CORBA

Ein **Object Request Broker** (ORB) ist ein Kommunikationsmechanismus für die **synchrone Interaktion zwischen verteilten**, in verschiedenen Programmiersprachen implementierten **Objekten** über verschiedene Netzwerke und Betriebssystemplattformen hinweg.

CORBA ist eine Abkürzung für **Common Object Request Broker Architecture**, es ist ein Standard einer Architektur zur **Kommunikation** in **verteilten Systemen** auf **Ebene** von **Objekten**.

In diesem Teil wird in die Architektur von CORBA und das Programmieren von ORB Anwendungen behandelt. Dazu wird die CORBA Implementierung MICO verwendet und C++ als Programmiersprache verwendet.

Im nächsten Kapitel wird dann CORBA Programmierung mit Java behandelt.

Inhalt

1.Überblick.. 3

2.Programmieren mit CORBA und C++.............................. 8

2.1.Beispielanwendung...8

2.2.Vorgehensweise beim Entwurf.................................10

2.3.Definition der Interfaces....................11
2.4.Implementierung des Serverobjektes....................16
2.5.Die Serverapplikation....................24
2.6.CORBA Clients.................... 33
2.7.Der MICO-Binder....................41
2.8.Der CORBA Naming Service....................51

1. Überblick

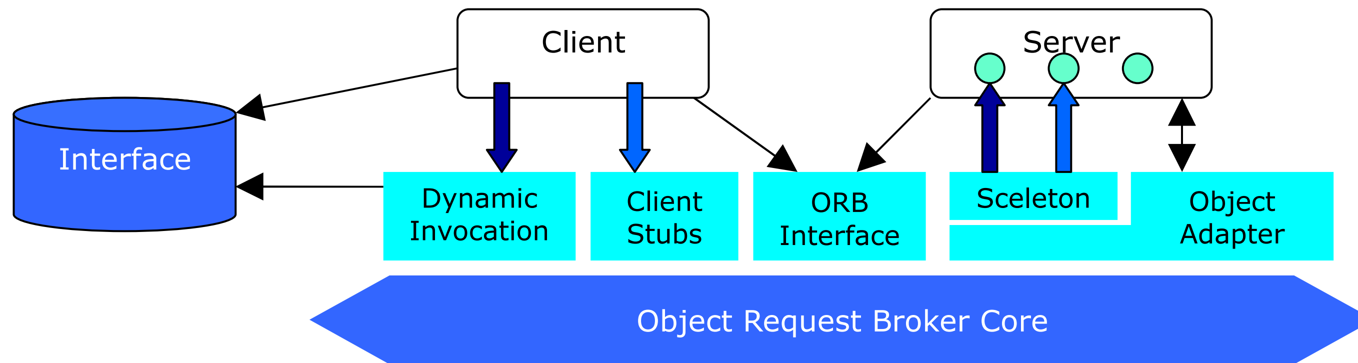
CORBA ist ein Standard einer Architektur zur Kommunikation in verteilten Systemen auf Ebene von Objekten.

Bei der Kommunikation vermittelt CORBA zwischen den Objekten auf Client-, bzw. Serverseite, und führt deren Anfragen (die sog. *Requests*) durch. CORBA ist dabei an keine spezielle Plattformarchitektur gebunden, es ist unabhängig von Betriebssystem, Hardware oder Programmiersprachen (sofern diese objektorientiert arbeiten).

Damit **vereint CORBA** zwei der wichtigsten **Konzepte** der Informatik, nämlich ***objektorientierte Programmierung*** und **Programmierung** in ***verteilten Systemen***.

Aufgabe des **ORB** ist es, eine **Kommunikation** zwischen Client und Server aufzubauen, wobei der **Client eine Methode eines Serverobjektes aufruft**.

Damit stellt **CORBA** einen **logischen Softwarebus** dar über den Methoden eines entfernten Objektes ausgeführt werden.



In CORBA werden die **Stubs** (=Sceleton) mittels eines **IDL-Programms** (IDL=Inferface Definition Language, ähnlch wie C++) **generiert**.

Die **Serverobjekte** werden im **Object Adapter verwaltet**. Über die Stubs kommunizieren die Objekte statisch. Zur dynamischen Bindung (Objekte werden erst zur Laufzeit bestimmt) werden vom ORB Schnittstellenspezifikationen in einem Interface Repository verwaltet. Das ORB Interface dient zum Zugriff von Objekten **auf ORB lokale Dienste.**

Ein ORB kann auf Basis von RPC implementiert werden. Im Vergleich mit einem RPC System müssen dabei zusätzlich die Bestandteile objektorientierter Konzepte wie Vererbung, Polymorphismus realisiert werden.

Wie der RPC baut der **ORB** eine **synchrone Punkt-zuPunkt Verbindung** zwischen Client und Server auf. Zur Entkopplung gibt es in CORBA einen Event Service, der im Sinne einer Publish Subscribe Gruppenbeziehung zwischen Objekten vermittelt.

Die folgende Beschreibung von CORBA entstammt (als Übersetzung) direkt der OMG.

CORBA, die Common Object Request Broker Architecture wurde von der **Object Management Group (OMG)**, einem Standardisierungsgremium mit mehr als 700 Mitgliedern, 1991 in der ersten Version **definiert**.

CORBA war eine Antwort auf die starke Zunahme von Hardware- und Software-Produkten und Ziel war es, eine **Middleware** zu schaffen, welche eine **orts**-, **plattform**- und **implementationsunabhängige Kommunikation** zwischen Applikationen erlaubt.

Wirklich interessant geworden ist CORBA seit der Verabschiedung der Version 2.0 im Dezember 1994. Diese Version brachte das **Kommunikationsprotokoll IIOP**, welches den Meldungsaustausch zwischen **Object Request Brokern** (**ORB**) verschiedener Hersteller und vor allem auch über das Internet ermöglicht.

ORBs sind die **technischen Implementationen** des Standards CORBA. Ein ORB ermöglicht es einem Client, eine Meldung transparent an ein Serverobjekt zu senden, wobei das Serverobjekt auf derselben oder einer anderen Maschine laufen kann. Der **ORB** ist dafür zuständig, das **Serverobjekt zu finden**, dort die **Funktion aufzurufen**, die **Parameter**

zu übergeben und das **Resultat** an den Client **zurückzureichen**. Dadurch wird die bereits erwähnte nahtlose Interoperabilität zwischen Applikationen erreicht, welche in einem völlig heterogenen Umfeld betrieben werden können.

Bisher wurde in einem heterogenen Umfeld typischerweise jede Schnittstelle spezifisch programmiert. Dabei müssen Plattform, Betriebssystem, Programmiersprache und anderes in Betracht gezogen werden. Bei Änderungen ist die Anpassung solcher Schnittstellen entsprechend schwerfällig.

Bei CORBA-basierenden Systemen ist dies anders. Es besteht eine strikte **Trennung** zwischen der **Schnittstellendefintion** eines Objektes und deren **Implementation**. Beim CORBA-Vorgehen wird zunächst die öffentliche Schnittstelle eines Objektes (d.h. die Funktionen) in der Interface Definition Language (**IDL**) definiert. IDL ist eine **implementa-tions-unabhängige** Beschreibungssprache. In einem zweiten Schritt erst wird diese Definition ausprogrammiert, und zwar sowohl für den Client als auch für den Server-Teil.

Dabei kann der Client beispielsweise in Java implementiert werden, während der Server in C++ programmiert wird. CORBA-IDL kann aber nicht nur auf Implementationssprachen "gemapped" werden, sondern auch auf andere Komponentenmodelle wie ActiveX/DCOM. Somit ist die Freiheit gegeben, die am besten passendste Implementations-Strategie zu wählen.

Neben der Kern-Architektur sind eine Reihe von zusätzlichen Services definiert worden, welche CORBA zu einem vollständigen **Middleware** wachsen ließen. Nur um die wichtigsten zu nennen:

- **Naming-Service**: hilft bei der Suche und Identifikation von Objekten
- **Event-Service**: definiert die Schnittstellen für Messaging-Systeme
- **Transaktions-Service**: Schnittstellen für Two-Phase-Commit und andere Verfahren
- **Datenbank-Services**: einheitliche Schnittstellen für Datenbankoperationen

Wie schon beim CORBA-Kern definiert OMG vorerst einen Standard. Typischerweise 1 - 2 Jahre später sind dann die ersten Implementationen auf dem Markt erhältlich. Da praktisch alle Service-Definitionen 1995 verabschiedet worden sind, liegen für die meisten von ihnen schon Implementationen verschiedener Anbieter vor.

2. Programmieren mit CORBA und C++

Die CORBA Implementierung kümmert sich um das Konvertieren und Verpacken der Daten, die an die Methode übergeben worden, um das Verschicken des Methodenaufrufes über das Netzwerk an das Zielobjekt, um das dortige Entpacken und Rückkonvertieren der Daten und den eigentlichen Methodenaufruf auf dem Zielobjekt.

Wenn die entsprechende Methode Daten an den Aufrufer zurückliefern soll, dann geschieht dies in umgekehrter Richtung ab. Durch diesen - vor dem Anwender versteckten - Ablauf ist es ohne weiteres möglich, zum Beispiel in einem unter Linux laufenden Javaprogramm ein Objekt zu verwenden, welches in C++ implementiert wurde und auf einem Windows NT Rechner residiert.

CORBA ist nur ein Standard - keine Implementierung. Es gibt jedoch diverse Implementierungen dieser Architektur von den unterschiedlichsten Herstellern. Für die in diesem Teil enthaltenen Beispielprogramme wurde die Version 2.3.3 von MICO verwendet. MICO ist eine voll funktionsfähige und *frei verfügbare* CORBA Implementierung unter dem GNU Copyleft.

Die Benutzung von CORBA (mit MICO) soll anhand einer kleinen **verteilten Applikation zur Verwaltung von Telefonnummern** demonstriert werden.

2.1. Beispielanwendung

Die Beispielanwendung besteht aus einem Server, der Paare von Namen und dazugehöriger Telefonnummer speichert, und aus Clients, die zum Eintragen neuer Namen/Nummern sowie zur Recherche dienen.

Der Server stellt eine Klasse zur Verfügung, die

- die Telefonnummern speichert (Eigenschaft `_numbers`) und
- Methoden zum Hinzufügen von neuen Telefonnummern (`addEntry()`) und Abfragen von Telefonnummern (`searchEntry()`).

Zunächst zeigt der folgende Auszug die Anwendung, die entwickelt wird.

Start des Servers:

```
$ cabsrv_co &
[4] 24940
$
```

Hinzufügen von Telefonnummern und Abfrage:

```
$ cabadd_co as 123
$ cabadd_co ila 12345
$ cabsearch_co ila
12345
$
```

Die Anwendung wird von einer einfachen Implementierung, bei der Client und Server auf dem selben Rechner laufen müssen weiter entwickelt, bis zu einer verteilten Anwendung, bei der Client und Server auf unterschiedlichen Rechnern laufen können und einen CORNBA Namensdienst verwenden, der die verfügbaren Objektreferenzen verwaltet.

2.2.Vorgehensweise beim Entwurf

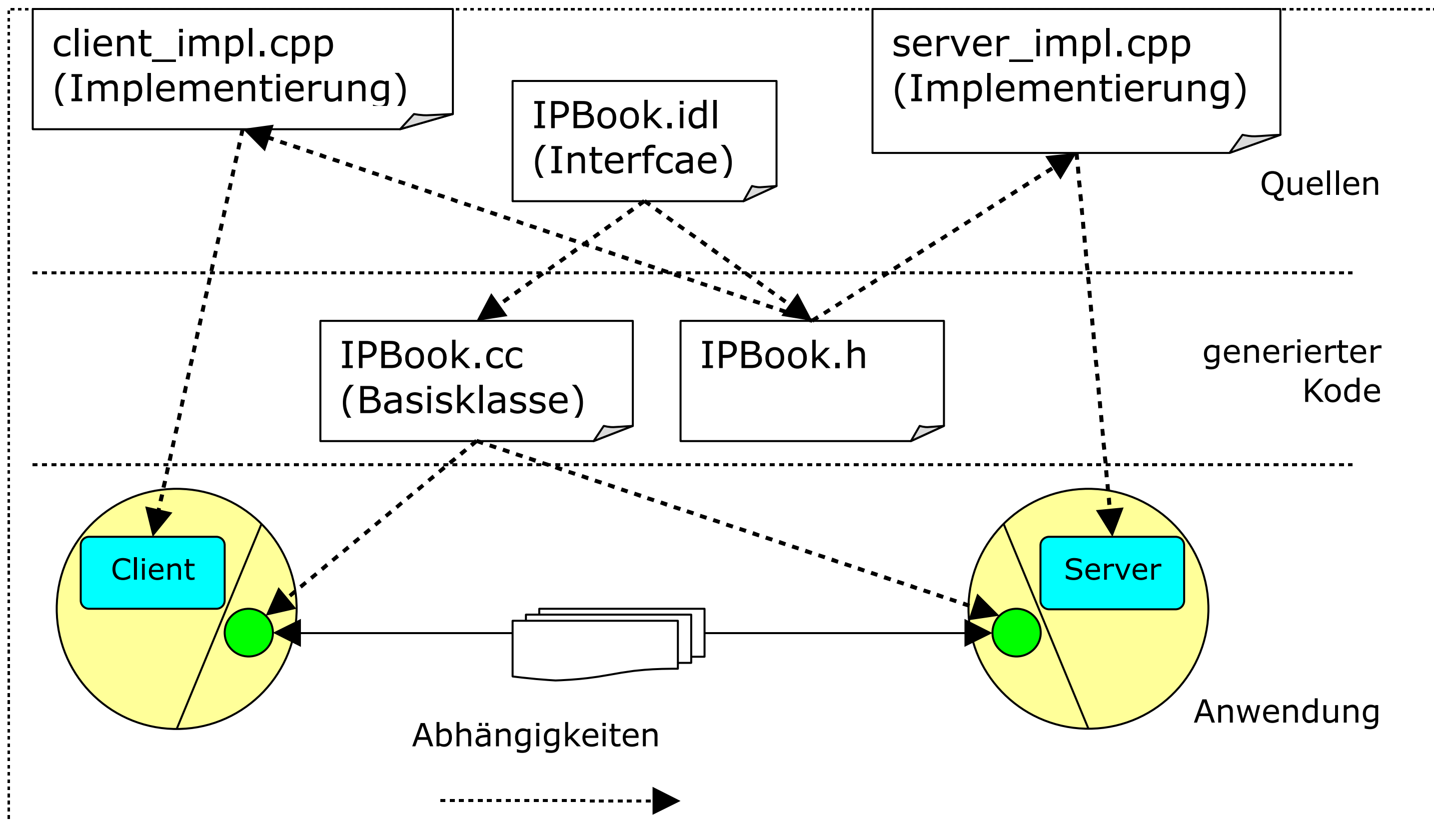
Zunächst wird aus einer **Beschreibung der Interfaces** vom IDL Compiler des Entwicklungssystems ein großer Teil der **Implementierung** des **Serverobjektes** erzeugt.

Unter anderem wird eine **Klasse** zur Verfügung gestellt, die als **Basisklasse** für das **Serverobjekt** dient.

Zu implementieren bleibt dann der Client, der die erzeugte Klasse verwenden kann und die Implementierung der Serveraktivitäten.

Dadurch wird die **Kommunikation automatisch** in die erzeugten Quellprogramme „hinein **generiert**" – dies ist der Vorteil der Verwendung von CORBA aus Entwicklersicht: Der **Entwickler** konzentriert sich auf die **Anwendungslogik**, die Verwaltung der Objekte mach CORBA.

Die folgende Abbildung skizziert die Vorgehensweise, die exakte Beschreibung wird schrittweise im Folgenden am Beispiel der Telefonanwendung verdeutlicht:



Ausgehend von einer IDL Datei werden im o.a. Beispiel C++ Dateien erzeugt. Dabei wird ein IDL-C++ Generator verwendet. Für andere Sprachen (Java, ADA, Smaltalk, Cobol) sind ebenfalls Generatoren verfügbar.

2.3.Definition der Interfaces

Der erste Schritt beim Erstellen einer CORBA Anwendung ist die **Definition aller benötigten Interfaces**.

Über diese Interfaces tauschen Client und Server später ihre Daten aus. Die Interfacedefinitionen erfolgen dabei in einer besonderen Sprache, der *Interface Definition Language* (**IDL**). Diese ähnelt in ihrer Syntax stark einer C++ Klassendefinition.

Die angestrebte Beispielapplikation kommt mit einem einzigen, einfachen Interface namens ***IPBook*** (Internet Phone Book) aus. Dieses enthält zwei Methoden:

- **Hinzufügen** eines neuen Telefonbucheintrages und
- **Suchen** nach Namen.

Die in IDL geschriebene Interfacedefinition wird in eine Datei *IPBook.idl* gespeichert und sieht folgendermaßen aus:

```
$ cat IPBook.idl
#ifndef _IPBook_idl
#define _IPBook_idl

/**
 * Interface for a very simple phone book
 **/
interface IPBook {
    /* Add an entry */
    void addEntry( in string name, in string number);

    /* Search for an entry */
    string searchEntry( in string name);
};

#endif
$
```

Der wichtigste **Unterschied zu einer C++ Klassendefinition** ist, dass bei den **Methodenparametern die Richtung der Datenübergabe** angegeben werden muss. Dies erfolgt aus Sicht des Objektanbieters (Server).

Die verwendete Spezifikation **in** bedeutet, dass Daten ausschließlich vom Aufrufenden zum Objekt transportiert werden, analog bedeutet **out** dann einen Datentransport zurück zum Aufrufenden und **inout** einen bidirektionalen Datentransfer. Ein Returnvalue (wie string bei searchEntry) bedeutet natürlich immer, dass Daten zum Aufrufer zurück übertragen werden, **out** muss hier nicht angegeben werden.

Die folgende Aufstellung skizziert die Sprache IDL:

- Datentypen:
  - Grundtypen
    float, double, short, long, unsigned short, unsigned long, char, unsigned char, octed, Boolean, string
  - Konstrukte:
    struct, union, enum, sequence, exception
- Operationsdeklarationen:
  - void *beep*();
  - boolean *isPositiv*(in long *aValue*);
  - oneway void *doCall*(in sequence<string> *names);*
  - void *doSort*(inout sequence<short> *valueList*);

Zu beachten ist generell bei CORBA Interfaces:

Jeder Methodenaufruf bewirkt eine Netzwerkübertragung! Gerade wenn nur wenige Bytes an Nutzdaten übertragen werden, entsteht ein beträchtlicher Overhead. Hüten muss man sich an dieser Stelle vor Theoretikern des objektorientierten Entwurfs, die alles und jedes in Objekte verpacken.

So wäre es in unserer Beispielapplikation vom objektorientierten Standpunkt durchaus sinnvoll, ein zusätzliches Interface *Entry* mit Methoden wie setName(), getName(), setNumber() und so weiter zu definieren. Die Funktion addEntry() in IPBook würde dann ein Entry Objekt als Parameter bekommen. Resultat dieses Designs wäre die dreifache Anzahl von entfernten Methodenaufrufen, die alle über das Netz gehen. Ein absoluter Performancekiller!

Beim Entwurf von CORBA Interfaces gilt also:

- **Minimiere** die **Anzahl** von **Interfaces**!
- Statt n Methodenaufrufen mit einem Parameter **besser ein Methodenaufruf mit n** Parametern!
- Versuche **zusammengehörige** Daten besser in **Strukturen** oder Felder als in Objekte (= Interfaces) zu packen!

2.4.Implementierung des Serverobjektes

Nach der Erstellung der Interfacedefinition ist der nächste Schritt die ***Implementierung der Serverobjekte*** *(**Servants**)*.

Ein Serverobjekt muss

- einerseits die **Applikationslogik** implementieren - die Funktionen addEntry() und searchEntry() müssen mit Leben gefüllt werden;
- andererseits muss das Serverobjekt auch die **Verbindung zum Object Request Broker**, der für die Vermittlung der verteilten Funktionsaufrufe verantwortlich ist, herstellen.

Für diese Verbindung gibt es verschiedene **Objekt *Adapter***. Der ältere ist der ***Basic Object Adapter*** (**BOA**). Seit dem CORBA Standard 2.2 gibt es den umfangreicheren und flexibleren ***Portable Object Adapter*** (**POA**). Obwohl MICO in der aktuellen Version beide Adapter zur Verfügung stellt, soll im Folgenden ausschließlich auf die Verwendung des modernen **POA** eingegangen werden.

Bei der Implementierung der Serverobjekte nimmt der in MICO enthaltene IDL Prozessor dem Programmierer eine Menge Arbeit ab. Der Aufruf des IDL Prozessors erfolgt an der Kommandozeile, zuerst sollte man hier mittels

```
. /opt/mico/lib/mico-setup.sh
```

die korrekte Umgebung einstellen (dies sollte im .profile bereits enthalten sein).

Der Aufruf des idl Compilers erzeugt aus der Interfacedefinitionsdatei IPBook.idl die C++ Dateien

- *IPBook.h* und
- *IPBook.cpp*.

```
$ idl --poa IPBook.idl
$ ls -l
insgesamt 24
-rw-rw-r--    1 as        users         8295 Jan 30 11:46 IPBook.cc
-rw-rw-r--    1 as        users         3014 Jan 30 11:46 IPBook.h
-rwxr-----    1 as        users          265 Jan 30 11:36 IPBook.idl
$
```

Diese Dateien enthalten bereits einen großen Teil der Implementierung des Serverobjektes. Unter anderem stellen diese Dateien eine **Klasse *POA_IPBook*** zur Verfügung, die als **Basis-klasse für das Serverobjekt** dient. Öffnet man mit einem Texteditor die Datei *IPBook.h* und sucht in ihr nach der Definition der Klasse *POA_IPBook*, so findet man unter anderem die Deklaration der pur virtuellen Funktionen

```
$ cat IPBook.h
…
/*
 * Base class and common definitions for interface IPBook
 */
class IPBook :
  virtual public CORBA::Object
{
  public:
    virtual ~IPBook();

    #ifdef HAVE_TYPEDEF_OVERLOAD
    typedef IPBook_ptr _ptr_type;
    typedef IPBook_var _var_type;
    #endif

    static IPBook_ptr _narrow( CORBA::Object_ptr obj );
    static IPBook_ptr _narrow( CORBA::AbstractBase_ptr obj );
    static IPBook_ptr _duplicate( IPBook_ptr _obj )
    {
      CORBA::Object::_duplicate (_obj);
      return _obj;
    }

    static IPBook_ptr _nil()
    {
      return 0;
```

```
    }

    virtual void *_narrow_helper( const char *repoid );

    virtual void addEntry( const char* name, const char* number ) = 0;
    virtual char* searchEntry( const char* name ) = 0;

  protected:
    IPBook() {};
  private:
    IPBook( const IPBook& );
    void operator=( const IPBook& );
};
…
$
```

Dies sind die Methoden, die in der Interfacedefinition spezifiziert wurden. Aus dem IDL Datentyp **string** ist jetzt nur der C++ Datentyp char* geworden, das IDL Attribut in findet seine Entsprechung in der const Spezifikation des Funktionsparameters.

Die vom Programmierer noch zu leistende Arbeit beschränkt sich nun darauf,

- eine neue Klasse zu schreiben, die von *POA_IPBook* abgeleitet ist und mindestens die beiden pur virtuellen Funktionen addEntry() und searchEntry() implementiert.

Es hat sich eingebürgert, diese Klasse mit dem Nachsatz *_impl* zu benennen, im Beispiel also *IPBook_impl*. Die Definition der Klasse steht also in der Datei *IPBook_impl.h*:

```
$ cat IPBook_impl.h
#ifndef IPBook_impl_h
#define IPBook_impl_h 1

#include "IPBook.h"

#include <map>
#include <string>

class IPBook_impl : virtual public POA_IPBook
{
public:
    // implement pure virtual functions from POA_IPBook
    virtual void addEntry( const char* name, const char* number );
    virtual char* searchEntry( const char* name );

private:
    map <string, string, less<string> > _numbers;
};

#endif
$
```

Neben den notwendigen Funktionen addEntry() und searchEntry() hat die Klasse noch die **private Membervariable _numbers, die zur Aufnahme der Daten des Telefonbuches**

dient. Üblich sind an dieser Stelle auch die Definitionen eigener Konstruktoren und des Destruktors. Dieses relativ einfache Beispiel kommt jedoch mit den vom C++ Compiler zur Verfügung gestellten Standard-Konstruktor und -Destruktor aus.

Die eigentliche **Implementierung der Klasse** in der Datei *IPBook_impl.cpp* sollte nun komplikationslos über die Bühne gehen. Die folgende Datei ist also zu erstellen:

```
$ cat IPBook_impl.cpp
#include <CORBA.h>
#include "IPBook_impl.h"

void IPBook_impl::addEntry( const char* name, const char* number)
{
    string nam = name;
    string num = number;

    _numbers[nam] = num;
}

char* IPBook_impl::searchEntry( const char* name )
{
    map <string, string, less<string> >::iterator r;
    r = _numbers.find( name);
    if (r != _numbers.end()) {
        return CORBA::string_dup( (*r).second.c_str());
    }
    else {
        return CORBA::string_dup( "???");
    }
}
$
```

Wichtig ist:

- Die **Übergabeparameter** vom Typ **char*** belegen Speicherplatz auf dem **Heap**. Dieser wird vom Aufrufer der Funktion, also dem ORB, allokiert. Nach Beendigung der Funktion wird dieser Speicherbereich vom ORB wieder freigegeben. Dann dürfen im Serverobjekt keinerlei Referenzen mehr auf diesen Speicher existieren! Deshalb werden die Parameter name und number zunächst in die C++ Strings nam und num *kopiert*, bevor sie in die Map _numbers eingefügt werden (dabei wird dann zwar nochmal kopiert, jedoch liegen nam und num ja auf dem Stack und werden beim Verlassen der Funktion zerstört).
- searchEntry() gibt einen Zeiger auf einen Speicherbereich vom Typ **char*** an den ORB zurück. Damit hier keine Speicherlecks entstehen können, wurde im CORBA Standard festgelegt, dass der **ORB** für die **Freigabe** dieses Speicherbereiches verantwortlich ist. Somit muss der **Programmierer beachten**, dass er auf dem **Heap liegenden Speicher zurückliefert**. Dafür stellt CORBA die Funktion *CORBA::string_dup()* zur Verfügung.

Nun können die bisher erstellten Module kompiliert werden:

```
$ mico-c++ -c IPBook.cc
$ mico-c++ -c IPBook_impl.cpp
$ ls -al
$ ls -l
-rw-rw-r--    1 as        users     8295 Jan 30 11:46 IPBook.cc
-rw-rw-r--    1 as        users     3014 Jan 30 11:46 IPBook.h
-rwxr-----    1 as        users      265 Jan 30 11:36 IPBook.idl
-rwxr-----    1 as        users      480 Jan 30 12:00 IPBook_impl.cpp
-rwxr-----    1 as        users      408 Aug  9 09:56 IPBook_impl.h
-rw-rw-r--    1 as        users    83488 Jan 30 12:08 IPBook_impl.o
-rw-rw-r--    1 as        users    76116 Jan 30 12:06 IPBook.o
$
```

Die Aufrufe des mico-c++ Compilers erzeugen die Objektmodule IPBook.o und IPBook_impl.o.

Um diese zum Leben zur erwecken, bedarf es noch der entsprechenden Server- und Clientapplikationen – diese Objektdateien stellen die Verbindungslogik des Opjektadapters (POA) dar.

2.5.Die Serverapplikation

Zur Erstellung eines kompletten CORBA-Serverprogramms fehlt nur noch die entsprechende *main()* Funktion. Diese wird sinnvollerweise in ein eigenes Sourcefile, zum Beispiel mit dem Namen ***cabsrv_co.cpp*** geschrieben. Zunächst werden die Headerfiles inkludiert - absolut notwendig sind hier das von MICO gelieferte *CORBA.h* sowie die Deklaration des Servants in *IPBook_impl.h*:

```
$ cat cabsrv_co.cpp
#include <CORBA.h>
#include "IPBook_impl.h"

#include <fstream.h>

int main( int argc, char **argv)
{
    int rc = 0;
```

Als nächstes müssen der **ORB und ein POA-Manager** (POA: Portable Object Adapter) **erzeugt und initialisiert** werden. Für dieses einfache Beispiel reicht es, den vom System zur Verfügung gestellten Root-POA-Manager zu verwenden. Alle CORBA Systemaufrufe werden in einem try-catch Block gekapselt:

```
    try {
        // init ORB and POA Manager
        CORBA::ORB_var orb = CORBA::ORB_init( argc, argv);
        CORBA::Object_var poaobj =
                      orb -> resolve_initial_references("RootPOA");
        PortableServer::POA_var poa =
                      PortableServer::POA::_narrow( poaobj);
        PortableServer::POAManager_var mgr = poa -> the_POAManager();
```

Nun kann der **Servant erzeugt und aktiviert** werden. CORBA stellt mehrere Aktivierungsmethoden zur Verfügung, hier wird die *implizite Aktivierung* verwendet:

```
        // create a new instance of the servant
        IPBook_impl *impl = new IPBook_impl;
        // activate the servant
        IPBook_var f = impl -> _this();
```

Jedes **CORBA-Serverobjekt** benötigt eine **eindeutige Identifikation**, damit ein Client auch den »richtigen« Servant finden kann. Diese Identifikation wird als ***IOR*** (**Interoperable Object Reference**) bezeichnet.

Seit CORBA 2.0 sind die ORB verschiedener Herstelle interoperable, u.a. dank folgender Spezifikationen:

- IIOP (Internet Inter ORB Protocol)
  Verbindung zwischen ORBs, Protokoll auf TCP/IP Basis zum Transport von Request/Replay und IDL Dateien
- **IOR** (**I**nteroperable **O**bject **R**eference)
  ORB übergreifende Objektreferenz. Der Aufbau garantiert weltweit eindeutige Identifizierung von Objekten. Eine Referenz beinhaltet: IP-Adresse, Port, Objekttyp und Objektschlüssel.

Um diese IOR vom Server zum Client transportieren zu können, gibt es mehrere Möglichkeiten. Das einfachste (und portable) Verfahren ist, die **IOR** in eine **Zeichenkette** umzuwandeln und diese in einer **Datei** zu speichern, auf die Server und Client gleichermaßen Zugriff haben (in den nächsten Kapiteln wird noch ausführlich auf bessere Alternativen dazu

eingegangen):

```
        // save the stringified object reference to file
        CORBA::String_var s = orb -> object_to_string( f);
        ofstream out( "IPBook.ref");
        out << s << endl;
        out.close();
```

Nun sind alle notwendigen Vorbereitungen abgeschlossen, jetzt kann der **POA Manager aktiviert und der ORB gestartet** werden. Damit wird der Servant endgültig zum Leben erweckt:

```
        // activate POA manager
        mgr -> activate();
        // run the ORB
        orb -> run();
```

Aus diesem Funktionsaufruf kehrt das Programm normalerweise nicht zurück, es sei denn, dass ein explizites Shutdown des ORBs ausgeführt wird. In diesem Falle sollten dann noch POA-Manager und Servant korrekt aufgeräumt werden:

```
        poa -> destroy( TRUE, TRUE);
        delete impl;
        rc = 0;
    } //try
```

Der Rest der *main()* Funktion befasst sich nun noch mit dem Auffangen von Exceptions aus dem CORBA System und der Fehlerausgabe:

```
    catch(CORBA::SystemException_catch& ex)
    {
        ex -> _print(cerr);
        cerr << endl;
        rc = 1;
    }
    return rc;
}
```

Das gesamte Programm im Überblick:

```
$ cat cabsrv_co.cpp
$ #include <CORBA.h>
#include "IPBook_impl.h"
#include <fstream.h>

int main( int argc, char **argv)
{
    int rc = 0;

    try {
        // init ORB and POA Manager
        CORBA::ORB_var orb = CORBA::ORB_init( argc, argv);
        CORBA::Object_var poaobj = orb -> resolve_initial_references("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow( poaobj);
        PortableServer::POAManager_var mgr = poa -> the_POAManager();

        // create a new instance of the servant
        IPBook_impl *impl = new IPBook_impl;
        // activate the servant
        IPBook_var f = impl -> _this();

        // save the stringified object reference to file
        CORBA::String_var s = orb -> object_to_string( f);
        ofstream out( "IPBook.ref");
        out << s << endl;
        out.close();
```

```
        // activate POA manager
        mgr -> activate();
        // run the ORB
        orb -> run();

        poa -> destroy( TRUE, TRUE);
        delete impl;
        rc = 0;
    }
    catch(CORBA::SystemException_catch& ex)
    {
        ex -> _print(cerr);
        cerr << endl;
        rc = 1;
    }
    return rc;
}
$
```

Damit ist unsere erste CORBA-Serverapplikation fertig programmiert, sie kann nun mittels des mico Compiler übersetzt werden.

```
$ mico-c++ -c cabsrv_co.cpp
$ ls -l
-rwxr-----    1 as       users     1157 Jan 30 12:44 cabsrv_co.cpp
-rw-rw-r--    1 as       users    59296 Jan 30 12:45 cabsrv_co.o
-rw-rw-r--    1 as       users     8295 Jan 30 11:46 IPBook.cc
-rw-rw-r--    1 as       users     3014 Jan 30 11:46 IPBook.h
-rwxr-----    1 as       users      265 Jan 30 11:36 IPBook.idl
-rwxr-----    1 as       users      480 Jan 30 12:00 IPBook_impl.cpp
-rwxr-----    1 as       users      408 Aug  9 09:56 IPBook_impl.h
-rw-rw-r--    1 as       users    83488 Jan 30 12:08 IPBook_impl.o
-rw-rw-r--    1 as       users    76116 Jan 30 12:06 IPBook.o
$
```

Abschließend müssen die bisher entstandenen Objektmodule zusammen mit der MICO-Library gelinkt werden, um ein ausführbares Programm zu erhalten:

```
$ mico-ld -o cabsrv_co cabsrv_co.o IPBook.o IPBook_impl.o -lmico2.3.6
$ ls -l
insgesamt 384
-rwxrwxr-x    1 as       users   113720 Jan 30 12:50 cabsrv_co
-rwxr-----    1 as       users     1157 Jan 30 12:44 cabsrv_co.cpp
-rw-rw-r--    1 as       users    59296 Jan 30 12:45 cabsrv_co.o
-rw-rw-r--    1 as       users     8295 Jan 30 11:46 IPBook.cc
-rw-rw-r--    1 as       users     3014 Jan 30 11:46 IPBook.h
-rwxr-----    1 as       users      265 Jan 30 11:36 IPBook.idl
-rwxr-----    1 as       users      480 Jan 30 12:00 IPBook_impl.cpp
-rwxr-----    1 as       users      408 Aug  9 09:56 IPBook_impl.h
-rw-rw-r--    1 as       users    83488 Jan 30 12:08 IPBook_impl.o
-rw-rw-r--    1 as       users    76116 Jan 30 12:06 IPBook.o
$
```

Jetzt kann cabsrv_co über die Kommandozeile gestartet werden. Wenn kein Fehler auftritt, läuft das Programm ohne weitere Ausgabe.

```
$ cabsrv_co &
 21285
$
```

Um überhaupt etwas Sinnvolles tun zu können, benötigt man noch Programme, die zumindest das Neuanlegen von Telefonbucheinträgen und eine Recherche darin erlauben.

2.6.CORBA Clients

Als erstes soll ein Programm erstellt werden, welches das Anlegen von Einträgen im Telefonbuch erlaubt. Eine Eingabe an der Kommandozeile in der Form

```
$ cabadd_co name nummer
```

soll einen Eintrag mit dem Namen *name* und der Nummer *nummer* dem Telefonbuch hinzufügen. Dazu wird die **Programmdatei *cabadd_co.cpp*** erstellt. Hier muss zunächst auch wieder *CORBA.h* inkludiert werden. Im Unterschied zum Serverprogramm darf hier aber die Deklaration des **Servants nicht** inkludiert werden, **stattdessen wird das vom IDL-Preprozessor generierte Stubfile *IPBook.h*** verwendet:

```
$ cat cabadd_co
#include <CORBA.h>
#include "IPBook.h"

#include <fstream.h>
….
```

Im Hauptprogramm wird als erstes der ORB initialisiert und dann ein **Test auf das Vorhandensein der Kommandozeilenargumente** durchgeführt. Diese Reihenfolge ist sinnvoll, da es so möglich ist, zusätzliche Parameter für den ORB mit auf der Kommandozeile zu übergeben (einige dieser Parameter werden wir im nächsten Abschnitt noch kennen lernen).

Die Funktion **CORBA::ORB_init(argc, argv)** wertet diese **ORB-spezifischen Parameter** aus und passt argc und argv entsprechend an. Der darauf folgende Test kann sich dann auf die Abfrage der programmspezifischen Kommandozeilenargumente beschränken:

```
int main( int argc, char **argv)
{
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv);

    int rc = 0;
    if (argc != 3) {
        cerr << "usage: " << argv[0] << " name number\n";
        exit(1);
    }
```

Das Clientprogramm benötigt außerdem die Information, **welchen CORBA-Server** es verwenden soll. Schließlich können in einer umfangreichen CORBA-Applikation durchaus mehrere hundert Servants ihren Dienst tun! Wir erinnern uns: Bei der Programmierung des Servers im vorhergehenden Teil wurde veranlasst, dass die *IOR* des Servants in eine Zeichenkette umgewandelt und in die **Datei IPBook.ref** geschrieben wird. Diese Datei wird nun vom Clientprogramm eingelesen:

```
    ifstream in( "IPBook.ref");
    char s[1000];
    in >> s;   //IOR
    in.close();
```

Die „stringifizierte" IOR (Interoperable Object Reference) steht jetzt also in der Variablen s. Diese kann nun zur Erstellung eines Objektes vom Typ *IPBook_var* verwendet werden. Zum Abfangen eventueller Fehler bei diesem Vorgang wird ein try-catch Block geöffnet:

```
    try {
        CORBA::Object_var obj = orb -> string_to_object( s);
        IPBook_var f = IPBook::_narrow( obj);
```

Mit dem nun vorliegenden Objekt f vom Typ *IPBook_var* können sämtliche Operationen durchgeführt werden, die ursprünglich in der Interfacedefinition ***IPBook.idl*** festgelegt worden sind.

Das CORBA-Laufzeitsystem und unsere Vorarbeiten bewirken nun, dass alle auf dem Objekt f (dem *Clientstub*, Stub = Stummel) ausgeführten Operationen automatisch auf den *Servant*, das heißt auf das im Adressbereich des Serverprogramms *cabsrv_co* liegende Objekt vom Typ *IP-Book_impl* weitergeleitet werden.

Das das Serverprogramm muss vorher dazu gestartet sein, bevor der Client aufgerufen werden kann (es gibt zwar die Möglichkeit, per Implementation-Repository eine Serveraktivierung *on demand* auszuführen, jedoch geht dies über die Thematik dieser „Einführung" in CORBA hinaus).
Wir rufen jetzt also die Methode *add* mit den beiden Argumenten *name* und *nummer* von der Kommandozeile auf:

```
        f -> addEntry( argv[1], argv[2]);
```

Der Rest des Programms ist dem Auffangen von Exceptions und der Ausgabe ihrer Ursache gewidmet:

```
}
    catch(CORBA::SystemException_catch& ex)
    {
        ex -> _print(cerr);
        cerr << endl;
        rc = 1;
    }

    return rc;
}
```

Nun muss das erste Client Programm übersetzt und gebunden werden.

```
$ mico-c++ -c cabadd_co.cpp
$ mico-ld -o cabadd_co cabadd_co.o IPBook.o  -lmico2.3.6
$
```

Das Servant-Objekt IPBook_impl.o braucht hier im Gegensatz zum Serverprogramm nicht dazugelinkt werden, da wir ja einen Client programmiert haben.

Als zweites benötigen wir noch ein Programm, um nach Namen im Telefonbuch suchen zu können. Die Eingabe

```
$ cabsearch_co name
```

soll die zu *name* gehörende Telefonnummer vom Server erfragen. Der Quelltext des Programmes (*cabsearch_co.cpp*) ist analog zum ersten Client-Programm:

```
$ cat cabsearch_co.cpp
#include <CORBA.h>
#include "IPBook.h"

#include <iostream.h>
#include <fstream.h>

int main( int argc, char **argv)
{
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv);

    int rc = 0;
    if (argc != 2) {
        cerr << "usage: " << argv[0] << " name\n";
        exit(1);
    }

    ifstream in( "IPBook.ref");
    char s[1000];
    in >> s;
    in.close();

    try {
        CORBA::Object_var obj = orb -> string_to_object( s);
        IPBook_var f = IPBook::_narrow( obj);

        cout << f -> searchEntry( argv[1]) << endl;
```

```
    }
    catch(CORBA::SystemException_catch& ex)
    {
        ex -> _print(cerr);
        cerr << endl;
        rc = 1;
    }

    return rc;
}
$
```

Jetzt können wir das zweite Clientprogramm übersetzen und linken:

```
$ mico-c++ -c cabsearch_co.cpp
$ mico-ld -o cabsearch_co cabsearch_co.o IPBook.o  -lmico2.3.6
$
```

Nun können wir unsere erste vollständige CORBA-Applikation testen:

```
$ cabsrv_co &
[2] 21804
$ cabadd_co as 123 456
usage: cabadd_co name number
$ cabadd_co as 123
$ cabadd_co "Petra Musterfrau" "+49 011 23456"
$ cabsearch as
bash: cabsearch: command not found
$ cabsearch_co as
123
$ cabsearch_co "Petra Musterfrau"
+49 011 23456
$ cabsearch_co "Petra  musterfrau"
NOT FOUND
$
```

Den Inhalt der IOR-Datei () kann man mit dem Kommando iordump ansehen:

```
$ iordump < IPBook.ref

    Repo Id:  IDL:IPBook:1.0

IIOP Profile
    Version:  1.0
    Address:  inet:linux.ibs:1964
   Location:  corbaloc::linux.ibs:1964//21804/1012403931/%5f0
        Key:  2f 32 31 38 30 34 2f 31 30 31 32 34 30 33 39 33 /21804/101240393
              31 2f 5f 30                                     1/_0

Multiple Components Profile
 Components:  Native Codesets:
              normal: ISO 8859-1:1987; Latin Alphabet No. 1
                wide: ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format
16-bit form
        Key:  (empty)


$
```

Ein Wermutstropfen bleibt: Damit Client und Server sich finden können, musste die Objektreferenz (IOR) des Servants in einer Datei gespeichert werden. Sowohl **Client**- als **auch Serverprogramm** benötigen **Zugriff** auf diese **IOR-Datei**. Solange beide Programme auf *einem* Rechner laufen, sollte dies kein Problem sein. **Was aber, wenn dabei Rechnergrenzen zu überwinden sind?**

Eventuell könnte man sich noch mit der Einrichtung von Netzlaufwerken (z.B. SAMBA- oder NFS-Shares) behelfen, jedoch wird man hier bald an administrative Grenzen stoßen. Daher werden in den folgenden zwei Kapiteln Methoden vorgestellt, um Objektreferenzen auf andere Art und Weise zu transportieren: Die Verwendung des **properitären MICO-Binders** und die **Einbeziehung des CORBA Naming Service**.

2.7.Der MICO-Binder

Die CORBA-Implementierung *MICO* stellt als Hilfe bei der Lokalisation von Serverobjekten den MICO-Binder zur Verfügung.

Diesem liegt folgende Überlegung zugrunde: Um einen Servant in einem Netzwerk eindeutig identifizieren und auffinden zu können, bedarf es nicht unbedingt einer IOR. Alternativ ist es möglich, einen Servant durch seine Netzwerkadresse zu bezeichnen. Netzwerkadressen bestehen in einem TCP/IP Netzwerk aus der IP-Adresse des Rechners (oder seinem Namen) und einer Portnummer. Da IP-Adressen und Hostnamen zentral vergeben werden, ist damit die weltweite Eindeutigkeit der Kombination IP-Adresse:Portnummer gewährleistet. Falls ein CORBA-Serverprogramm mehrere Objekte der Außenwelt zur Verfügung stellt, können diese durch den Typ des Servant (genauer gesagt durch die Repository-Id der Interfacedefinition) unterschieden werden. Wenn dann noch mehrere Servants des gleichen Typs existieren, dann muss der Programmierer diesen noch einen (je Serverprogramm) eindeutigen Namen verpassen.

Da unser in im vorletzten Kapitel erstellter Telefonbuchserver *cabsrv_co* nur ein einziges CORBA-Objekt noch außen liefert, kann das Programm ohne Veränderung für die Benutzung des MICO-Binders verwendet werden. Es muss lediglich sichergestellt werden, dass der Servant beim Starten immer die gleiche Portnummer benutzt. Dies erfolgt durch Angabe der Kommandozeilenoption

-ORBIIOPAddr inet:*hostname*:*port*.

Ohne diese Option würde sich der Server bei jedem Start an irgendeine freie Portnummer binden. Der Aufruf

```
$ cabsrv_co -ORBIIOPAddr inet:localhost:4500
$
```

bewirkt also, dass der Server auf Port 4500 des Loopback-Netzwerkinterfaces auf Verbindungen wartet.

Am Serverprogramm ist also keine Änderung erforderlich.

Um mit unseren beiden Clients den MICO-Binder verwenden zu können, sind noch einige Änderungen an deren Sourcecode nötig. Im wesentlichen muss der Aufruf von

`string_to_object(»stringified ior«)`

durch

`bind(»repository id«,»address«)`

ersetzt werden.

Die `»repository id«` kann man mit dem iordump-Kommando aus der Datei ipBook.ref ersehen:

```
as@linux> iordump < IPBook.ref

    Repo Id:  IDL:IPBook:1.0

IIOP Profile
    Version:  1.0
    Address:  inet:localhost.fbi.fh-darmstadt.de:4500
   Location:  corbaloc::localhost.fbi.fh-
darmstadt.de:4500//18820/1021211326/%5f0
        Key:  2f 31 38 38 32 30 2f 31 30 32 31 32 31 31 33 32 /18820/102121132
              36 2f 5f 30                                     6/_0

Multiple Components Profile
 Components:  Native Codesets:
              normal: ISO 8859-1:1987; Latin Alphabet No. 1
                wide: ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format
16-bit form
        Key:  (empty)


as@linux>
```

Die Adresse sollte im Programm als Kommandozeilenargument übergeben werden, um einfach zwischen verschiedenen Servern umschalten zu können.

Exemplarisch sei im folgenden das modifizierte Programm *cabadd_co.cpp*, welches wir jetzt *cabadd_mi.cpp* nennen, vorgestellt.

Zunächst erfolgt wieder das Inkludieren der notwendigen Header und der Test auf das Vorhandensein der Kommandozeilenargumente. Es muss jetzt als viertes Argument die Adresse des CORBA-Servers mit übergeben werden:

```
$ cat cabadd_mi.cpp
#include <CORBA.h>
#include "IPBook.h"

#include <iostream.h>

int main( int argc, char **argv)
{
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv);

    int rc = 0;
    if (argc != 4) {
        cerr <<"usage: "<<argv[0]<<" name number server_address \n";
        exit(1);
    }
```

Nun kommt die entscheidende Stelle: Anstelle der Verwendung von string_to_object() wird direkt bind() aufgerufen. Der erste Parameter ist die Repository-ID unseres Servants, der zweite die per Kommandozeile übergebene Netzwerkadresse des Servers:

```
    try {
        CORBA::Object_var obj = orb->bind( "IDL:IPBook:1.0", argv[3]);
        if (CORBA::is_nil( obj)) {
            cerr << "no object at " << argv[3] << " found.\n";
            exit(1);
        }
        IPBook_var f = IPBook::_narrow( obj);
```

Falls alles gut gegangen ist, haben wir jetzt wieder ein *IPBook_var* Objekt, mit dem alle Methoden des Interfaces IPBook ausgeführt werden können. Der verbleibende Rest des Programmes bringt daher nicht Neues:

```
        f -> addEntry( argv[1], argv[2]);
    }
    catch(CORBA::SystemException_catch& ex)
    {
        ex -> _print(cerr);
        cerr << endl;
        rc = 1;
    }

    return rc;
}
```

Die Änderungen am Programm *cabsearch* sind analog und daher ohne besondere Beschreibung nachzuvollziehen.

```
$ cat cabadd_mi.cpp
#include <CORBA.h>
#include "IPBook.h"

#include <iostream.h>

int main( int argc, char **argv)
{
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv);

    int rc = 0;
    if (argc != 4) {
        cerr << "usage: "<<argv[0]<<" name number server_address \n";
        exit(1);
    }

    try {
        CORBA::Object_var obj = orb->bind( "IDL:IPBook:1.0", argv[3]);
        if (CORBA::is_nil( obj)) {
            cerr << "no object at " << argv[3] << " found.\n";
            exit(1);
        }
        IPBook_var f = IPBook::_narrow( obj);
        f -> addEntry( argv[1], argv[2]);
    }
    catch(CORBA::SystemException_catch& ex)
    {
```

```
        ex -> _print(cerr);
        cerr << endl;
        rc = 1;
    }
    return rc;
}
$
```

```
$ cat cabsearch_mi.cpp
#include <CORBA.h>
#include "IPBook.h"

#include <iostream.h>

int main( int argc, char **argv)
{
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv);

    int rc = 0;
    if (argc != 3) {
        cerr << "usage: " << argv[0] << " name server_address\n";
        exit(1);
    }

    try {
        CORBA::Object_var obj = orb->bind( "IDL:IPBook:1.0", argv[2]);
        if (CORBA::is_nil( obj)) {
            cerr << "no object at " << argv[2] << " found.\n";
            exit(1);
        }
        IPBook_var f = IPBook::_narrow( obj);
        cout << f->searchEntry( argv[1]) << endl;
    }
    catch(CORBA::SystemException_catch& ex)
    {
```

```
        ex->_print(cerr);
        cerr << endl;
        rc = 1;
    }
    return rc;
}
$
```

Zum leichteren Übersetzen und Binden wird ab jetzt das make-Kommando verwendet. Dazu existiert ein Makefile, indem definiert ist, wie ein Programm zu übersetzen und zu binden ist.

Aufruf:

```
$ make <Ziel>
```

Dabei ist <Ziel> der Name des zu erzeugenden Programms.

```
$ make cabadd_mi
mico-c++    -c -o cabadd_mi.o cabadd_mi.cpp
mico-ld -o cabadd_mi cabadd_mi.o IPBook.o -lmico2.3.6
$
$ make cabsearch_mi
mico-c++    -c -o cabsearch_mi.o cabsearch_mi.cpp
mico-ld -o cabsearch_mi cabsearch_mi.o IPBook.o -lmico2.3.6
$
```

Die Programme werden wie bekannt übersetzt und gelinkt.

Die Programme werden nun wie folgt gestartet:

```
$ cabsrv_co -ORBIIOPAddr inet:localhost:4500 &
[2] 22008
$ cabadd_mi as 123 inet:localhost:4500
$ cabadd_mi ila 18 inet:localhost:4500
$ cabsearch_mi ila inet:localhost:4500
18
$ cabsearch_mi ila inet:localhost:6666
no object at inet:localhost:6666 found.
$ cabsearch_mi ila inet:trex:4500
no object at inet:trex:4500 found.
$
```

Nun können **Server** und **Client** auch auf **verschiedenen Rechnern** gestartet werden. Dabei muss nur die Angabe *localhost* durch den Netzwerknamen bzw. die IP-Adresse des Servers ersetzt werden.

Die Verwendung des MICO-Binders ermöglicht es dem Anwender, auf den Transport von Objektreferenzen zu verzichten und stattdessen Netzwerkadressen zur Lokalisierung eines CORBA-Servers zu verwenden. Diese Methode dürfte für die meisten Anwendungen vollkommen ausreichend sein; jedoch hat sie noch zwei Nachteile:

- Es handelt sich um eine properitäre Erweiterung des CORBA-Standards. Falls es in einer verteilten Anwendung Objekte gibt, die anstelle von MICO eine andere CORBA-Implementierung benutzen, so wird diese Methode nicht funktionieren.

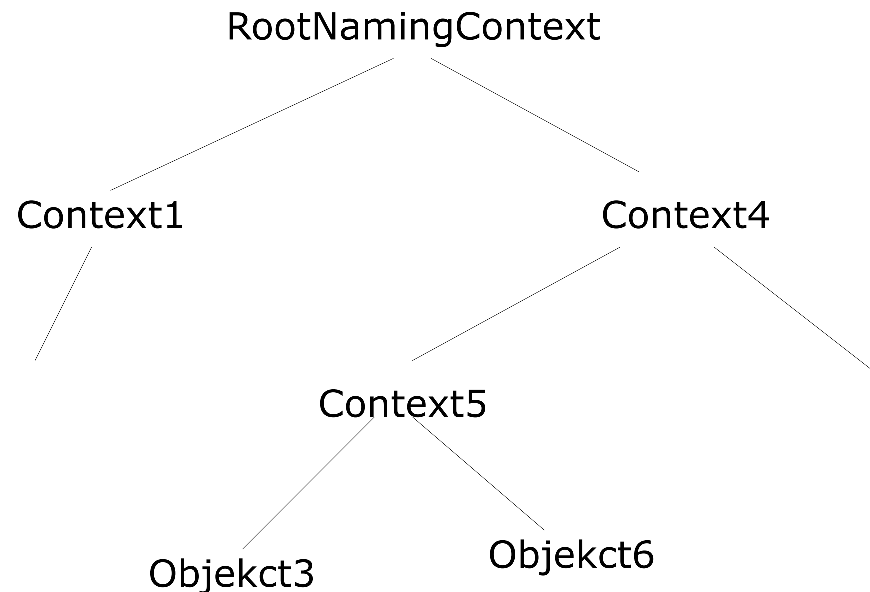
- Wenn eine Applikation auf sehr viele CORBA-Objekte zugreifen muss, so kann die Verwaltung der vielen unterschiedlichen Portnummern kompliziert werden. Schließlich muss ja auch die Information, dass zum Beispiel Port 4500 für die Kommunikation benutzt werden soll, zwischen Client und Server abgestimmt werden.

In diesen Situationen hilft die Verwendung des CORBA-Naming-Service weiter.

2.8.Der CORBA Naming Service

Der Naming Service ist ein Standard CORBA **Dienst**, der für die einfache **Verwaltung** einer Vielzahl von **Objektreferenzen** geeignet ist. Er dürfte in den meisten Implementierungen, wie auch im MICO, zur Verfügung stehen.

Der Naming Service ist ein hierarchisches Verzeichnis, in dem es ***Namenskontexte*** und ***Namenseinträge*** gibt. Einen Namenskontext kann man sich als eine Art Unterverzeichnis vorstellen. Ein Nameneintrag enthält die IOR eines konkreten CORBA-Objektes. Sowohl Namenskontexte als auch Nameneinträge haben einen symbolischen Namen, mittels dessen das Objekt eindeutig lokalisiert werden kann:



Die Clientprogramme brauchen somit nur noch den Namen des Objektes, das sie benutzen wollen, sowie die Adresse des Naming Service zu kennen. Letztere kann per Kommandozeilenparameter `–ORBNamingAddr inet:`*hostname*:*port* an die Anwendung übergeben werden.

Allerdings muss der Softwareentwickler in Server- und Clientapplikation einige Modifizierungen vornehmen, damit die Objektreferenzen über den Naming Service aufgelöst werden können.

Schauen wir uns zunächst den Serverquellcode *cabserv_ns.cpp* an. Die ersten Zeilen sind bis zur Aktivierung des Servants sind fast identisch mit dem bereits erstellten Programm *cabserv_co.cpp*, es muss nur **zusätzlich die Headerdatei CosNaming.h inkludiert** werden:

```
$ cat cabserv_ns.cpp
#include <CORBA.h>
#include <mico/CosNaming.h>

#include "IPBook_impl.h"

int main( int argc, char **argv)
{
    int rc = 0;

    try {
        // init ORB and POA Manager
        CORBA::ORB_var orb =
                      CORBA::ORB_init( argc, argv);
        CORBA::Object_var poaobj =
                      orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa =
                      PortableServer::POA::_narrow( poaobj);
        PortableServer::POAManager_var mgr = poa->the_POAManager();

        // create a new instance of the servant
        IPBook_impl *impl = new IPBook_impl;
        // activate the servant
        IPBook_var f = impl->_this();
```

Nun muss der **Rootkontext des Naming Service aufgelöst** werden. Dazu wird zuerst mittels *resolve_initial_references* die per Kommandozeilenoption „-ORBNamingAddr“ übergebene

Objektreferenz auf den Naming Service besorgt. Anschliessend wird diese auf einen NamingContext »eingeengt«:

```
        // resolve the naming service
        CORBA::Object_var nsobj =
          orb->resolve_initial_references ("NameService");
        if (CORBA::is_nil( nsobj)) {
            cerr << "can't resolve NameService\n";
            exit(1);
        }

        // narrow the root naming context
        CosNaming::NamingContext_var nc =
          CosNaming::NamingContext::_narrow (nsobj);
```

Ausgehend von diesem NamingContext (der ja der *RootNamingContext* ist), kann nun die gesamte Hierarchie des Naming Service durchlaufen werden. In diesem einfachen Beispiel wollen wir uns darauf beschränken, direkt den einfachen **Namenseintrag** »**AddressBook**« anzulegen. Dies erfolgt mittels der Funktion ***bind()***. Diese wirft allerdings eine Exception, falls der Eintrag schon (von einem vorherigen Lauf des Programms) existiert. In diesem Fall muss dann *rebind()* verwendet werden:

```
// create a name entry
CosNaming::Name name;
name.length (1);
name[0].id = CORBA::string_dup ("AddressBook");
name[0].kind = CORBA::string_dup ("");

// bind or rebind the servant to the naming service
try {
    nc->bind (name, f);
}
catch (CosNaming::NamingContext::AlreadyBound_catch &ex) {
    nc->rebind (name, f);
}
```

Der Rest des Programms läuft wie gehabt, lediglich der Code zur Behandlung von Exceptions ist erweitert worden, damit Fehler bei der Angabe der Adresse des Naming Service abgefangen werden:

```
        // activate POA manager
        mgr->activate();
        // run the ORB
        orb->run();

        poa->destroy( TRUE, TRUE);
        delete impl;
    }
    catch(CORBA::ORB::InvalidName_catch& ex)
    {
        ex->_print(cerr);
        cerr << endl;
        cerr << "possible cause: can't locate Naming Service\n";
        rc = 1;
    }
    catch(CORBA::SystemException_catch& ex)
    {
        ex->_print(cerr);
        cerr << endl;
        rc = 1;
    }
    return rc;
}
```

Beim **Linken** des Programmes muss mittels **`-lmicocoss2.3.3`** die Library mit dem Zugriffsroutinen auf den **Naming Service dazugebunden** werden.

Der **Code des Clients *cabadd_ns.cpp* ist analog an die Verwendung des Naming Service anzupassen**, anstelle von *bind()* oder *rebind()* wird hier ***resolve()*** aufgerufen, um an den vom Server angelegten Namenseintrag heranzukommen:

```
#include <CORBA.h>
#include <mico/CosNaming.h>
#include "IPBook.h"

#include <iostream.h>

int main( int argc, char **argv)
{
    // init ORB
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv);

    int rc = 0;
    if (argc != 3) {
        cerr << "usage: " << argv[0] << " name number\n";
        exit(1);
    }

    try {
        // resolve the naming service
        CORBA::Object_var nsobj =
            orb->resolve_initial_references ("NameService");
        if (CORBA::is_nil( nsobj)) {
            cerr << "can't resolve NameService\n";
            exit(1);
        }
        // narrow the root naming context
        CosNaming::NamingContext_var nc =
```

```
            CosNaming::NamingContext::_narrow (nsobj);

        // create a name component
        CosNaming::Name name;
        name.length (1);
        name[0].id = CORBA::string_dup ("AddressBook");
        name[0].kind = CORBA::string_dup ("");

        // resolve the name component with the naming service
        CORBA::Object_var obj = nc->resolve( name);

        // narrow this object to IPBook
        IPBook_var f = IPBook::_narrow( obj);

        // work with IPBook
        f->addEntry( argv[1], argv[2]);
    }
    catch(CORBA::ORB::InvalidName_catch& ex)
    {
        ex->_print(cerr);
        cerr << endl;
        cerr << "possible cause: can't locate Naming Service\n";
        rc = 1;
    }
    catch(CosNaming::NamingContext::NotFound_catch& ex)
    {
        cerr << "Name not found at Naming Service\n";
```

```
        rc = 1;
    }
    catch(CORBA::SystemException_catch& ex)
    {
        ex->_print(cerr);
        cerr << endl;
        rc = 1;
    }

    return rc;
}
```

Bevor die Programme nun ausprobiert werden können, muss als erstes der CORBA (bzw. MICO) Naming Service **nsd** gestartet werden. Damit dieser sich an einen bestimmten Netzwerkport bindet, wird dabei die bekannte Option `-ORBIIOPAddr` verwendet werden:

```
$ nsd -ORBIIOPAddr inet:localhost:5000 &
 25307
```

Nun können unsere Server- und Clientprogramme in Aktion treten; sie benötigen jedoch nun immer die Angabe der Adresse des Naming Service:

```
$ cabsrv_ns -ORBNamingAddr inet:localhost:5000 &
[4] 25309
$
$ ps -f
UID          PID  PPID  CMD
as          3573  3572  -bash
as         25307  3573  nsd -ORBIIOPAddr inet:localhost:5000
as         25309  3573  cabsrv_ns -ORBNamingAddr inet:localhost:5000
as         25346  3573  ps -f
$
$
$ cabadd_ns as 123 -ORBNamingAddr inet:localhost:5000
$ cabadd_ns ila 12345 -ORBNamingAddr inet:localhost:5000
$ cabsearch_ns as  -ORBNamingAddr inet:localhost:5000
123
$ cabsearch_ns as  -ORBNamingAddr trex:localhost:5000
IDL:omg.org/CORBA/ORB/InvalidName:1.0
possible cause: can't locate Naming Service
$
```

Zur Administration des Naming Service gibt es das Konsolprogramm nsadmin:

```
$ nsadmin -ORBNamingAddr inet:localhost:5000
CosNaming administration tool for MICO(tm)
nsadmin> help
CosNaming administration tool for MICO(tm)
available commands:
        help            - print this message
        pwd             - print the current naming context
        ls [context]    - list the naming context
        cd context      - change the current naming context
        mkdir context   - create a new naming context
        bind name oref  - create a new binding for the object
        bind_context name oref - connect a naming context
        cat entry       - show entry
        iordump entry   - iordump entry
        url entry       - show corbaname url for entry
        locurl entry    - show corbaloc url for entry
        rm entry        - delete the naming entry
        exit            - quit nsadmin
```

```
nsadmin> ls
        AddressBook
nsadmin> cat AddressBook
IOR:
010000000f00000049444c3a4950426f6f6b3a312e300000020000000000000002c00000001010000
0a0000006c696e75782e69627300230814000002f32353330392f31303132343834323435 2f5f30
01000000240000000100000001000000010000001400000001000000010001000000000009010100
00000000
nsadmin> iordump AddressBook
    Repo Id:  IDL:IPBook:1.0
IIOP Profile
    Version:  1.0
    Address:  inet:linux.ibs:2083
   Location:  corbaloc::linux.ibs:2083//25309/1012484245/%5f0
        Key:  2f 32 35 33 30 39 2f 31 30 31 32 34 38 34 32 34 /25309/101248424
              35 2f 5f 30                                      5/_0

Multiple Components Profile
 Components:  Native Codesets:
              normal: ISO 8859-1:1987; Latin Alphabet No. 1
                wide: ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format
16-bit form
        Key:  (empty)

nsadmin> exit
$
```

Der Vorteil der Verwendung des Naming Service kommt erst dann voll zur Wirkung, wenn eine Vielzahl von CORBA Objekten eines »Frameworks« verwaltet werden sollen. Statt dann mit einer Unmenge von stringified IORs oder Netzwerkadressen zu arbeiten, reicht es nun aus, allen Anwendungskomponenten lediglich die Adresse des Namingservice mitzuteilen. Bei geschickter Strukturierung unter Verwendung von Namenskontexten kann eine Vielzahl von Objekten verwaltet werden.