Programmiermodelle verteilter Systeme

Nachrichtenbasiertes Modell
Sockets

siehe Kapitel Linux

Auftragsorientiertes Modell
Remote Procedure Call – RPC

bisher:

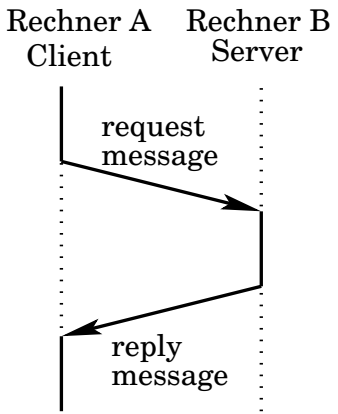
- System war bereits in Clients, Server oder verteilte Prozesse untergliedert → verteilte Struktur liegt bereits vor
- Programmiermodell: senden und empfangen von Nachrichten zwischen den Prozessen

jetzt:

- bestehende, nicht-verteilte Anwendung soll auf mehrere Rechner verteilt werden
- monolithische Anwendung: Sammlung von Prozeduren
- aufteilen in Prozeduraufrufer (Clients) und die Prozedur selbst (Server)

RPC (2)

Aufruf einer entfernten Prozedur:



- der aufrufende Prozess wird blockiert
- Abarbeitung der Prozedur findet auf anderer Maschine statt
- Parameter und Ergebnisse werden zwischen den Maschinen transportiert
- Remote Procedure Call zeigt (fast) das vertraute Verhalten von lokalen Prozeduraufrufen

RPC (3)

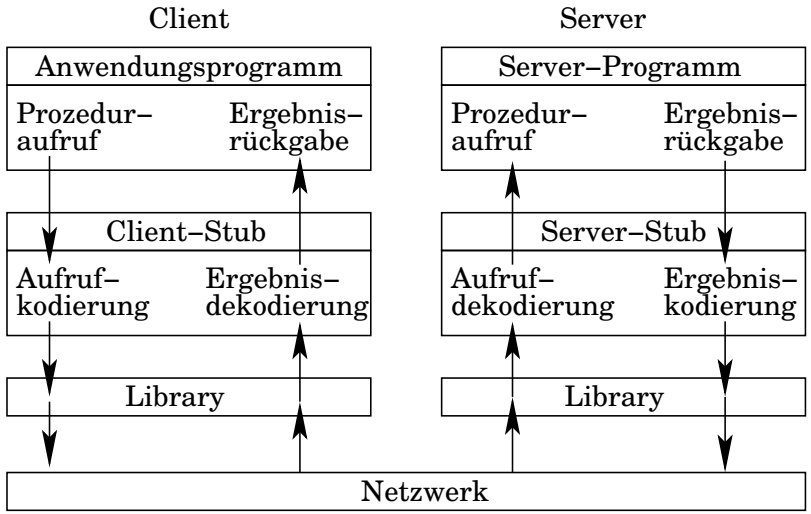
RPC soll wie lokaler Prozeduraufruf aussehen → verwende **Stub-Procedure** (Stub: Stummel, Stumpf)

- binden (bestimmen der Server-Adresse),
- zusammenstellen und verschicken der Nachricht,
- überwachen der korrekten Übertragung, ...

Server-Stub:

- führt Endlosschleife aus und wartet auf Nachrichten
- Nachricht entpacken und gewünschte Prozedur aufrufen
- Ergebnis verpacken und an Client schicken

RPC (4)



Parameterübertragung

- **call by value:** für die entfernte Prozedur ist ein Wert-Parameter eine initialisierte lokale Variable, die beliebig modifizierbar ist → keine Probleme
- **call by reference:** die Adresse einer Variablen kann nicht an die entfernte Prozedur übergeben werden, da Aufrufer und Prozedur in verschiedenen Adressräumen laufen

  Lösung: verbieten von Pointern und Referenzübergaben oder Nachbilden der call by reference-Semantik durch **call by copy/restore**

call by copy/restore

- kopiere die Variablen auf den Stack des Aufgerufenen (wie call by value)
- kopiere die Parameter am Ende der Prozedur zurück in die Variablen (Überschreiben der Werte des Aufrufes)
- Problem: gleicher Parameter mehrfach in Parameterliste

```
proc(int *x, int *y) {
    *x = *x + 1;
    *y = *y + 1;
}
```

```
main() {
    int a = 0;
    proc(&a, &a);
    printf("%d\n", a);
}
```

Anmerkungen

- Zeiger auf komplexe Datenstrukturen (lineare Liste): komplette DS auf den Server kopieren (Adressen durch Werte ersetzen) oder der Server erfragt beim Client den Inhalt jeder Adresse (sehr ineffizient)
- Zugriff auf globale Variablen von entfernten Prozeduren nicht möglich, da Aufrufer und Prozedur in verschiedenen Adressräumen laufen
- Unterschiedliche Datenrepräsentation:
  * Strings: EBCDIC- oder ASCII-Code
  * Integer: Einer- oder Zweierkomplement
  * Float: Größe von Mantisse und Exponent

  → **automatische Konvertierung notwendig**

automatische Konvertierung

direkte Konvertierung:

- der Client-Stub hängt vor die Nachricht eine Indikation des verwendeten Formats
- der Server-Stub wandelt ggf. vom fremden Datenformat ins eigene Format um

Problem: bei n verschiedenen Datenformaten im Netz sind $n \cdot (n-1)$ Konvertierungsfunktionen notwendig

Lösung: verwende maschinenunabhängiges Netzwerkdatenformat: Reduktion auf $2n$ Konvertierungsfunktionen

automatische Konvertierung (2)

maschinenunabhängiges Netzwerkdatenformat:

- der Client-Stub wandelt das eigene Datenformat in die Netzwerkdatendarstellung
- der Server-Stub wandelt die Netzwerkdatendarstellung ins eigene Format

Nachteile:

- zwei unnötige Konvertierungen, falls Client und Server dieselbe Datendarstellung verwenden
- direkte Konvertierung erfordert nur eine Umwandlung im Gegensatz zu zwei Umwandlungen bei Verwendung von maschinenunabhängigen Formaten

automatische Konvertierung (3)

Beispiele:

- Xerox Courier RPC-Protokoll: big endian, Feldgröße 16 Bit, 8-Bit ASCII kann auf andere Zeichensätze umgeschaltet werden
- das Sun RPC-Protokoll verwendet **XDR** (eXternal Data Representation): big endian, Feldgröße 32 Bit
- **NDR** (Network Data Representation) unterstützt mehrere Formate: Client kann eigenes Format benutzen, falls es unterstützt wird; Server muss ggf. umwandeln (wird von DCE der OSF verwendet).

Typing

implizites Typing: nur der Wert einer Variablen wird übertragen, nicht der Datentyp (XDR und NDR)

explizites Typing: Beschreibung von Datenstrukturen nach ASN.1 (Abstract Syntax Notation 1, ISO) durch Angabe des Datentyps (verschlüsselt in 1 Byte) und des Wertes

Beispiel: Übertragen eines 32-Bit Integer Wertes

- implizites Typing: nur der Wert wird übertragen
- ASN.1: ein Byte gibt an, das der Wert ein Integer ist, ein weiteres Byte gibt die Länge des Integer-Feldes in Byte an, dann wird der Wert übertragen

Identifikation und Binden der Parameter

Lokalisierung des Servers (binden):

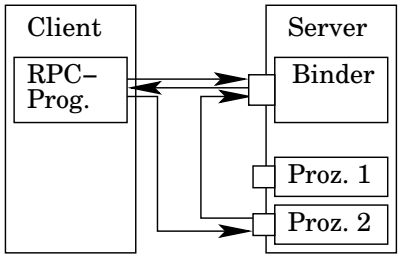
- **statisch:** direkte Angabe der Server-Adresse
- **dynamisch:** mittels Broadcast oder über einen Broker

Für die Server muss ein Mechanismus zum Exportieren der angebotenen Prozedurschnittstellen existieren:

- Server schickt Name, Versionsnummer, Identifikation und Adresse zu dem Broker (bei RPCs: Binder)
- der Binder trägt Service und Adresse in Tabelle ein (Registrierung) oder löscht die Einträge (Deregistrierung)
- der Binder hat eine feste Adresse (`sunrpc 111/tcp` in `/etc/services`)

Identifikation und Binden der Parameter (2)

In Unix-Systemen heißt der Binder `portmap` oder `rpcbind`.



1. Server registriert die Prozedur beim Binder
2. Client erfragt Serveradresse beim Binder
3. Binder liefert Serveradresse
4. Client ruft RPC-Prozedur auf

In `/etc/rpc` sind Standard RPC-Dienste aufgelistet:

```
portmapper      100000  portmap sunrpc rpcbind
nfs             100003  nfsprog
ypserv          100004  ypprog
mountd          100005  mount showmount
```


Identifikation und Binden der Parameter (3)

Auszug aus `portmap(8)`:

Portmap is a server that converts RPC program numbers into DARPA protocol port numbers. It must be running in order to make RPC calls.

When an RPC server is started, it will tell portmap what port number it is listening to, and what RPC program numbers it is prepared to serve. When a client wishes to make an RPC call to a given program number, it will first contact portmap on the server machine to determine the port number where RPC packets should be sent.

Portmap must be started before any RPC servers are invoked. ...

Identifikation und Binden der Parameter (4)

`rpcinfo` liefert die Zuordnung zwischen RPC und Port:

```
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
 536870920    1   udp  32771
 536870920    1   tcp  45182
```

Auszug aus `rpcinfo(8)`:

rpcinfo makes an RPC call to an RPC server and reports what it finds.

OPTIONS -p Probe the portmapper on host, and print a list of all registered RPC programs. If host is not specified, it defaults to the value returned by `hostname(1)`.

Sun RPC

- Open Network Computing (ONC)
- Grundlage für NFS und NIS (früher YP)
- Routinen für RPC und XDR
- bietet blockierende, asynchrone und Broadcast-RPCs
- asynchron: Server zyklisch abfragen oder Callback-RPC
- broadcast: Client schickt ein Broadcast-Paket und wartet dann auf mehrere Antworten
- **Remote Procedure Call Programming Guide** unter
  `http://docs.freebsd.org/44doc/psd/23.rpc/paper.pdf`
  oder direkt bei SUN Microsystems (Network Interfaces Programmer's Guide)

RPC: Programmierung

Includes unter `/usr/include/rpc` und `/usr/include/rpcsvc`

Entwickler muss nicht alle RPC-Funktionen kennen: Großteil der Arbeit übernimmt der RPC-Compiler `rpcgen`:

- Auf der Server-Seite müssen nur die Funktionen implementiert werden, die aufgerufen werden sollen.
- Funktion, die von der Client-Seite die Kommunikation mit dem Server in Gang setzt:

  `CLIENT *clnt_create(char *server, PROG, VERS, "tcp")`
  `PROG` ist eine Programm-, `VERS` eine Versionsnummer, die einheitlich von `rpcgen` festgelegt werden. Neben `tcp` wird zur Zeit noch `udp` unterstützt.
- Liste aller low-level Funktionen: `man -S3 rpc`

Routinen der RPC-Bibliothek

- `clnt_pcreateerror(char *s)`
  gibt eine Fehlermeldung auf `stderr` aus, falls `clnt_create` fehlgeschlagen ist. Meldung wird an `s` angehängt.
- `clnt_perror(CLIENT *clnt, char *s)`
  gibt eine Fehlermeldung auf `stderr` aus, falls eine entfernte Prozedur einen Fehler zurückgeliefert hat. `clnt` ist das Handle, mit dem die Prozedur aufgerufen wurde.
- `clnt_freeres(CLIENT *clnt, xdrproc_t proc, char *out)`
  gibt den Speicher frei, der vom RPC/XDR-System für das Dekodieren des Prozedur-Ergebnisses belegt wurde.
- `clnt_destroy(CLIENT *clnt)`
  gibt den Speicher wieder frei, der bei `clnt_create` allokiert wurde.

RPC-Bibliothek

Die Interface-Routinen benutzen XDR-Routinen, um die Daten von maschinenabhängiger Form in XDR-Standard zu konvertieren und umgekehrt.

bisher: low-level RPC-Routinen zum Gestalten der Netzwerkfunktionalitäten, XDR-Routinen zum Konvertieren der Daten von und zum gemeinsamen Format direkt aufrufen.

jetzt: `rpcgen` generiert aus einer Spezifikationsdatei (in RPC Language) mehrere C-Files und Header-Files.

- C-Files: Quellcode, enthält Aufrufe der low-level RPC- und XDR-Routinen
- Header-Files: enthalten Strukturdefinitionen, die von den entfernten Prozeduren benötigt werden

Spezifikationsdatei

- wird abgelegt in einer Datei mit Endung `.x`
- Programmdefinition

```
program identifier {
    version_list
} = value
```

  `identifier`: der Name des Programms, `value`: positive, ganze Zahl von 2000 000 bis 3fff fff, kleinere Nummern sind reserviert

- Versionsliste

```
version identifier {
    procedure_list
} = value
```


Spezifikationsdatei (2)

- jede Version enthält eine Liste von Prozeduren der Form

```
data_type procedure_name(data_type) = value
```

  `data_type` kann irgendein einfacher oder ein komplexer C-Datentyp sein. Komplexe Datentypen werden in der Spezifikationsdatei definiert.

- üblich: Programm-, Versions- und Prozedurnamen werden in Großbuchstaben geschrieben

Spezifikationsdatei: Konstanten

der `rpcgen`-Compiler übersetzt Konstanten

```
const MAX_ENTRIES = 1024;
```

in ein `#define`-Konstrukt:

```
#define MAX_ENTRIES 1024
```

Spezifikationsdatei: Strukturen

der `rpcgen`-Compiler transferiert Strukturen in eine Header-Datei und fügt eine korrespondierende `typedef`-Definition hinzu. Aus

```
struct pair {
    int a, b;
};
```

wird:

```
struct pair {
    int a, b;
};
typedef struct pair pair;
```


Spezifikationsdatei: Aufzählungen

Aufzählungen werden in eine Header-Datei transferiert und es wird eine korrespondierende `typedef`-Definition hinzugefügt. Aus

```
enum trafficlight {
    RED = 0, AMBER = 1, GREEN = 2
};
```

wird:

```
enum trafficlight {
    RED = 0, AMBER = 1, GREEN = 2
};
typedef enum trafficlight trafficlight;
```


Spezifikationsdatei: Unions

Unions haben in RPCL eine andere Syntax als in C. Aus

```
union result switch (int status) {
    case 0: char time[32];
    case 1: int errno;
};
```

wird:

```
struct result {
    int status;
    union {
        char time[32];
        int errno;
    } result_u;
};
typedef struct result result;
```


Spezifikationsdatei: Felder

Ein Feld wird transformiert in eine Struktur bestehend aus Längenspezifizierer und einem Zeiger auf das Feld. Aus

```
int height[12];
int widths<>;
```

wird:

```
struct {
    u_int heights_len;
    int *heights_val;
} heights;

struct {
    u_int widths_len;
    int *widths_val;
} widths;
```

Spezifikationsdatei: Zeiger

Zeiger als solches machen bei RPC keinen Sinn aufgrund der unterschiedlichen Adressräume von Client und Server. Aber Zeiger werden in dynamischen Datenstrukturen verwendet. Aus

```
struct list {
    int data;
    list *nextp;
};
```

wird:

```
struct list {
    int data;
    struct list *nextp;
};
typedef struct list list;
```


Beispiel: Spezifikationsdatei (a)

Datei `ggt.x`

```
struct args {
    long p;
    long q;
};

program GGT_PROG {
    version GGT_VERS {
        long GGT(args) = 1;
    } = 1;
} = 0x20000008;
```


Beispiel: Spezifikationsdatei (b)

`rpcgen ggT.x` generiert `ggT.h`: (Ausschnitt)

```
#include <rpc/rpc.h>
struct args {
    long p;
    long q;
};
typedef struct args args;

#define GGT_PROG 0x20000008
#define GGT_VERS 1

extern long *ggt_1(args *, CLIENT *);
extern long *ggt_1_svc(args *, struct svc_req *);
```


Beispiel: Spezifikationsdatei (c)

`rpcgen ggT.x` generiert `ggT_clnt.c`: (Ausschnitt)

```
static struct timeval TIMEOUT = {25, 0};

long *ggt_1(args *argp, CLIENT *clnt) {
    static long clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call(clnt, GGT, (xdrproc_t)xdr_args,
            (caddr_t)argp, (xdrproc_t)xdr_long,
            (caddr_t)&clnt_res, TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

Beispiel: Spezifikationsdatei (d)

`rpcgen ggT.x` generiert `ggT_xdr.c`: konvertieren der Daten in heterogener Umgebung

```
#include "ggT.h"

bool_t xdr_args(XDR *xdrs, args *objp) {
    register int32_t *buf;

    if (!xdr_long(xdrs, &objp->p))
        return FALSE;
    if (!xdr_long(xdrs, &objp->q))
        return FALSE;
    return TRUE;
}
```


Beispiel: Spezifikationsdatei (e)

Außerdem erzeugt `rpcgen ggT.x` den kompletten Server, der auf den Prozeduraufruf reagiert: `ggT_svc.c`

Nun ist noch die Funktion auf Server-Seite zu schreiben:

- Der Name besteht aus dem in der Spezifikationsdatei angegebenen Namen, jedoch in Kleinbuchstaben. An den Funktionsnamen wird bei Solaris ein `_1`, bei Linux ein `_1_svc` für die Versionssnummer 1 angehängt.
- Rückgabewerte und Argumente sind Zeiger auf die in der Spezifikationsdatei angegebenen Typen.

Beispiel: Server (a)

Datei `server.c`

```
#include <math.h>
/* ggT.h is generated by rpcgen */
#include "ggT.h"

/* den Funktionen auf der Serverseite wird zusätzlich
 * ein Requesthandler "*req" mitgegeben
 */
long *ggt_1_svc(args *arg, struct svc_req *req) {
    long r, p;
    static long q;
```


Beispiel: Server (b)

```
    p = arg->p;
    q = arg->q;
    do {
        r = p % q;
        if (r != 0) {
            p = q;
            q = r;
        }
    } while (r != 0);
    return &q;
}
```

Beispiel: Server (c)

übersetzen und linken des Servers:

```
gcc -Wall -pedantic -c server.c
gcc -Wall -pedantic -c ggT_svc.c
gcc -Wall -pedantic -c ggT_xdr.c
gcc -o ggTServer server.o ggT_svc.o ggT_xdr.o -lnsl
```

Schließlich ist noch der Client zu schreiben ...

Beispiel: Client (a)

Datei `client.c`

```
#include <stdio.h>
#include "ggT.h"

int main(int argc, char **argv) {
    CLIENT *cl;
    long *r;
    args arg;

    if (argc != 4) {
        fprintf(stderr, "usage: %s server num1 num2\n",
                argv[0]);
        exit(1);
    }
```


Beispiel: Client (b)

```
    cl = clnt_create(argv[1], GGT_PROG, GGT_VERS, "tcp");
    if (cl == NULL) {
        clnt_pcreateerror(argv[1]); exit(2);
    }
    arg.p = atol(argv[2]);
    arg.q = atol(argv[3]);
    r = ggt_1(&arg, cl);
    if (r == NULL) {
        clnt_perror(cl, "ggt_1"); exit(3);
    }
    printf("ggT(%ld, %ld) = %ld\n", arg.p, arg.q, *r);

    exit(0);
}
```


Beispiel: Client (c)

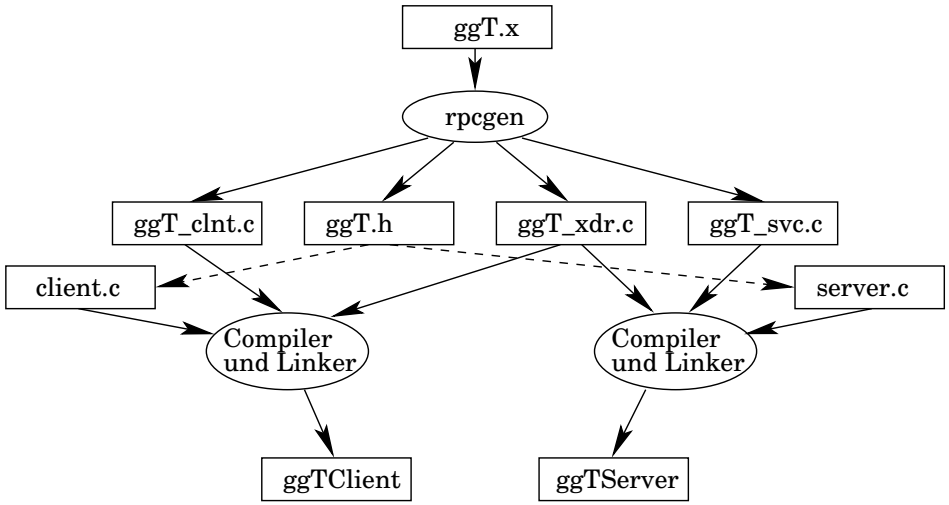
übersetzen und linken des Clients:

```
gcc -Wall -pedantic -c client.c
gcc -Wall -pedantic -c ggT_clnt.c
gcc -Wall -pedantic -c ggT_xdr.c
gcc -o ggTClient client.o ggT_clnt.o ggT_xdr.o -lnsl
```

Testen des Programms:

- ggf. starten des Portmappers
- starten des Servers
- starten des Clients

RPC: Übersicht



Objektbasiertes Modell

Objektbasierte Ansätze:

- ermöglichen Kommunikation und Koordination von Objekten auf verschiedenen Rechnern
- jedes entfernte Objekt spezifiziert ein Interface: welche Methoden können von Clients aufgerufen werden?

zur Zeit wichtige Systeme:

- auf Java basierende Remote Methode Invocation (RMI)
- von Microsoft getragenes Distributed Component Object Model (DCOM)
- von der Object Management Group (OMG) definierte Common Object Request Broker Architecture

RMI & DCOM

Java Remote Method Invocation:

- plattformunabhängig durch Java Virtual Maschine
- es lassen sich nur Java-Objekte ansprechen

Microsofts Distributed Component Object Model:

- an Microsoft-Betriebssysteme gebunden
- entwickelt aus dem Component Object Model (COM) und dem Object Linking and Embedding (OLE)
- Spezifikation der Schnittstellen durch Object Definition Language (ODL) oder von DCE-RPCs abgeleitete Interface Definition Language (IDL)
- DCOM-Objekte besitzen keinen Zustand

Object Management Group (OMG) & CORBA

- internationales Konsortium: 1989 von 8 Mitgliedern gegründet, hat mittlerweile über 700 Vertreter aus allen wesentlichen Computerfirmen
- implementiert keine Produkte, sondern legt Spezifikationen für Schnittstellen und Protokolle fest
- Mitglieder reichen Spezifikationen ein, die veröffentlicht und diskutiert werden, und schließlich einer Abstimmung unterliegen
- hat ein abstraktes Objektmodell und eine objektorientierte Referenzarchitektur definiert: Object Management Architecture (OMA), besser bekannt unter CORBA
- freie CORBA-Implementierung unter Linux: MiCO Informationen unter `http://www.mico.org`

Implementationsschritte

- Interface definieren (IDL-Datei)
- Serverklasse erstellen (abgeleitet von Skeleton-Klasse)
- Server-Programm schreiben, das Objekt erzeugt und in Endlosschleife geht
- Client-Programm schreiben, das entferntes Objekt (Stub-Klasse) "erzeugt"
- Einfachste Objektidentifikation über Interoperable Object Reference (IOR) in einer Datei
- Ohne gemeinsame Datei: Client findet Objekt über CORBA Naming Service oder MICO-Binder

Interface-Definition

erster Schritt: Definition aller benötigten Interfaces (welche Methoden können vom Client auf den Server-Objekten aufgerufen werden?)

Interface-Definitionen erfolgen in der Interface Definition Language (IDL) (Syntax ähnelt C++ Klassendefinition)

Beispiel: einfaches Adressbuch, ein Eintrag besteht aus Name und Telefonnummer.

Das Interface `AddrBook` enthält zwei Methoden:

- Hinzufügen eines neuen Eintrags
- Suchen eines Eintrags anhand des Namens

Interface-Definition (2)

Interface-Definition in Datei `AddrBook.idl`:

```
exception AddrNotFoundException {
    string msg;
};
interface AddrBook {
    void addAddr(in string name, in string number);
    string getAddrByName(in string name)
            raises (AddrNotFoundException);
};
```

Jeder Methodenaufruf ist eine Netzwerkübertragung!

- wenn nur wenige Bytes an Nutzdaten übertragen werden, entsteht ein beträchtlicher Overhead
- abweichen von Theorie des objektorientierten Entwurfs

Interface-Definition (3)

Interface Definition Language:

- definiert die Schnittstelle zwischen dem Client-Code und der server-seitigen Objektimplementierung
- ist sprachunabhängig und wurde von der OMG bei der ISO als Standard eingereicht
- spezifiziert die
  * benutzerdefinierten Datentypen,
  * die öffentlichen Attribute einer Komponente,
  * die Basisklasse, von der sie erbt,
  * die Ausnahmen, die ausgelöst werden und
  * die Methoden inkl. Parameter und Rückgabewert.

Erzeugung Stub-/Skeleton-Klasse

Verantwortlich für die Vermittlung der verteilten Prozeduraufrufe ist der **Objekt-Adapter**

- alt: Basic Object Adapter (BOA)
- seit CORBA 2.2: Portable Object Adapter (POA)

Abhänging von der Wahl des Objektadapters werden verschiedene Stub- und Skeletonklassen erzeugt.

Für POA: Das Übersetzen der Datei `AddrBook.idl` mittels

```
idl --poa --c++-suffix cpp AddrBook.idl
```

liefert die Dateien `AddrBook.h` und `AddrBook.cpp`: Dateien enthalten einen Großteil der Implementierung des Server-Objektes sowie die Stub-Klassen und weitere Funktionen.

Implementieren des Server-Objekts

zweiter Schritt nach Erstellen der Interface-Definitionen:

- Server-Objekt muss Applikationslogik implementieren, also Methoden `addAddr` und `getAddrByName` bereitstellen
- Serverobjekt muss über den Objekt Adapter die Verbindung zum Object Request Broker herstellen

Der IDL-Prozessor nimmt dem Programmierer eine Menge Arbeit beim Erstellen des Server-Objekts ab!

- die Klasse `POA_AddrBook` dient als **Basisklasse** für das Server-Objekt (Skeleton-Klasse)
- neue Klasse schreiben, die von `POA_AddrBook` abgeleitet ist und mindestens die beiden pur virtuellen Funktionen `addAddr()` und `getAddrByName()` implementiert
- **üblich:** benenne die Klasse mit Nachsatz `_impl`

Implementieren des Server-Objekts (2)

Deklaration der Server-Klasse in Datei `AddrBook_impl.h`:

```
#include "AddrBook.h"
#include <map>
#include <string>

class AddrBook_impl : virtual public POA_AddrBook {
  public:
    virtual void addAddr(const char* name,
            const char* number);
    virtual char* getAddrByName(const char* name);
  private:
    map <string, string, less<string> > _items;
};
```


Implementieren des Server-Objekts (3)

Definition der Server-Klasse in Datei `AddrBook_impl.cpp`:

```
#include <CORBA.h>
#include "AddrBook_impl.h"

void AddrBook_impl::addAddr(const char* name,
        const char* number) {
    _items[name] = number;
}
...
```

Übergabeparameter belegen Speicherplatz auf dem Heap: wird vom ORB allokiert und nach Beendigung der Funktion wieder freigegeben. Danach dürfen im Server-Objekt keinerlei Referenzen mehr auf diesen Speicher existieren!

Implementieren des Server-Objekts (4)

Fortsetzung:

```
char* AddrBook_impl::getAddrByName(const char* name) {
    map <string, string, less<string> >::iterator r;
    r = _items.find(name);
    if (r != _items.end()) {
        return CORBA::string_dup((*r).second.c_str());
    } else throw AddrNotFoundException(name);
}
```

`getAddrByName()` liefert einen Zeiger auf `char` an den ORB zurück. **CORBA-Standard:** der ORB ist für die Freigabe dieses Speicherbereiches verantwortlich → auf dem Heap liegenden Speicher zurückliefern: `CORBA::string_dup()`

Implementieren des Server-Objekts (5)

kompilieren der bisher erstellten Module:

```
mico-c++ -c AddrBook.cpp
mico-c++ -c AddrBook_impl.cpp
```


Implementieren der Server-Applikation

dritter Schritt: Erstellen eines kompletten CORBA-Server-Programms durch implementieren der `main()`-Funktion.

Header-Files inkludieren: absolut notwendig sind `CORBA.h` sowie die Deklaration des Servers in `AddrBook_impl.h`:

```
#include <CORBA.h>
#include "AddrBook_impl.h"
#include <fstream.h>

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


Implementieren der Server-Applikation (2)

ORB initialisieren, bevor Kommandozeilenargumente ausgewertet werden → zusätzliche Parameter für den ORB können mit auf der Kommandozeile übergeben werden

die Funktion `CORBA::ORB_init(argc, argv)` wertet die ORB-spezifischen Parameter aus und passt `argc` und `argv` entsprechend an:

```
try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

alle CORBA-Systemaufrufe in try-catch Block kapseln

Implementieren der Server-Applikation (3)

POA-Manager erzeugen und initialisieren: verwende vom System zur Verfügung gestellten Root-POA-Manager

```
CORBA::Object_var poaobj =
      orb -> resolve_initial_references("RootPOA");
PortableServer::POA_var poa =
      PortableServer::POA::_narrow(poaobj);
PortableServer::POAManager_var mgr =
      poa -> the_POAManager();
```


Implementieren der Server-Applikation (4)

Server-Objekt erzeugen und aktivieren: CORBA stellt mehrere Aktivierungsmethoden zur Verfügung (hier implizite Aktivierung):

```
AddrBook_impl *impl = new AddrBook_impl;
AddrBook_var f = impl -> _this();
// AddrBook_var und _this sind in AddrBook.h definiert
```


Implementieren der Server-Applikation (5)

Interoperable Object Reference (IOR): Eindeutige Identikation des CORBA-Server-Objekts → muss vom Server zum Client transportiert werden

einfachste Möglichkeit: IOR in Zeichenkette umwandeln und in eine Datei speichern, auf die Server und Client beide Zugriff haben:

```
CORBA::String_var s = orb -> object_to_string(f);
ofstream out("AddrBook.ref");
out << s << endl;   // IOR schreiben
out.close();
```


Implementieren der Server-Applikation (6)

POA-Manager aktivieren und ORB starten:

```
mgr -> activate();
orb -> run();
```

Programm kehrt nur bei explizitem Shutdown des ORBs zurück → POA-Manager und Server-Objekt aufräumen:

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

Implementieren der Server-Applikation (7)

Fehlerbehandlung:

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

übersetzen und linken des Servers:

```
mico-c++ -c server.cpp
mico-ld -o server server.o AddrBook.o \
        AddrBook_impl.o -lmico2.3.11
```


Implementieren des Client

Inkludieren der Header-Files: Notwendig sind `CORBA.h` und die vom IDL-Prozessor erzeugte Datei `AddrBook.h`, die den Client-Stub enthält.

```
#include <CORBA.h>
#include "AddrBook.h"
#include <fstream>

#define APPEND 1
#define FIND   2

int main(int argc, char **argv) {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    int rc = 0;
```


Implementieren des Client (2)

Überprüfen der Programm-Parameter:

```
int mode;
if ((argc < 3) || (argc > 4)) {
   cout << "usage: " << argv[0] << " -a name ...
   exit(1);
}
if (strcmp(argv[1], "-a") == 0)
   mode = APPEND;
else if (strcmp(argv[1], "-f") == 0)
   mode = FIND;
else {
   cout << "usage: " << argv[0] << " -a name ...
   exit(1);
}
```


Implementieren des Client (3)

IOR des Servers lesen: der Client benötigt Information, welchen CORBA-Server er verwenden soll → vom Server erzeugte Datei `AddrBook.ref` lesen

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

mittels IOR ein Objekt vom Typ `AddrBook_var` erstellen:

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

Implementieren des Client (4)

Anfrage starten, Antwort ausgeben, ggf. Fehlerbehandlung:

```
    if (mode == APPEND)
        f -> addAddr(argv[2], argv[3]);
    else {
        string number = f -> getAddrByName(argv[2]);
        cout << "number of " << argv[2] << ": "
             << number << endl;
    }
} catch (AddrNotFoundException& ex) {
    cout << "number of " << ex.msg << " not found\n";
    rc = 0;
}
```


Implementieren des Client (5)

Auffangen von Exceptions und Ausgabe ihrer Ursache:

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

    return rc;
}
```


Implementieren des Client (6)

übersetzen und linken des Client:

```
mico-c++ -c client.cpp
mico-ld -o client client.o AddrBook.o -lmico2.3.11
```

Problem dieser Implementierung: Objektreferenz (IOR) des Servers wird in Datei gespeichert. Client- und Server-Programm benötigen Zugriff auf diese Datei (kein Problem bei einzelnem Rechner oder gemeinsamen Netzlaufwerken wie SAMBA- oder NFS-Shares)

Lösung: verwende den properitären MiCO-Binder oder den CORBA Naming Service

der MiCO-Binder

Idee:

- eindeutige Identifizierung des Servers nicht durch IOR sondern durch seine Netzwerkadresse: IP-Adresse des Rechners und Portnummer
- stellt das CORBA-Server-Programm mehrere Objekte zur Verfügung: unterscheide Objekte anhand des Typ des Servers (Repository-Id der Interface-Definition)
- existieren mehrere Server gleichen Typs: Programmierer muss jedem Server-Programm einen eindeutigen Namen geben

der MiCO-Binder (2)

unser Adressbuch-Server stellt nur ein CORBA-Objekt nach außen bereit → der Server kann ohne Veränderung für die Benutzung des MICO-Binders verwendet werden

Voraussetzung: stelle sicher, dass der Server beim Starten immer die gleiche Portnummer benutzt

Kommandozeilenoption:

```
-ORBIIOPAddr inet:<hostname>:<port>
```


der MiCO-Binder (3)

Änderungen am Client:

- der Aufruf von `string_to_object(stringified_ior)` muss durch `bind(repository_id, address)` ersetzt werden
- Adresse als Kommandozeilenargument übergeben

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


der MiCO-Binder (4)

Mittels des MiCO-Tools `iordump` kann der Inhalt des IOR analysiert werden. `cat AddrBook.ref | iordump` liefert:

```
     Repo Id:  IDL:AddrBook:1.0

IIOP Profile
     Version:  1.0
     Address:  inet:endor.myhome.de:42042
    Location:  corbaloc::endor.myhome.de:42042//...
...
```


der MiCO-Binder (5)

Nachteile:

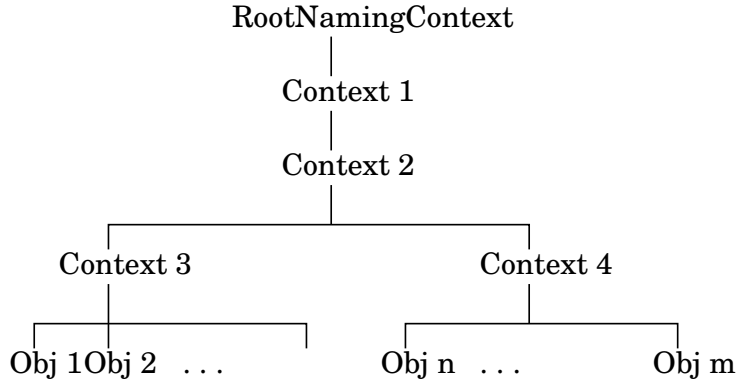
- properitäre Erweiterung des CORBA-Standards: falls es in der verteilten Anwendung Objekte gibt, die eine andere CORBA-Implementierung benutzen, so funktioniert diese Methode nicht
- falls die Applikation auf viele CORBA-Objekte zugreift: Verwaltung der vielen Portnummern wird kompliziert

⇒ CORBA Naming Service

CORBA Naming Service

hierachisches Verzeichnis (ähnlich: LDAP)

- Kontext: eine Art Unterverzeichnis
- Eintrag: enthält IOR eines konkreten CORBA-Objektes



CORBA Naming Service (2)

Änderungen am Server:

- Header-Datei `coss/CosNaming.h` inkludieren
- den Root-Kontext des Naming Service auflösen: mittels `resolve_initial_references` die per Kommandozeilenoption `-ORBNamingAddr` übergebene Objektreferenz auf den Naming Service besorgen ...

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


CORBA Naming Service (3)

- und anschliessend auf einen NamingContext einengen

```
CosNaming::NamingContext_var nc =
      CosNaming::NamingContext::_narrow (nsobj);
```

- vom NamingContext (`RootNamingContext`) aus kann die Hierarchie des Naming Service durchlaufen werden (hier: beschränken auf einfachen Namenseintrag `AddressBook`)

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


CORBA Naming Service (4)

- anlegen des Namenseintrags mittels `bind()`:

```
try {
    nc->bind (name, f);
} catch (CosNaming::NamingContext
          ::AlreadyBound_catch &ex) {
    nc->rebind (name, f);
}
```

- Fehler bei Adressangabe des Naming Service abfangen

```
catch(CORBA::ORB::InvalidName_catch& ex) {
    ex->_print(cerr);
    cerr << "\ncan't locate Naming Service\n";
    rc = 1;
}
```

CORBA Naming Service (5)

Änderungen am Client:

- Header-Datei `coss/CosNaming.h` inkludieren
- Verwenden des Naming Service: Root-Kontext auflösen und auf einen NamingContext einengen

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

CosNaming::NamingContext_var nc =
        CosNaming::NamingContext::_narrow (nsobj);
```


CORBA Naming Service (6)

- `resolve()` aufrufen, um den vom Server angelegten Namenseintrag zu lesen

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

CORBA::Object_var obj = nc->resolve(name);
```

- Fehlerbehandlung erweitern:
  * `CORBA::ORB::InvalidName_catch`
  * `CosNaming::NamingContext::NotFound_catch`

CORBA Naming Service (7)

übersetzen und linken:

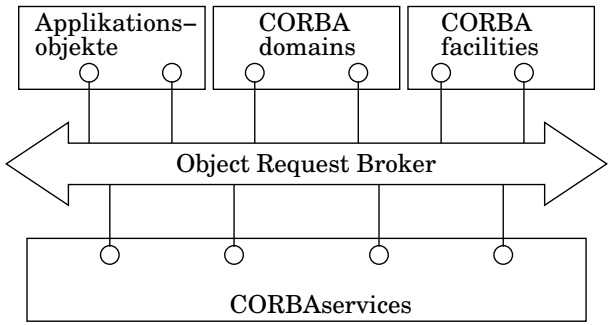
```
mico-c++ -c server.cpp
mico-ld -o server server.o AddrBook.o AddrBook_impl.o \
        -lmico2.3.11 -lmicocoss2.3.11
mico-c++ -c client.cpp
mico-ld -o client client.o AddrBook.o \
        -lmico2.3.11 -lmicocoss2.3.11
```

CORBA Naming Service:

- starten: `nsd -ORBIIOPAddr inet:<hostname>:<port>`
- Administration: `nsadmin -ORBNamingAddr inet:...`

Server starten: `server -ORBNamingAddr inet:<host>:<port>`
Client starten: `client -ORBNamingAddr inet:<host>:<port>`

Object Management Architecture



- **Object Request Broker:** schafft die Kommunikationsinfrastruktur zum Weiterleiten von Anfragen an andere Architekturkomponenten (Kreise: Objekte, die Anforderungen versenden und empfangen können)

Object Management Architecture (2)

- **CORBAservices:** Fundament der Architektur. Services auf Systemebene bieten Basisoperationen auf Objekten:
  * Lifecycle Services: create, delete, copy und move
  * Concurrency Control: Lock Manager führt im Auftrag von Transaktionen oder Threads Lock/Unlock-Funktionen für nebenläufige Verarbeitung aus
  * Transaction Service: stellt ein Zwei-Phasen-Commit-Protokoll bereit
  * Event Service: Dynamische Registrierung für Ereignisse → permanentes Objekt `EventChannel` sammelt Ereignisse und verteilt diese an Komponenten, die nichts voneinander wissen
  * Naming Service, Time Service, Security Service, Persistence Service, Trader Service, ...

Object Management Architecture (3)

- **CORBAfacility:** direkt von Applikationsobjekten nutzbare Services. CORBAfacilities können CORBAservices benutzen, von ihnen erben oder sie erweitern.

  bereitstellen allgemeiner Funktionalität analog zu großen Klassenbibliotheken

  Umfassen Distributed Document Component, System Management, Data Interchange, Time Operation, ...
- **CORBAdomain:** Beschreibung von anwendungsunabhängigen Konzepten (bereichsspezifische Rahmenwerke wie Business Object Framework, Manufacturing, Transportation, Gesundheitswesen, Kunden, Produkte, Zahlungsvorgänge, Bestellungen, Geld, ...)

Object Management Architecture (4)

- **CORBAdomain:** (Fortsetzung) Abstraktion von Werkzeugen, Anwendungen, Datenbanken → Modellierung und Präsentation von wieder erkennbaren Dingen des täglichen Lebens

Object Request Broker: Ziele des forwarding Brokers

- Ortstranzparenz: der Client muss nicht wissen, welcher Server den Objektdienst bereitstellt
- Kommunikationsservice zwischen Client und Server, allgemeine Services wie Namens- oder Sicherheitsdienst
- Transparenz bzgl. Betriebssystem, Netzwerkprotokoll, Implementierungssprache, ...

Interfaces auf Client-Seite

- **IDL Stub:** enthält die vom IDL-Prozessor erzeugten Funktionen für das Client-Programm
- **Dynamic Invocation Interface (DII):** Wird verwendet, wenn zur Compile-Zeit das Interface des Objektes noch nicht bekannt war und nicht zum Client-Code hinzugebunden werden kann. Informationen über das Objekt werden im **Implementation Repository** abgelegt, so dass der ORB die Objektimplentierung lokalisieren und aktivieren kann.
- **ORB-Interface:** direkter Zugriff auf Funktionen des ORB, z.B. die Funktion zur Umwandlung einer Objektreferenz in einen String

Interfaces auf Server-Seite

- **IDL Skeleton:** Gegenstück zum IDL Stub, wird aus der Interface-Definition generiert.
- **Dynamic Skeleton Interface (DSI):** Gegenstück zum Dynamic Invocation Interface, bindet zur Laufzeit Anfragen vom ORB an eine Objektimplementierung. DSI inspiziert Parameter einer vom ORB eingehenden Anfrage, bestimmt Zielobjekt und Methode mit Hilfe des Implementation Repositories, nimmt Antwort entgegen
- **Object Adapter:** übernimmt die Kommunikationsanbindung an die Objektimplementierung
  * Laufzeitumgebung: Instanziieren von Server-Objekten
  * Weiterleiten der Anforderungen an die Server-Objekte
  * Abbilden der Objektreferenzen auf die Server-Objekte

Objekt-Adapter

Server kann mehrere Objektadapter unterstützen, aber ein Standardadapter **BOA** (Basic Object Adapter, alt) oder **POA** (Portable Object Adapter) ist immer vorhanden:

- Implementierungsverzeichnis zur Installation und Registrierung von Objektimplementierungen
- Methoden zur Generierung und Interpretation von Objektreferenzen
- aktivieren/deaktivieren von Objektimplementierungen
- Methodenaufrufe durch Skeletons

Vier Aktivierungsverfahren: Shared Server, Unshared Server, Server-per-Method, Persistent Server

Interfaces: Begriffe

Marschalling: Vorgehensweise für die Zusammensetzung von mehreren Datenelementen in ein Format, das für die Übertragung geeignet ist

Übersetzen elementarer Datentypen und zusammengesetzter Datenstrukturen in eine externe Darstellung

Un-Marschalling: Zerlegen der zusammengesetzten Daten in äquivalente Datenelemente nach Übertragungsende

ORB-Interoperabilität

CORBA ist ein offener Standard → Unterstützung unterschiedlicher Produkte erforderlich:

- Verantwortung für die Interoperabilität liegt bei ORB
  * Festlegung der Kommunikation/Koordination durch das Protokoll **GIOP** (General Inter-ORB Protocol)
  * Definition der Transfersyntax **CDR** (Common Data Representation) und 7 Nachrichtentypen
- Beispiel: **IIOP** (Internet Inter-ORB Protocol)
  * Beschreibung der Abbildung GIOP auf TCP/IP
  * Interoperable Objektreferenz enthält die ORB-interne Objektreferenz, Internetadresse, Portnummer → Verwaltung durch ORB, unsichtbar für SW-Entwickler
- analog: DCE Environment Specific Inter-ORB Protocol

dynamisches CORBA

Bindung zur Laufzeit ohne Stubs: DII ermöglicht Client, irgendein Objekt zur Laufzeit auszuwählen und dynamisch seine Methoden aufzurufen (ähnlich: Java Reflection API)

Wie lokalisieren die Clients die entfernten Objekte?

- der Client bekommt eine Zeichenkette zugeschickt, die in eine Objektreferenz umgewandelt wird. Anschließend wird die Verbindung aufgebaut.
- Client durchsucht den Namensdienst von CORBA nach dem Namen des Objekts
- Client fragt die gelben Seiten von CORBA (ein Trader-Dienst) ab und sucht nach einer Funktionalität unter Angabe von Metainformationen

dynamisches CORBA (2)

1. `get_interface()` liefert Referenz auf Objekt im Schnittstellenverzeichnis, das diese Schnittstelle beschreibt
   → Einstieg für Navigation im Schnittstellenverzeichnis
2. Methodenbeschreibung verschaffen:
   - `lookup_name()` findet die gewünschte Methode
   - `describe()` liefert IDL-Definition der Methode
3. Argumentliste erzeugen: `create_list()` und `add_item()`
4. Anfrage erzeugen:
   `Create_request(ObjectReference, Method, Arguments)`
5. Entfernte Objektmethode aufrufen
   - RPC: `invoke()` / Datagram: `send_oneway()`
   - Send/Receive: `send_deffered()` bzw. `get_response()`

Web-basiertes Modell

- HTML/XML
- dynamische Dokumente
  * Common Gateway Interface (CGI)
  * Active Server Pages (ASP)
  * Servlets
- aktive Dokumente
  * JavaScript
  * Applets

Komponentenbasiertes Modell
Enterprise Java Beans – EJB

- Komponenten verpacken Objekte in die nächst höhere, abstrakte Form:

  Container (Behälter) beinhaltet Komposition mehrerer Objekte, die von außen über ein gemeinsames Interface angesprochen werden
- EJB ist kein Produkt, sondern eine Spezifiktion von Sun
- Komponententechnologie basierend auf Java:
  * client-seitig: JavaBeans
  * server-seitig: Enterprise JavaBeans

JavaBeans

Java-Klassen, die gewisse Konventionen erfüllen: umfassen das Erzeugen (Instanziieren) von Objekten, den Zugriff und Festlegen der Eigenschaften.

- mindestens ein Konstruktor der Bean wird ohne Übergabeparameter implementiert
- jede *Getter*-Methode (fragt einen Eigenschaftswert ab) beginnt mit `get`, endet mit dem Namen der Eigenschaft und liefert den entsprechenden Datentyp zurück
- eine *Setter*-Methode (legt Wert einer Eigenschaft fest) hat als Rückgabewert `void`, beginnt mit `set` und endet mit dem Namen der entsprechenden Eigenschaft

Die Fähigkeiten einer Bean werden in einer Tabelle (Eigenschaftsliste) aufgeführt.

JavaBeans (2)

JavaBeans sind lokal und nicht verteilt. Technologie zum Erzeugen von generischen Java-Komponenten.

JavaBeans besitzen

- zum Erzeugen ein Properties Interface: erlaubt einem Erzeuger-Werkzeug (Builder Tool) das Feststellen der Beans-Funktionalität
- eine Informationsklasse `BeanInfo`
- Editoren zur Festlegung der Eigenschaften der Bean und
- Customizer, die Anpassungen an bestimmte Erfordernisse durchführen
- BDK (Beans Development Kit) enthält alle notwendigen Tools zum Bearbeiten der Beans

Enterprise-Beans

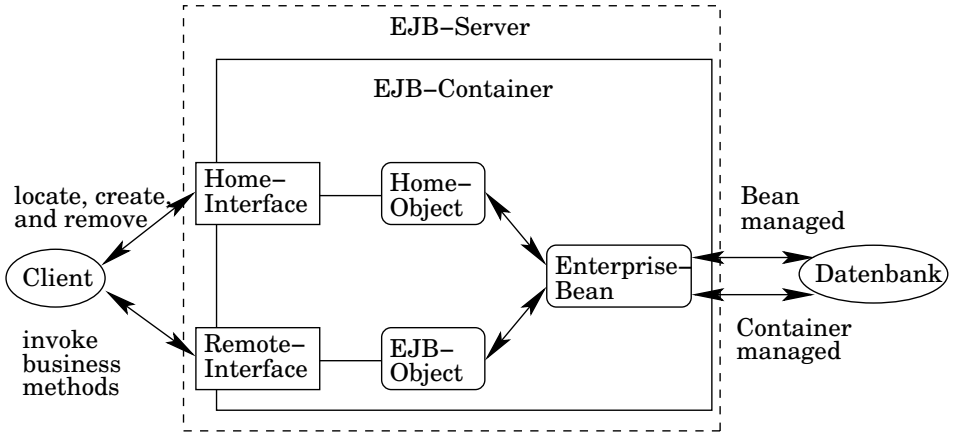
- Komponenten, die vielen Clients auf einem Server ihre Dienste zur Verfügung stellen
- zwei Ausprägungen:
  * Session Beans: für eine Sitzung des Clients
  * Entity Beans: zur Modellierung der Geschäftslogik
- Dienste zur Laufzeit der Enterprise-Beans:
  * Transaktionsdienst
  * Verwaltung von Bean-Instanzen
  * Persistenzunterstützung
  * Sicherheitsdienste (Zugriff über Access Control List)

EJB: Architektur

three tier model:

- mehrere **Clients**
- **EJB-Server:** hier laufen die Enterprise-Beans ab, die EJB-Komponenten sind in EJB-Container eingebettet
- **Datenbasis:** Zugriff erfolgt entweder direkt über JDBC (Java DataBase Connectivity) oder mittels Container im Auftrag der Enterprise-Bean

EJB: Software-Architektur



EJB: Software-Architektur (2)

- der **EJB-Server** stellt Ausführungsumgebung bereit
- der **EJB-Container** verwaltet mehrere EJB-Klassen: ein Client kommuniziert nicht direkt, sondern durch Home- bzw. Remote-Interface mit einer Enterprise-Bean
- **Home-Interface:** definiert life cycle methods (anlegen, entfernen und finden einer Bean)
- **Home-Object:** implementiert das Home-Interface
- **Remote-Inerface:** definiert Methoden, die nach außen hin angeboten werden
- **EJB-Object:** implementiert das Remote-Interface
- **Client:** Java-Applikation, Applet, Servlet, CORBA-Applikation oder mittels COM-CORBA-Brücke angebundene DCOM-Applikation