Verteilte Systeme

Jürgen Kleinöder

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.cs.fau.de

Sommersemester 2013

http://www4.cs.fau.de/Lehre/SS13/V_VS

Überblick

7 Verteilte Anwendungen und Middleware
7.1 Verteilte Anwendungen
7.2 Middleware

Verteilte Anwendungen, Interaktionsformen

- Klassifikation von Interaktionsformen
  - explizit
  - implizit
  - orthogonal
  - nicht-orthogonal
  - uniform
  - nicht-uniform
  - transparent
  - nicht-transparent

explizite, orthogonale Interaktion

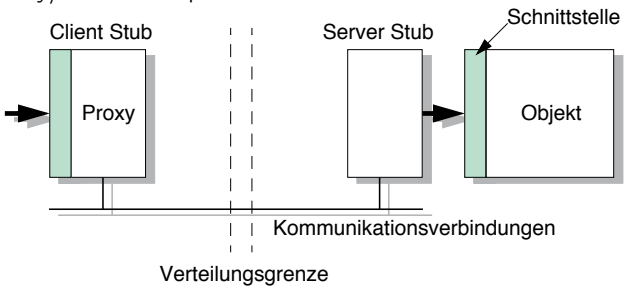
- weit verbreitete Vorgehensweise
- von klassischen Interprozesskommunikations-Mechanismen geprägt
  - Nachrichten (Datagramm-Sockets, Messages, -)
  - Verbindungen (Stream-Sockets, Pipes, -)

\+ Vorteile
- meist weit verbreitete, etablierte Infrastruktur vorhanden
- kein „unsichtbarer“ Overhead

– Nachteile
- Programmierung aufwändig
- Bruch im Programmierparadigma (vor allem bei Objektorientierung)
  - Serialisierung von Parametern, Verlust von Typ-Information
- Verteilung wird durch die Programmierung „fest verdrahtet“
  - Software sehr unflexibel in Bezug auf Änderungen

implizite, nicht-orthogonale Interaktion

- Interaktion zwischen lokalen und verteilten Funktionen oder Objekten unterscheidet sich prinzipiell nicht
  - nur ein Interaktionsmechanismus: Funktions-/Methodenaufruf
- grundlegendes Konzept: Remote Procedure Call
  - Verteilung wird durch Vermittler- oder Stellvertreterobjekte vor den Kommunikationspartnern (weitgehend) verborgen
  - Proxy/Stub-Prinzip



uniforme / nicht-uniforme Interaktion

- ⋆ nicht-uniforme Interaktion
  - unterschiedliche Methodenaufrufe für lokale und remote-Referenzen
  - unterschiedliche Semantik bei der Parameterübergabe
    - by reference, by value
    - Problem: Übergabe von lokalen Objektreferenzen
- ⋆ uniforme Interaktion
  - keinerlei Unterschied zwischen lokalen und remote-Aufrufen

transparente / nicht-transparente Verteilung

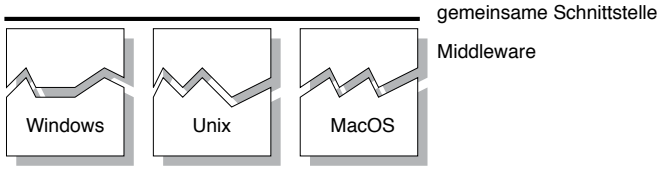
- volle Transparenz:
  Anwendungsentwickler sieht keinerlei Unterschied zwischen lokalen und verteilten Objekten
- Probleme:
  - im verteilten Fall können spezielle Fehler auftreten
    - unabhängiger Objektausfall → Remote Exception
  - verteilte Interaktion ist implizit signifikant teurer
    - Transparenz kann zu Ineffizienz führen
  - Verteilung ist häufig ein Entwurfskriterium
    - Verbergen der Verteilung in der Implementierung ist unsinnig
- Fazit:
  - bei der Programmierung sollte zwischen potentiell verteilten und definitiv lokalen Objekten unterschieden werden können

Herausforderungen

- Überwindung heterogener Hardware- und Softwarestrukturen
  - verschiedene Hardware
  - verschiedene Betriebssysteme
  - verschiedene Programmiersprachen
- Ortstransparenz
  - statische Konfiguration
  - Objektmigration
- Globale Dienste
  - z.B. Namensdienste, Transaktionsdienst, Persistenz, Kontrolle der Nebenläufigkeit

Middleware für verteiltes Programmieren

- Middleware - Software zwischen Betriebssystem und Anwendung



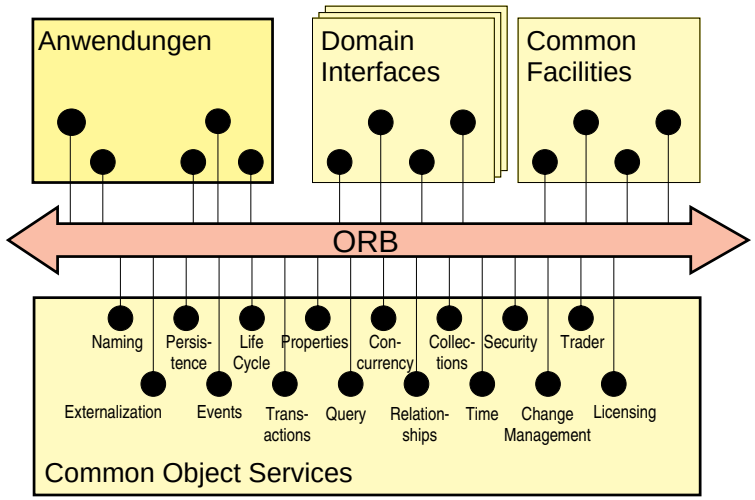
  - Bereitstellung von Abstraktionen und Diensten für verteilte Anwendungen
  - „Betriebssystemschnittstelle“ für ein verteiltes System
- CORBA
  - plattformunabhängige Middleware-Architektur für verteilte Objekte
  - Standard der OMG (Object Management Group)
  - erste umfangreiche Normierung von Middleware-Konzepten

Motivation für CORBA

- Verteilte objektbasierte Programmierung
  - verteilte Objekte mit definierter Schnittstelle
- Heterogenität in Verteilten Systemen
  - verschiedene Hardware-Architekturen
  - verschiedene Betriebssysteme und Betriebssystem-Architekturen
  - verschiedene Programmiersprachen

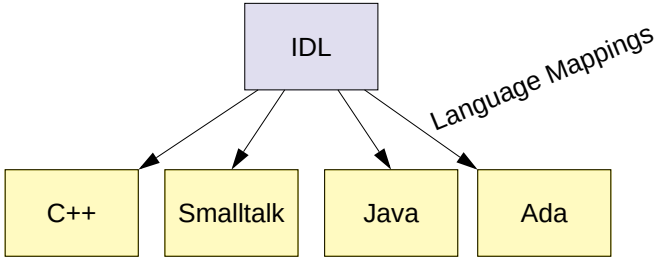
  ⇒ Transparenz der Heterogenität
- Dienste im verteilten System
  - Namensdienst, Zeitdienst,...
- Was ist CORBA?
  - **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
    - plattformunabhängige Middleware-Architektur für verteilte Objekte
    - Sammlung von Standard-Dokumenten verwaltete durch die OMG
    - `http://www.omg.org/spec/CORBA/3.1`

OMA – Object Management Architecture



Interface Definition Language (IDL)

- Sprache zur Beschreibung von Objekt-Schnittstellen
  - unabhängig von der Implementierungssprache des Objekts
  - Sprachabbildung (Language-Mapping) definiert, wie IDL-Konstrukte in die Konzepte einer bestimmten Programmiersprache abgebildet werden
  - Language-Mapping ist Teil des CORBA standards
  - Language mappings sind festgelegt für:
    - Ada, C, C++, COBOL, Java, Lisp, PL/I, Python, Smalltalk
    - weitere inoffizielle Language-Mappings existieren,z.B. für Perl



Interface-Definition-Language

- IDL angelehnt an C++
  - geringer Lernaufwand
- eigene Datentypen
  - Basistypen
  - Aufzählungstyp
  - zusammengesetzte Typen
- Module
  - definieren hierarchischen Namensraum für Anwendungsschnittstellen und -typen
- Schnittstellen
  - beschreiben einen von außen wahrnehmbaren Objekttyp
- Exceptions
  - beschreiben Ausnahmebedingungen

Interface-Definition-Language

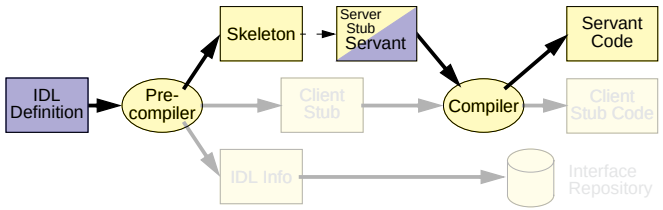
- Umsetzung von IDL in Sprachkonstrukte (z.B. Java)

```
// IDL module MyModule interface MyInterface attribute long
lines; void printLine( in string toPrint ); ; ;
```

```
//Java package MyModule;
public interface MyInterface extends ... public int lines(); public
void lines( int lines ); public void printLine( java.lang.String toPrint
); ... ;
```

Objekterzeugung

- Ablauf auf Serverseite
  - Beschreibung der Objektschnittstelle in IDL
  - Programmierung des Server-Objekts in der Implementierungssprache
- Stub/Skeleton-Erzeugung

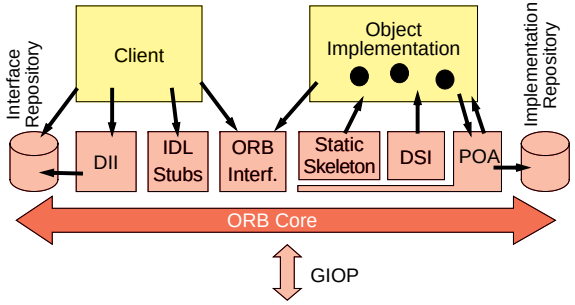


Objekte Binden

- Ablauf auf Serverseite
  - Registrierung des Objekts am **ORB** (bzw. dem Portable-Object-Adaptor (POA))
    - der ORB erzeugt eine Interoperable Object Reference (IOR)
- Binden des Clients an das Server-Objekt
  - Referenz auf Server-Objekt besorgen
    - Rückgabewert eines Methodenaufrufs
    - Ergebnis einer Namensdienst-Anfrage
    - von 'ausserhalb' des Systems: Benutzer kennt Referenz als String (IORs können in Strings konvertiert werden und umgekehrt)
- Erzeugung des Client-Stub & Methodenaufruf über Client-Stub

ORB Architektur

- **O**bject **R**equest **B**roker – ORB
  - ist das Rückgrat einer CORBA-Implementierung
  - vermittelt Methodenaufrufe von einem zum anderen Objekt
    - ... innerhalb/zwischen Adressräumen/Prozessen von (verschiedenen) ORBs



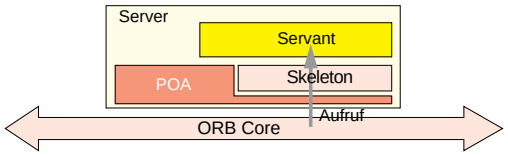
- Zentrale Komponenten eines ORB
  - mehrere interne Komponenten (teilweise nicht für Anwender sichtbar)

Portable-Object-Adaptor (POA)

- Aufgaben des Objektadapters
  - Generierung der Objektreferenzen
  - Entgegennahme eingehender Methodenaufruf-Anfragen
  - Weiterleitung von Methodenaufrufen an den entsprechenden Servant
  - Authentisierung des Aufrufers (Sicherheitsfunktionalität)
  - Aktivierung und Deaktivierung von Servants
- Portabilität von Objektimpl. zwischen verschiedenen ORBs
- Trennung von Objekt (CORBA-Objekt mit seiner Identität) und Objektimplementierung
  - eine Objektimplementierung kann **mehrere** CORBA-Objekte realisieren
  - Objektimplementierung kann bei Bedarf **dynamisch aktiviert** werden
  - persistente CORBA-Objekte (Objekte, die Laufzeit eines Servers überdauern)
  - eine Object Reference beim Client steht für ein bestimmtes CORBA Objekt – **nicht unbedingt** für einen bestimmten Servant!

Portable-Object-Adaptor (POA)

- Jeder Servant kennt seine POA-Instanz
  - POA-Instanz kennt die an ihm angemeldeten Objekte
  - POA stellt Kommunikationsplattform bereit und nimmt Aufrufe entgegen (für seine Objekte)



- Mehrere POA-Instanzen (mit unterschiedlichen Strategien) innerhalb eines Servers möglich
- Beispiel: Lifespan policy
  - `TRANSIENT` für Objekte, die Servant nicht überleben
  - `PERSISTENT` für Objekte, die POA und Servant überleben können