMessage());
            e.printStackTrace();
        }
    }
    public void call_back(String msg) {
         System.out.println ("Der Client hat folgende Nachricht erhalten >" + msg + '<');
    }
```

build.xml

Hier nun das build File. Das in den Code-examplen mitgeschickte build.xml wurde wieder als Grundlage verwendet[2].

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xml>

<project name="client">

    <!-- Setzen aller Variablen -->
    <property name="src.dir" value="src" />
    <property name="build.dir" value="build" />
    <property name="classes.dir" value="${build.dir}/classes" />
    <property name="doc.dir" value="doc" />
    <property name="idl.dir" value="../idl" />
    <property name="gen.dir" value="${build.dir}/generated" />
    <property name="resources.dir" value="resources" />
    <property name="jacorb.dir" value="/root/opt/jacorb /> 
    <property name="tmp.dir" value="${build.dir}/tmp" />
    <property name="host" value="127.0.0.1" />

    <!-- Uebergeben der Argumente -->
    <property name="jaco.args" value="-Dignored=value" />

    <!-- Setzen des Classpaths von JacORB -->
    <path id="jacorb.classpath">

        <!-- Setzen des Pfades zu, und inkludieren der Libaries -->
        <fileset dir="${jacorb.dir}/lib">
            <include name="*.jar" />
        </fileset>
    </path>

    <!-- Setzen des Classpaths des Projekts (classes Ordner in build) -->
    <path id="project.classpath">
        <pathelement location="${classes.dir}" />
    </path>

    <!-- Definieren eines in einer bestimmten Klasse vorhandenen Tasks -->
    <target name="idl.taskdef">
        <taskdef name="jacidl" classname="org.jacorb.idl.JacIDL"
            classpathref="jacorb.classpath" />
    </target>

    <!-- Generieren des aus dem idl File resultierenden Quellcodes  -->
    <target name="idl" depends="idl.taskdef">
        <mkdir dir="${idl.dir}" />
        <jacidl srcdir="${idl.dir}" destdir="${gen.dir}" includes="*.idl"
            helpercompat="jacorb" includepath="${jacorb.dir}/idl/omg" />
    </target>

    <!-- Kompilieren des Quellcodes -->
    <target name="compile" depends="idl">
        <mkdir dir="${classes.dir}" />
```

```
    <javac destdir="${classes.dir}" debug="true" includeantruntime="false">
        <src path="${gen.dir}" />
        <src path="${src.dir}" />
        <classpath refid="jacorb.classpath" />
    </javac>
</target>

<!-- Ausfuehren des Clients -->
<target name="run-client" depends="compile">
    <description>
        Dem Client kann eine Hostadresse mitgegeben werden.
        Ein Aufruf ist mit 'ant run-client -Dhost=host' möglich.
        Beispielaufruf: ant run-client -Dhost=127.0.0.1

        Sollte kein Host angegeben werden, so wird localhost als Host verwendet.
    </description>
    <java fork="true" classname="cb.Client">

        <!-- Wuerde folgendem Aufruf entsprechen: java helloworld.Client -ORBInitRef NameService=corbaloc::127.0.0.1:2
        <arg value="-ORBInitRef" />
        <arg value="NameService=corbaloc::${host}:2809/NameService" />
        <classpath refid="project.classpath" />
    </java>
</target>

<!-- Loeschen des build Ordners -->
<target name="clean">
    <delete dir="${build.dir}" />
</target>
```

2.4 Testen

Für das testen einer Anwendung müssen folgende Schritte getätigt werden.

- Terminal 1:Starten des in Kapitel 2.2 beschriebenen NameServices
    - omniNames –start –always
- Terminal 2: Starten der Server anwendung
    - Make run im Server Verzeichniss
- Terminal 3 : Starten der Client Anwendung
    - ant run-client

HelloWorld testen

Zuerst Testen wir das HelloWorld Programm aus den Code Examplen

Server starten:

```
root@debian:/SYT/code-examples/corba/halloWelt/server# make run
# Start Naming service with command 'omniNames -start -always' as root
./server -ORBInitRef NameService=corbaname::localhost
IOR:010000001800000049444c3a68656c6c6f776f726c642f4563686f3a312e3000010000000000
0000680000000101020010000000003139322e3136382e3134392e31333200cee000000e000000fe71
4819570000164c0000000000000002000000000000000800000001000000005454410100000001c00
0000100000001000100010000000100010509010100010000000 9010100
```

Client ausführen

```
idl.taskdef:

idl:
   [jacidl] processing idl file: /SYT/code-examples/corba/halloWelt/idl/echo.idl

compile:

run-client:
     [java] Der Server sagt: Hallo Welt

BUILD SUCCESSFUL
Total time: 1 second
root@debian:/SYT/code-examples/corba/halloWelt/client#
```

Callback testen

Server starten

```
root@debian:/SYT/DezSys07_CORBA_Frassl/server# make run
# Start Naming service with command 'omniNames -start -always' as root
./server -ORBInitRef NameService=corbaname::localhost
IOR:010000001200000049444c3a63622f5365727665723a312e3000000001000000000000000
0000010102001000000003139322e3136382e3134392e31333200458700000e000000fec55719
0018620000000000000002000000000000000800000001000000005454410100000001c000000
0000010001000100000001000105090101000100000009010100
```

Client starten

```
idl.taskdef:

idl:
   [jacidl] processing idl file: /SYT/DezSys07_CORBA_Frassl/idl/echo.idl

compile:
     [javac] Compiling 1 source file to /SYT/DezSys07_CORBA_Frassl/client/build/c
lasses

run-client:
     [java] Der Client hat folgende Nachricht erhalten >Aufruf der einmaligen Ca
llback Methode<
     [java] Der Client hat folgende Nachricht erhalten >Aufruf der regelmäßigen
Callback Methode<
```

3 Quellen / Literatur

"omniORB : Free CORBA ORB"; Duncan Grisby; 28.09.2015; online: http://omniorb.sourceforge.net/

"Orbacus"; Micro Focus; online: https://www.microfocus.com/products/corba/orbacus/orbacus.aspx

"JacORB - The free Java implementation of the OMG's CORBA standard."; 03.11.2015; online: http://www.jacorb.org/

"The omniORB version 4.2 Users’ Guide"; Duncan Grisby; 11.03.2014; online: http://omniorb.sourceforge.net/omni42/omniORB.pdf

"CORBA/C++ Programming with ORBacus Student Workbook"; IONA Technologies, Inc.; September 2001; online: http://www.ing.iac.es/~docs/external/corba/book.pdf

[1]FOLIENSATZ **Distributet-Objects Corba** aus dem Unterricht

[1]Code Examples: https://github.com/mborko/code-examples.git

4 Zeitaufwand/ Probleme

Die Konfiguration der ORB’s und des Naming Services empfand ich zuerst als sehr unübersichtlich und schwierig. Trotzdem habe ich es dann aufgrund der ausführlichen Erklärung im Unterricht und auch durch Hilfestellung von einem Mitschüler geschafft, die Konfiguration innerhalb der Laborunterrichtszeit zu meistern.

Das Einbauen der beiden Aufaben (HelloWorld,Callback) wurde aufgrund der code-examples und der Beispielprogramme der beiden ORB’s deutlich verständlicher. So konnten bereits vorhandene Files für das eigene Programm angepasst werden. Trotzdem sind mir häufig Fehler in den source- Files und den Makefiles unterlaufen, die es zu beheben galt.

Tätigkeit	Zeitaufwand	Datum/Ort
ORB’s Konfigurieren NamingService	180	Labor 16.04.2016
HelloWorld Implementieren	60	Labor 16.04.2016
Callback Serverseite und IDL (omni examples)	60	Zuhause 19.04.2016
Callback ClientSeite	150	Zuhause 21.04.2016
Protokoll	300	Zuhause 22.04.2016
Gesamt	12.5 Stunden	

5 GITHUB

https://github.com/gfrassl-tgm/DezSys07_CORBA.git