JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

CORBA

Systemtechnik Labor
4BHITT 2015/16, Gruppe Y

Maximilian Seidl

Version 0.2

Note:

Betreuer: M. Borko

Begonnen am 2. Oktober 2015

Beendet am 2. Oktober 2015

Inhaltsverzeichnis

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.

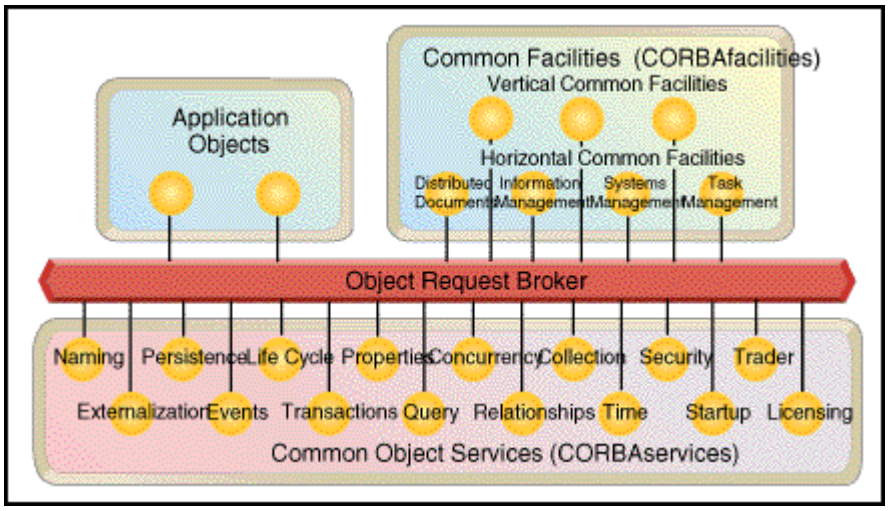
Vorschlag: Verwenden Sie für die Implementierungsumgebung eine Linux-Distribution, da eine optionale Kompilierung einfacher zu konfigurieren ist.

2 Ergebnisse

2.1 Definition CORBA?

CORBA ist eine Abkürzung für „Common Object Request Broker" . Mit CORBA ist es möglich Objekte für mehrere Prozesse im Netz nutzbar zu machen. Der grobe Unterschied zu RMI (Remote Method Invokation) ist dabei, dass sogenannte CORBA-Objekte nicht abhängig von einer Programmiersprache sind. Dies bedeutet es können Programme verschiedenster Programmiersprachen mit einander Objekte austauschen und bearbeiten. Zum Beispiel C++ mit Java.

Die Kommunikation zwischen diesen Programmen erfolgt mittels „ORB´s" über ein Bussystem.



http://twimgs.com/ddj/sdmagazine/images/sdm9704a/9704af1.gif

Der Bus , oder „Object Request Broker" genannt, beschreiben den Ort des Ziels bei einer Anfrage, senden an das Objekt einen Request und eine jeweilige Response an den Anfragenden zurück.

2.2 Vorbereitung für die Laborübung

Zunächst benötigt man folgende Packages:

- Java und ein Java-Build Tool: zb.: jdk8-openjdk und ant
- C/C++ und make als Build Tool
- Python für das Kompilieren von omniORB: python2

Wir haben uns JacORB als CORBA-Libary für Java ausgesucht und omniORB für

C++ und Pyhton. omniORB muss selber kompiliert werden und bei JacORB werden auf die Binaries zurückgegriffen.

Nachdem die beiden ORB´s kompiliert wurden kann man diese auch Testen mittels des zur Verfügung stehenden Testcodes. Dieser ist im git-Repo von Michael Borko zu finden:

https://github.com/mborko

omniORB

Downloadlink für omniORB:
https://sourceforge.net/projects/omniorb/files/omniORB/omniORB-4.2.1/omniORB-4.2.1-2.tar.bz2/download

Um omniORB zu kompilieren muss man zunächst ein build Verzeichnis erstellen und in diesem die komprimierten Dateien entpacken. Danach wird das script-File „configure" mit *./configure* ausgeführt, und es erstellen sich make-Files. Nun ruft man mittels dem Build Tool *make* die Option *install* auf. Da sich nun neue Libaries hinzugefügt haben, muss man diese auf den richtigen Pfad updaten. Dies Funktioniert mittels dem Befehl ldconfig. Für die Übung arbeiten wir mit dem omniORB-Namingservice welcher vor jedem Test gestartet werden muss. Der Namingservice wird mittels *omniNames -start -always* gestartet.

JacORB

Downloadlink für JacORB:

http://www.jacorb.org/releases/3.7/jacorb−3.7−binary.zip

Wenn die benötigten Pakete bereits installiert sind ist die Konfiguration ziemlich simpel. Man muss lediglich die Files in das Directory */opt/* kopieren und dort dann einen Symlink zur neuesten Version machen.

Test mit HelloWorld

Nachdem die beiden ORB´s kompiliert wurden kann man diese auch Testen mittels des zur Verfügung stehenden Testcodes. Dieser ist im git-Repo von Michael Borko zu finden:

https://github.com/mborko

3 Implementieren des eigenen Programmes

Der Code ist vollständig auf meinem git-Repo zu finden:

https://github.com/mseidl-tgm/syt_lab

Ich habe das bereits bestehende Hello World Programm abgeändert in einen simplen Rechner, welcher addieren, subtrahieren, dividieren und multiplizieren kann.

3.1 Server

Die Schnittstelle zwischen den ORB´s und dem eigentlichen Code übernehmen sogenannte POA´s (Portable Object Adapter). Diese leiten Aufrufe an das konkrete Programm weiter.

```
Static CORBA::Boolean bindObjectToName(CORBA::ORB_ptr, CORBA::Object_ptr);


class Calculator_i : public POA_calc::Calculator {
public:
  inline Calculator_i() {}
  virtual ~Calculator_i() {}
  virtual int addition( ons tint number1,  ons tint number2);
  virtual int subtraction( ons tint number1,  ons tint number2);
  virtual int multiplication( ons tint number1,  ons tint number2);
  virtual int division( ons tint number1,  ons tint number2);
};

int Calculator_i::addition( ons tint number1,  ons tint number2) {
    return CORBA::Float(number1+number2);
}

int Calculator_i::subtraction( ons tint number1,  ons tint number2) {
    return CORBA::Float(number1-number2);
}

int Calculator_i::multiplication( ons tint number1,  ons tint number2) {
    return CORBA::Float(number1*number2);
}

int Calculator_i::division( ons tint number1,  ons tint number2) {
    return CORBA::Float(number1/number2);
}
```

Server.cc – Klasse welche die ORB und POA´s erstellt und Objekte bindet

3.2 IDL (Interface)

Das idl-File dient als Schnittstelle zwischen den beiden Programmen. Es wirkt wie ein Interface, welches beide Implementieren müssen.

```
fndef __CALC_IDL__
#define __CALC_IDL__
module calc {
    interface Calculator {
        long addition(in long number1, in long number2);
        long subtraction(in long number1, in long number2);
        long multiplication(in long number1, in long number2);
        long division(in long number1, in long number2);
    };
};
#endif // __CALC_IDL__
```

Echo.idl – Interface/Schnittstelle

Hier ein die konkrete Implementation des idl-Files im Client.

```
public static int addition(Calculator calculator, int number1, int number2) {
        return calculator.addition(number1, number2);
    }

    public static int subtraction(Calculator calculator, int number1, int
number2) {
        return calculator.subtraction(number1,number2);
    }

    public static int multiplication(Calculator calculator, int number1, int
number2) {
        return calculator.multiplication(number1,number2);
    }

    public static int division(Calculator calculator, int number1, int number2)
{
        return calculator.division(number1,number2);
}
```

Client.java – Implementation der Methoden

3.3 Client

Hier ist die main-Methode des Java-Clients. Diese erstellt ein ORB, holt sich die Referenzen vom NamingService und gibt den Pfad zum Objekt an.

```
public static void main(String[] args)  {

        Calculator calculator;

        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("Calculator", "Object");

                        /* Aufloesen der Objektreferenzen  */
            calculator = CalculatorHelper.narrow(rootContext.resolve(name));

            int number1 = 4;
            int number2 = 3;

            System.out.println("Addition von " + number1 + " und " + number2 +
":" + addition(calculator, number1, number2));
            System.out.println("Subtraktion von " + number1 + " und " + number2
+
":" + subtraction(calculator, number1, number2));
            System.out.println("Multiplikation von " + number1 + " und " +
number2 + ":" + multiplication(calculator, number1, number2));
            System.out.println("Division von " + number1 + " und " + number2 +
":" + division(calculator, number1, number2));

        }       catch (Exception e)     {
            System.err.println("Es ist ein Fehler aufgetreten: " +
e.getMessage());
            e.printStackTrace();
        }
}
```

4 Zeitaufwand

Datum	Dauer	Beschreibung
29.4.2016	4 Stunden	Tutorial einbinden
5.5.2016	3 Stunden	Umschreiben in eigene Implementation
1.6.2016	2 Stunden	Protokoll schreiben

5 Quellen

"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

REF __RefHeading___Toc113_1963609346 \h **Fehler! Verweisquelle konnte nicht gefunden werden.**