Verteilte Systeme

J. Reichardt
FBI / FHD

VERTEILTE SYSTEME

J. Reichardt

Themen

1. Abschnitt

- **Einleitung**
  (Beispiel, Definition, Charakteristische Merkmale)
- **Kommunikation in Verteilten Systemen**
- **Das Client-Server-Modell**
  (Szenarien, Client-Server-Dialog, RPC, RPC-Implementierung)
- **Die Broker-Architektur**

Praktikum

Common Object Request Broker Architecture
(CORBA, OmniORB AT & T)

Literatur

1. **G. Coulouris et al.:**
   Distributed Systems - Concepts and Design
   2. Auflage, Addison-Wesley 1994

2. **F. Buschmann et al.:**
   A System of Patterns - Pattern-oriented Software Architecture
   Wiley 1996

3. **J. Zimmermann et al.:**
   Verteilte Komponenten und Datenbankanbindung
   Addison-Wesley 2000

4. **M. S. Henning et al.:**
   Advanced CORBA Programming with C++
   Addison-Wesley 1999

5. **AT & T omniORB:**
Produktinformation
www.uk.research.att.com/omniORB

6. **R. Monson-Haefel:**
Enterprise JavaBeans
O'Reilly 2000

Erste Definition

Technische Sicht

Ein **Verteiltes System** besteht aus

- einer Ansammlung autonomer Programme, die
- auf unterschiedlichen Rechnern plaziert sind und
- ohne zentrale Kontrolle
- miteinander kooperieren

Beispiel

Konkrete Beispiele sind verteilte Anwendungen zur Automatisierung von Bürovorgängen, zur Steuerung von Fertigungsabläufen oder zur Verwaltung von Management-Information. Diese Bereiche sind gekennzeichnet durch eine inhärente physische Verteilung ihrer Datenhaltungs- und Verarbeitungskomponenten; z.B. sind Fertigungsvorgänge auf verschiedene Maschinen verteilt, die oft dezentral kontrolliert werden.

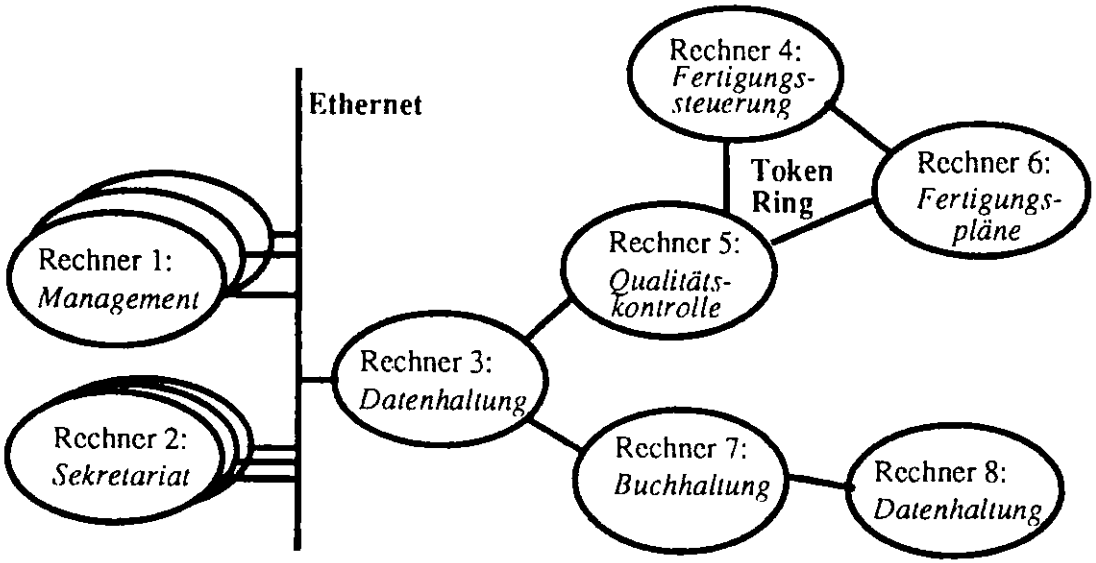


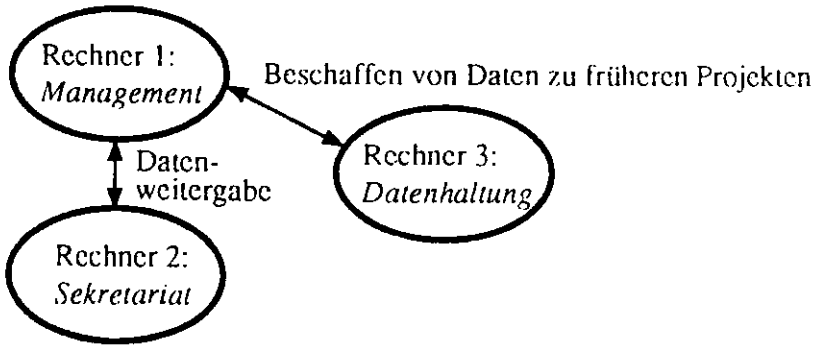
Abb. 1-1 zeigt ein Beispiel einer verteilten Anwendung aus dem Gesamtbereich der Büro- und Fertigungsautomatisierung.

Beispiel

Fortsetzung

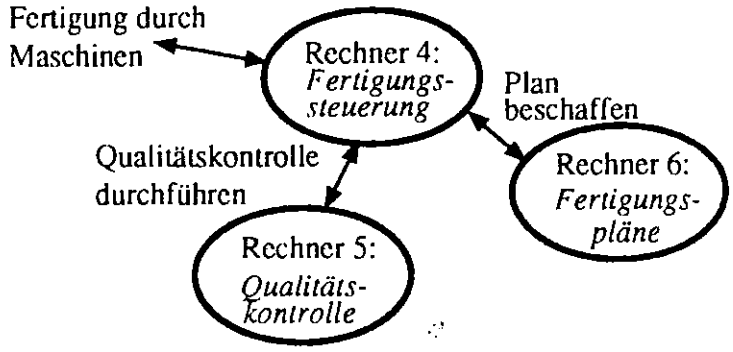
Projektplanung

Ein Manager (auf Rechner 1) beschafft sich Daten zu früheren Produkt-Entwicklungsprojekten von der entfernten Komponente zur Datenhaltung (auf Rechner 3), etwa durch Aufruf einer entfernten Datenbankoperation. Danach werden lokale Planungsaufgaben unter Einsatz eines Spreadsheet-Programms durch den Manager durchgeführt. Gegebenenfalls sendet der Manager im Verlauf dessen Informations- oder Anfragenachrichten an andere Manager auf anderen Rechnern; auf diese Weise können z.B. Koordinationsfragen entfernt geklärt werden. Nach Abschluß einer groben Produktplanungsphase sendet der Manager die resultierenden Daten zur weiteren Überarbeitung an seine Sekretärin (Rechner 2), indem er eine entfernte Speicherprozedur dort aufruft. Danach folgen zahlreiche Interaktionen mit anderen Abteilungen zur Feinplanung, z.B. mit der technischen Entwicklungsabteilung, die hier nicht gezeigt sind.



Fertigung

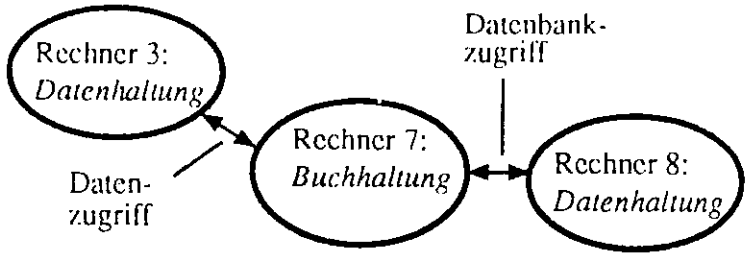
Im Rahmen der Fertigung von Produkten wird die Fertigungssteuerung (Rechner 4) mit der Koordination beauftragt. Je nach Fertigungsauftrag beschafft sie sich einen zugehörigen Fertigungsplan von Rechner 6, der eine Datenbasis von Fertigungsplänen verwaltet. Anschließend werden ggf. mehrere Maschinensteuerungen mit den einzelnen Fertigungsschritten beauftragt, die wiederum auf verschiedene Rechner verteilt sein können (in der Abbildung nicht gezeigt). Schließlich nimmt die Fertigungssteuerung die Fertigmeldungen der Maschinen entgegen und beauftragt die Qualitätskontrolle (Rechner 5) mit der Abschlußprüfung des gefertigten Produkts.



Beispiel
Fortsetzung

Buchhaltung

Die Buchhaltung (Rechner 7) ist zunächst recht konventionell als eine zentrale Instanz realisiert. Die von ihr verwendeten Daten (Rechnungen, Lieferscheine etc.) werden aber z.B. wegen ihres großen Umfangs durch eine separate, entfernte Datenbank auf Rechner 8 verwaltet. Es ist daher erforderlich, dort entfernte Operationen zum Anfordern und Abspeichern von Daten aufzurufen. Außerdem greift die Buchhaltung teilweise auch auf die Dienste der Datenhaltung auf Rechner 3 zu, um z.B. bei Bedarf eine Zuordnung zwischen Rechnungstiteln und Produktbeschreibungen durchführen zu können.



Zusammenfassung

Am Beispiel wurde aufgezeigt, daß eine bestimmte Plazierung von Anwendungskomponenten auf Rechnerknoten vorgenommen wird. Es ist auch offensichtlich, daß geeignete Kommunikationsmechanismen zwischen den Komponenten bereitstehen müssen.

Außerdem wird schon hier deutlich, daß verschiedene Abläufe weitgehend entkoppelt und dadurch auch parallel ausgeführt werden können. Es ist auch offensichtlich, daß manche Dienste bzw. Betriebsmittel (z.B. die Projektdatenbank) von mehreren Komponenten verwendet werden und dadurch eine verbesserte Integration erzielt wird. Andererseits kann die Verwaltung komponentenspezifischer Daten (z.B. des Managements) stärker dezentralisiert und damit leichter durch die jeweilige Komponente kontrolliert werden.

Zweite Definition

Benutzer-Sicht

Ein System bezeichnet man als **verteilt**, wenn

- es sich seinen Benutzern als "zentrales" System zeigt, aber
- verschiedene, "voneinander unabhängige" Prozessoren benutzt.

Schlüsselkonzept ist das der T r a n s p a r e n z (die Verwendung mehrfacher Prozessoren ist für den Benutzer unsichtbar, d.h. transparent).

Der Benutzer sieht das System als ein "virtuelles Ein-Prozessorsystem", und nicht als eine Ansammlung verschiedener Prozessoren.

"A distributed system is one that stops you from getting any work done when a machine you've never heard of crashes" (Lamport).

Transparenz

in ihren unterschiedlichen Formen

- **L o k a l i t ä t**

  Gleichberechtigte Nutzung von lokalen und entfernten Resourcen. Ihre Position ist dem Benutzer i.A. unbekannt.

- **V e r v i e l f ä l t i g u n g**

  Verteilung und Repliken von Resourcen sind dem Benutzer verborgen. Er sieht nur e i n e logische Resource.

- **K o n k u r r e n z**

  Der Benutzer sieht nicht den konkurrierenden Zugriff anderer Benutzer auf dieselben physikalischen Resourcen.

- **H e t e r o g e n i t ä t**

  Der Benutzer sieht nicht die Verschiedenartigkeit der Hardware, Betriebssysteme, Programmiersprachen, etc.

- **F e h l e r**

  Das System verbirgt und behebt Fehler, die durch den Zugriff auf entfernte Resourcen entstehen

Abgrenzung

Rechnernetz vs. Verteiltes System

Die Unterscheidung der Begriffe Rechnernetz und Verteiltes System hängt wesentlich von dem Grad der Transparenz ab, mit der durch den Systemverbund den Endnutzern Dienste erbracht werden.

Verteiltes System

Braucht sich der Endnutzer um die Lokalität der Diensterbringung nicht zu kümmern, existiert also ein (Netz-)Betriebssystem, das die zu einer Realisierung einer Anwendung notwendigen Verarbeitungsschritte koordiniert, die eventuell auf verschiedenen Systemen des Verbundes ablaufen (verteilte Anwendung), so nennen wir den Systemverbund ein *Verteiltes System (Distributed System)*.

Rechnernetz

Bleibt die Verbindungsstruktur des Systemverbundes sichtbar, müssen also die Endsysteme des Verbundes und die verteilten Anwendungsprozesse über Adressen angesprochen werden, so wird vom *Rechnernetz (Kommunikationsnetz, Communication Network)* gesprochen, wobei die Verbindungsstrukturen oft auch *Transportnetz (Connection Network, Transport System)* genannt werden.

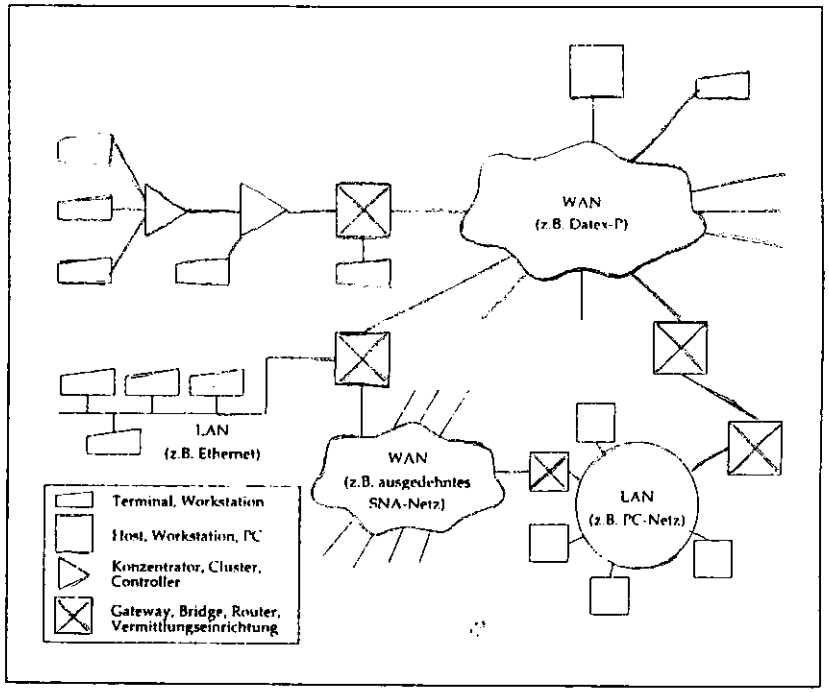


Bild 1-1: Typisches Rechnernetz

Merkmale

eines verteilten Systems

1) Mehrfache CPU's

2) Verbindungs-Hardware

3) Unabhängiges Fehlerverhalten der CPU's

4) Gemeinsamer Systemzustand

K e i n Beispiel für ein Verteiltes System

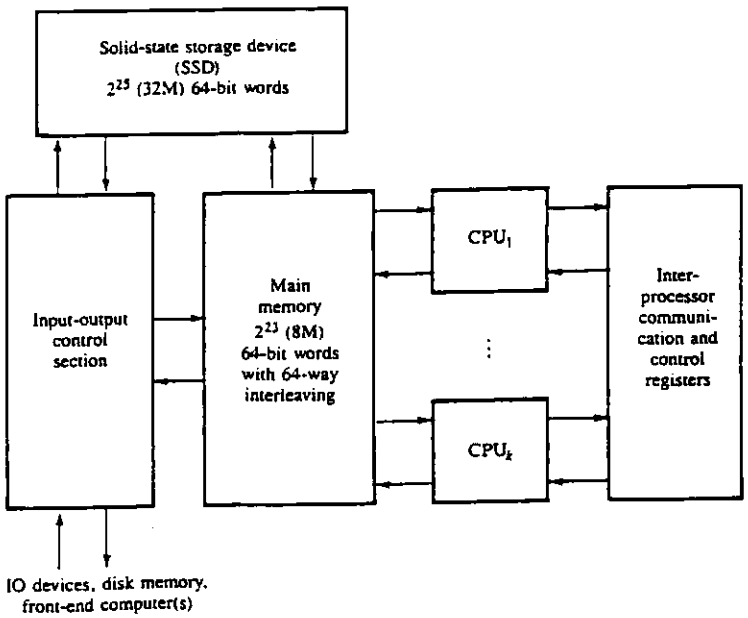
"M u l t i p r o z e s s o r - C o m p u t e r m i t S h a r e d - M e m o r y"

- mehrfache Prozessoren
- Shared-Memory (= gemeinsamer Systemzustand)
- Interrupts zwischen den Prozessoren, sowie Memory-Bus (= Kooperation über Signale und Speicher)

a b e r

k e i n e F e h l e r u n a b h ä n g i g k e i t !!!

Zu eng gekoppelt !



System organization of the Cray X-MP.

K e i n Beispiel für ein Verteiltes System

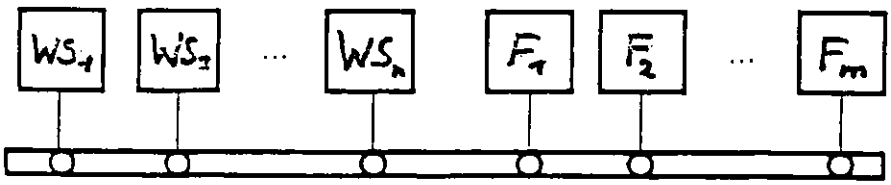
"P l a t t e n - l o s e W o r k s t a t i o n s
m i t F i l e - S e r v e r n"

- jede WS und jeder FS besitzt einen Prozessor mit Arbeitsspeicher
- es gibt Netzwerk-Verbindungen
- es besteht Fehler-Unabhängigkeit

a b e r

es gibt k e i n e n g e m e i n s a m e n S y s t e m z u s t a n d !!!

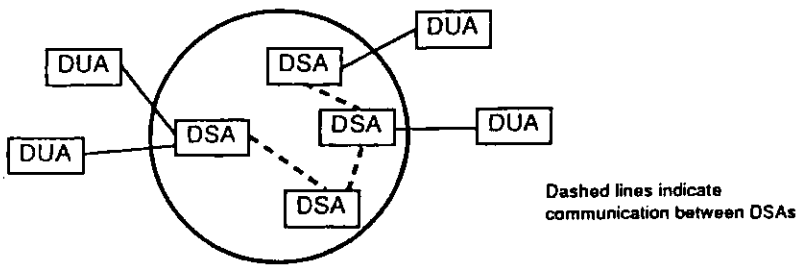
Zu lose gekoppelt !



B e i s p i e l für ein Verteiltes System

"D i r e c t o r y S e r v i c e"

- DS-Agents auf verschiedenen Netzknoten mit lokaler Rechnerleistung und Datenbank
- Austausch von Directory-Information über ein Directory-System-Protokoll (DSP) unter Benutzung öffentlicher Netze
- Gemeinsamer Systemzustand in Form von lokalen Directory-Informationsbasen und Querverweisen
- Fehlerunabhängigkeit

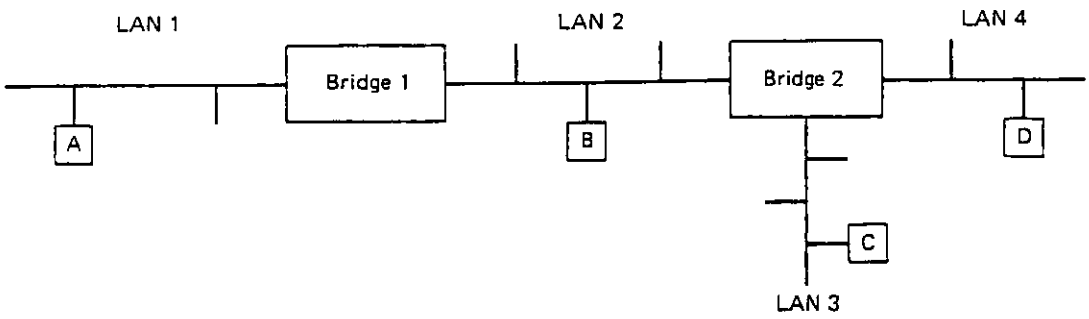


Model of the Directory and its users.

B e i s p i e l für ein Verteiltes System

"E T H E R N E T - S e g m e n t e m i t
t r a n s p a r e n t e n B r i d g e s"

- Bridges enthalten Prozessoren und lokalen Arbeitsspeicher
- Verbindungen über die Kabelsegmente
- Ausfall einer Bridge führt zu einer Änderung der Routing-Tabellen in den verbleibenden Bridges
- Gemeinsamer Systemzustand in Form der Routing-Tabellen



A configuration with four LANs and two bridges.

Nutzen

Verteilter Systeme

- *Dezentralisierung und Lokalität:* Durch die Verteilung der Module wird eine weitgehend dezentrale Verarbeitung möglich. Dadurch kann eine hohe Lokalität in bezug auf die Zuordnung von Daten und bearbeitenden Operationen erzielt werden. Dies wiederum verbessert die Laufzeiteigenschaften und erleichtert die organisatorische Verwaltung.
- *Parallele Verarbeitung:* Da jeder beteiligte Rechner eigene Betriebsmittel, insbesondere Prozessor und Speicher, bereitstellt, können die einzelnen Module stark parallel arbeiten; dadurch wird die Gesamtbearbeitungszeit eines Auftrags verkürzt.
- *Fehlertoleranz:* Die beteiligten Rechner sind weitgehend autonom; dies gilt auch für Systemausfälle; bei Ausfall eines Rechners bleibt der übrige Teil der Anwendung auf anderen Rechnern funktionsfähig. Explizite Fehlertoleranz kann zusätzlich durch Replikation von Daten und Operationen erzielt werden.
- *Gemeinsame Betriebsmittelnutzung:* Da verschiedene Rechner über ein Kommunikationsnetz gekoppelt sind, können sie auch gemeinsam auf die Ressourcen eines bestimmten Rechners zugreifen. Dadurch wird die gemeinsame Nutzung teurer Peripheriegeräte (z.B. Drucker, Plotter oder Speichermedien) sowie komplexer Softwarekomponenten (z.B. Datenbanken) ermöglicht.
- *Integration von Teilanwendungen:* Die Gesamtfunktionalität existierender Teilanwendungen kann oft durch ihre Integration zu einer verteilten Anwendung gesteigert werden; beispielsweise kann eine Management-Datenbank mit einem Spreadsheet-System gekoppelt werden, um automatisierte Datenauswertungen durchzuführen.

Spezielle Probleme bei verteilten Anwendungen

- *Systemgröße:* Die Kopplung von Teilanwendungen bzw. der integrierte Entwurf einer Gesamtanwendung führt oft zu sehr großen Softwaresystemen, die nur durch spezielle Programmier- und Modularisierungstechniken beherrschbar sind.

- *Parallelität und Indeterminismus:* Bedingt durch die parallele Verarbeitung und durch nicht vorhersagbare Nachrichtenlaufzeiten zwischen Rechnern treten Indeterminismen und in Konflikt stehende Datenzugriffe auf. Hieraus resultiert die Anforderung nach speziellen Synchronisationsmechanismen und Testwerkzeugen.

- *Kommunikationsmechanismen:* Die Kommunikation wird in vielen verteilten Systemen über sehr maschinennahe Mechanismen abgewickelt. Da dies schwer handhabbar und fehleranfällig bei der Programmierung ist, müssen höhere Mechanismen angeboten werden, die stärkere Transparenz bezüglich der Verteilung aufweisen.

- *Heterogenität:* Fast alle praxisnahen verteilten Rechnerumgebungen umfassen Rechner unterschiedlicher Hersteller mit verschiedenartigen Datendarstellungen und Befehlssätzen, Betriebssystemen, Programmiersprachen und Kommunikationsmechanismen. Um trotz der resultierenden Heterogenität eine Kooperation zu ermöglichen, sind spezielle Adaptionstechniken von zentraler Bedeutung.

Spezielle Probleme

Fortsetzung

- *Schutz und Sicherheit:* In großen verteilten Systemen bestehen sehr viel mehr Möglichkeiten für unberechtigte Zugriffe und Datenmanipulationen über das Netzwerk, als es bei herkömmlichen Anwendungen der Fall ist. Diesen essentiellen - und im wirtschaftlichen Wettbewerb bedeutsamen - Problemen kann nur durch wirksame Techniken des verteilten Datenschutzes begegnet werden.

- *Namensverwaltung:* In verteilten Anwendungen ist es wichtig, kommunizierende Module mit logischen Namen zu versehen - und sie nicht etwa durch physikalische Netzadressen oder interne Kennummern gegenüber dem Anwender zu identifizieren. Nur so werden benutzerfreundlicher Umgang mit einer verteilten Anwendung und Möglichkeiten der dynamischen Erweiterung und Umstrukturierung von Anwendungskomponenten gewährleistet. Dies erfordert umfangreiche Mechanismen zur Namensverwaltung und Namensinterpretation.

- *Fehlerbehandlung:* Die angesprochenen Vorteile in bezug auf Fehlertoleranz kommen nur zum Tragen, wenn eine entsprechende Fehlerbehandlung vorhanden ist. Dabei müssen insbesondere unabhängige Ausfälle verschiedener Rechner sowie der zugehörigen Kommunikationsmedien berücksichtigt werden.

- *Systemadministration:* Bedingt durch die Heterogenität und Größe eines verteilten Systems bzw. einer darauf ablaufenden verteilten Anwendung wird die Systemadministration deutlich erschwert. Daher müssen dem Systemmanager komfortable und möglichst systemweit einheitliche Administrationswerkzeuge und -schnittstellen angeboten werden. Diese müssen vor allem auch gezielt Unterstützung für verteilungsspezifische Probleme bieten; als Beispiel sei die Administration eines verteilten Namensraums oder eines verteilten Dateidienstes genannt

Vergleich

Verteilte vs. Zentrale Systeme

Vorteile VS

- Resource Sharing
  geogr + org übergreifend
- Kosten
  Kleine Endsysteme
- Wachstum
  Kleine Schritte, großer Bereich
- Unabhängigkeit
  Beschaffung, Hersteller

Vorteile ZS

- Zugänglichkeit
  gleich für alle Informationen und Ressourcen
- Einheitlichkeit
  Funktionsausführung, Objektnamen
- Management
  homogen

---

etwa gleichauf in Punkto "Sicherheit" und "Verfügbarkeit"

Vergleich

Verteilte vs. Zentrale Systeme

Verfügbarkeit

VS

Nachteile:

- höhere Ausfallwahrscheinlichkeit durch kooperierende, voneinander unabhängige Komponenten
- Kommunikationsfehler

Vorteile:

- gezielte Ausnutzung der fehlerunabhängigkeit reduziert die Ausfallwahrscheinlichkeit
- bei Systemausfall sind nicht alle Benutzer betroffen

ZS

Vorteile:

- gut kontrollierbare, und damit stabile Hardware und Betriebssysteme

Nachteile:

- bei Systemausfall sind alle Benutzer betroffen

Vergleich

Verteilte vs. Zentrale Systeme

Sicherheit

VS

Nachteile:

- viele Sicherheitsbereiche
- unterschiedliche physikalische Absicherung
- unterschiedliche Sicherheits-strategien
- unterschiedliche Kontroll-instanzen

Vorteil:

- durch eine einzelne Schwach-stelle ist **nicht** das gesamte System gefährdet

ZS

Vorteile:

- ein einziger Sicherheits-bereich
- eine einzige Kontrollinstanz
- ein einziges Betriebssystem
- ein einziger Rechner in phy-sikalisch sicherer Umgebung

Nachteil:

- durch eine einzige Schwach-stelle wird das **gesamte** System gefährdet

BOB

Best of Both Worlds

ein idealisiertes Beispiel aus /Mullender/; vereinigt die Vorteile zentraler und dezentraler Systeme; besitzt zusätzlich eine hohe Sicherheit und Verfügbarkeit

Merkmale

- heterogene Hardware, Software und Information
- flexibles Wachstum
- vernetzt
- verfügt über eine einheitliche Menge von Diensten (Naming, entfernter Funktionsaufruf, Benutzerregistrierung, Zeit, Files, ...)
- besitzt globale Eigenschaften (Namen, Zugänglichkeit, Sicherheit, Verfügbarkeit, Management)

BOB

Globale Eigenschaften

1) Globale Namen

Global eindeutige Namen für Benutzer, Maschinen, Files, Dienste, etc., mit systemweiter Gültigkeit

2) Globaler Zugriff

Programme sind – unabhängig von den benötigten Ressourcen – überall ausführbar

3) Globale Sicherheit

systemweit gültige Sicherungsprozeduren bzgl. Logins, Zugriff auf Files, Drucker, ...

4) Globales Management

systemweit, von einem Punkt des Systems aus

5) Globale Verfügbarkeit

systemweit, auch im Fehlerfall

BOB

Basis-Dienste

1) **Name Service**
Verteilte Datenbasis für globale Namen von Maschinen, Diensten, Benutzern, Files, etc.

2) **Remote Procedure Call**
Einheitliches Basisprotokoll für den entfernten Funktionsaufruf

3) **Benutzer-Registrierung**
Authentifizierung, Zugriffszertifikate

4) **Zeit**
Synchronisation, Genauigkeit

5) **Files**
einheitliche Service-Schnittstelle

6) **Management**
einheitliche Sicht auf alle Komponenten

BOB

zusätzliche Services, je nach Bedarf

1) **Records**
strukturierter Zugriff, Locking

2) **Drucker**
Standardformate (Postscript, ...)

3) **Remote Job Entry**
Interaktive und Batch Jobs

4) **Mailboxes**
Electronic Mail

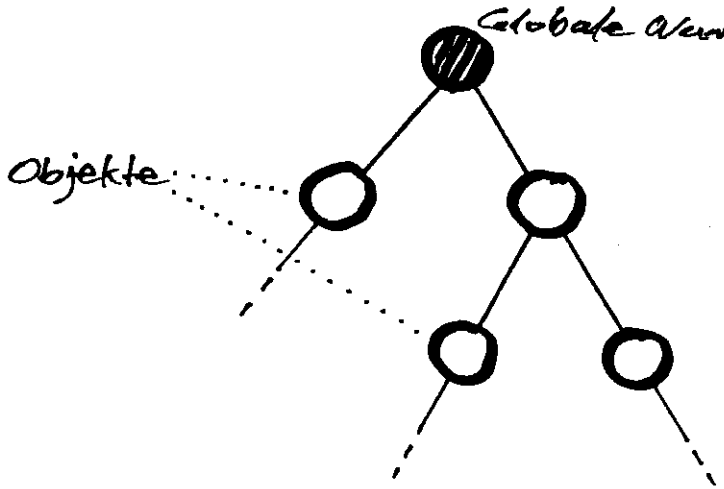
5) **Terminals**
GUI-Zugang für beliebige Anwendungen

6) **Accounting**
Abrechnung, Überwachung

BOB

Namens-Modell

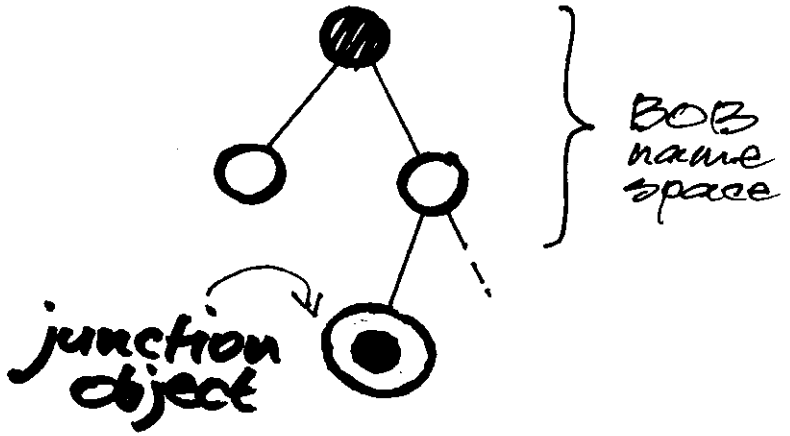
Globaler Namensbaum



Objekte:
- Directories
- Benutzer
- Server
- Gruppen
- ⋮

"junction"

Integration unterschiedlicher Namensräume



enthält:
- eine Menge von Servern
- Auswahlregeln für diese Menge
- eine Service Interface ID
- Objekt-Parameter

BOB

Zugriffs-Modell

- Jedes Client-Anwendungsprogramm kann prinzipiell auf jeder BOB-Plattform ausgeführt werden
- Die Programmumgebung ändert sich dabei nicht, d.h., das Client-Programm greift – ortsunabhängig – stets auf dieselben Benutzerdateien über dieselben Namen zu
- Das Ergebnis ist – bis auf Performance – jeweils dasselbe

Voraussetzung

- globale Namen
- Standard-Dienste
- beide exportiert über einen einheitlichen RPC-Mechanismus

BOB

Zugriffs-Modell

Konsistenzgrade

beim **Name Service** (Erhöhung der Performance, Verfügbarkeit und Erweiterbarkeit auf Kosten der Konsistenz)

- kurze Client-Wartezeit bei Updates
- nur ein einziger Server muß verfügbar sein
- skalierbar durch asynchronen Verteilungsvorgang
- vorübergehende Inkonsistenz der Namensbasen

beim **File Service** (Erhöhung der Konsistenz auf Kosten der Performance, Verfügbarkeit, Erweiterbarkeit)

1. Methode "mit caching"
Änderungen werden global erst dann wirksam, wenn das geänderte File geschlossen und Lesekopien erneut geöffnet werden

2. Methode "ohne caching"
Änderungen werden global unmittelbar wirksam. Der Zugriff erfolgt konkurrierend auf eine gemeinsame Schreib/Lesekopie

BOB

Sicherheits-Modell

Anforderungen

1) Authentisierung
Zuverlässige Identifizierung von Benutzern bei jedem Aufruf einer Operation

2) Zugriffskontrolle
Zugriffskontrollisten für alle Ressourcen und alle Operationen auf diesen Ressourcen - Kontrolle bei jedem Operationsaufruf.

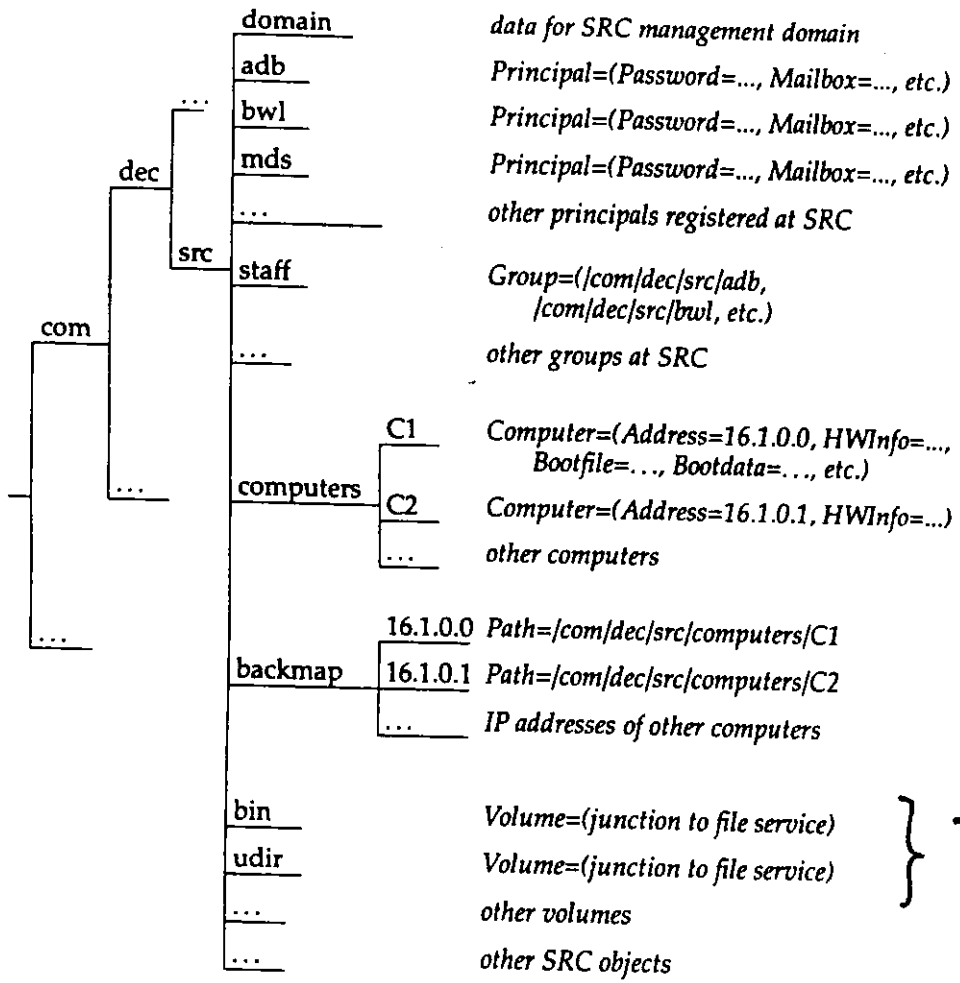
3) Auditing
Protokollieren der Zugriffe auf die Ressourcen; für Nachprüfungen

Basisdienste:

- Name Service: Identifizierung über Namen Zugriffskontrolle für Gruppen
- Time Service: Zeitstempel, Synchronisation und Timer für Authentisierung, Zugriff und "sicheren Kanal"

Beispiel

Global name-space hierarchy in BOB



Einzelbenutzer

Gruppen, z.B. Verteilerlisten, Zugriffskontrollisten, Server-Mengen

'junction' zum File-Service enthält jeweils eine Gruppe von Servern, die dieses File-Volume implementiert

Authentisierung

Urheberschaft und Integrität

Urheberschaft

Nachweis, daß ein Request von einem eindeutig bestimmten Teilnehmer stammt

- Nachweis durch den Teilnehmer, daß er ein "Geheimnis" (pwd) kennt
- Name des Teilnehmers zuverlässig feststellen

Integrität

Nachweis, daß ein Request "unterwegs" nicht modifiziert wurde

Basisfunktionen

- Verschlüsselung
- Digitale Unterschrift

BOB

Management-Modell

"Domains"

Grund-Konzept

- Jede Komponente ist einer Domain zugeordnet
- Für jede Domain ist ein Systemmanager verantwortlich
- Domains sind disjunkt, Manager sind disjunkt
- Domains sind unabhängig voneinander arbeitsfähig

Beispiele

- Komponenten, die einer Benutzergruppe mit gemeinsamen Zielen zugeordnet sind
- größtmögliche Ansammlung von Komponenten unter einem Manager
- Cell-Verbund des einleitenden Beispiels aus dem Bereich Büro- und Fertigungsautomatisierung

BOB

Management-Modell

Anforderungen an die Management-Schnittstelle

1) Entfernter Zugriff

.. auf sämtliche Management-Funktionen und -Daten; sichere Übertragung; Zugriffskontrolle

2) Programmier-Schnittstelle

Zugriff und Auswertung durch Programme - nicht durch Personen; RPC von Management-Tools

3) Relevanz

Selektiver Zugriff auf management-relevante Daten; ggfs. dezentrales Management

4) Einheitlichkeit

Dieselbe Schnittstelle für verschiedenartige Komponenten

BOB

Verfügbarkeits-Modell

Grundprinzip → **Replikate**

Mehrfache Server mit unabhängigem Fehlerverhalten.

Zwei gebräuchliche Schemata:

1) **"Primary/back-up replication"**

Nur ein (Primär-)Server wird benutzt, der im Fehlerfall "sanft" beendet wird; im Fehlerfall Übergang auf einen Back-up-Server

2) **"Active Replication"**

Mehrere Server werden parallel benutzt, um eine Operation auszuführen

Primary / Back-up Replication

unter Ausnutzung des Name-Service

- Auswahl eines "Primary-servers" aus der Menge, die in einem Objekt des Name-Service enthalten ist, unter Verwendung der dort definierten Auswahlregeln
- Fehlererkennung durch Time-out
- Im Fehlerfall: Überwechseln zum nächsten ("Back-up"-) Server der obigen Menge und Wiederholen der Operation
- transparenten Fehlerverhalten und Performance-Verbesserung durch "Clerk-Komponente"

Clerk-Komponente

"Verkapselung" einer Server-Menge

Aufgaben

- Zugriff auf Name-Services
- Server-Auswahl
- Erkennen des Fehlerfalls und Server-Wechsel
- Caching von Nutz- und Verwaltungsdaten
- Asynchrone Ausführung von Prozeduren

Nutzen

- Export eines logisch zentralen und lokalen Service
- Vereinfachung der Service-Schnittstelle
- Fehlertransparenz
- Performance

BOB

Best-of-Both-Worlds

Zusammenfassung

- Kombination der Vorteile zentraler und dezentraler DV
- hohe Sicherheit und Verfügbarkeit
- Kostengünstig
- benutzer-freundlich

- **realisierbar** nur durch schrittweisen Ausbau und Verbesserung bestehender Systeme

Modularisierung

Kriterien

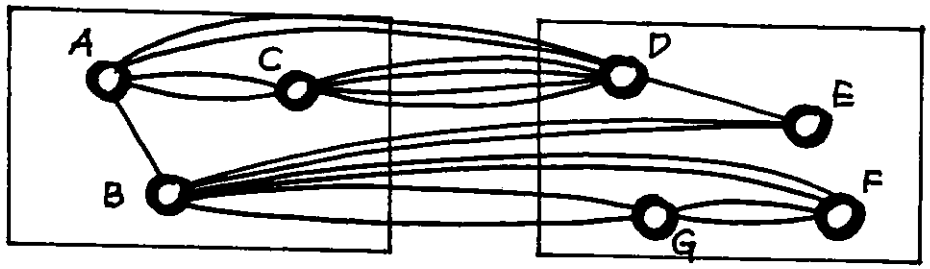
1) Der interne Kommunikationsbedarf eines Moduls ist wesentlich größer als sein externer Kommunikationsbedarf.

2) An der Schnittstelle werden selten Daten ausgetauscht. Die ausgetauschten Datenstrukturen sind einfach.

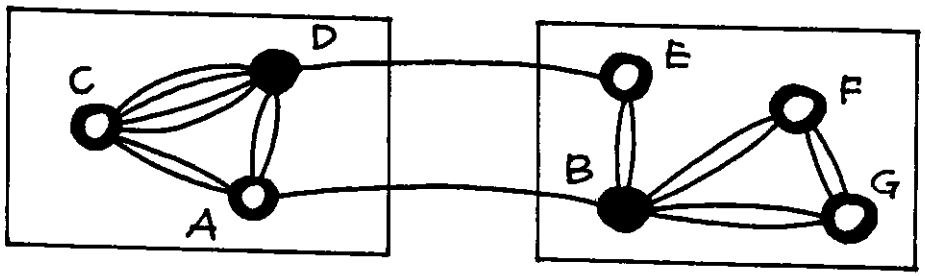
Modularisierung

Beispiel

vorher

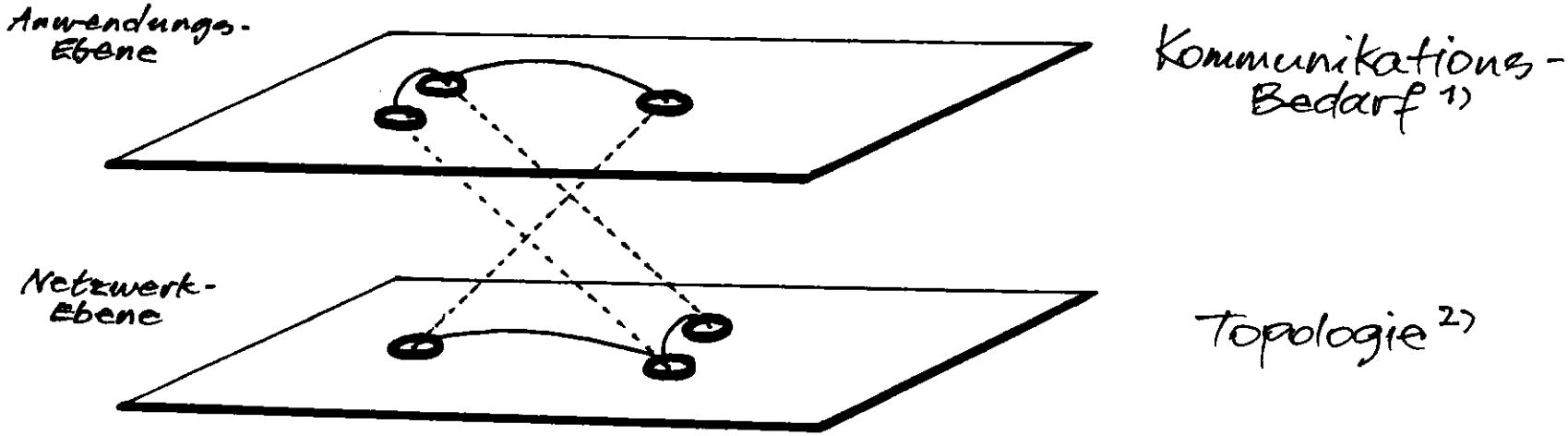


nachher



Plazierung

Abbildung zwischen
Kommunikationsbedarf und Topologie



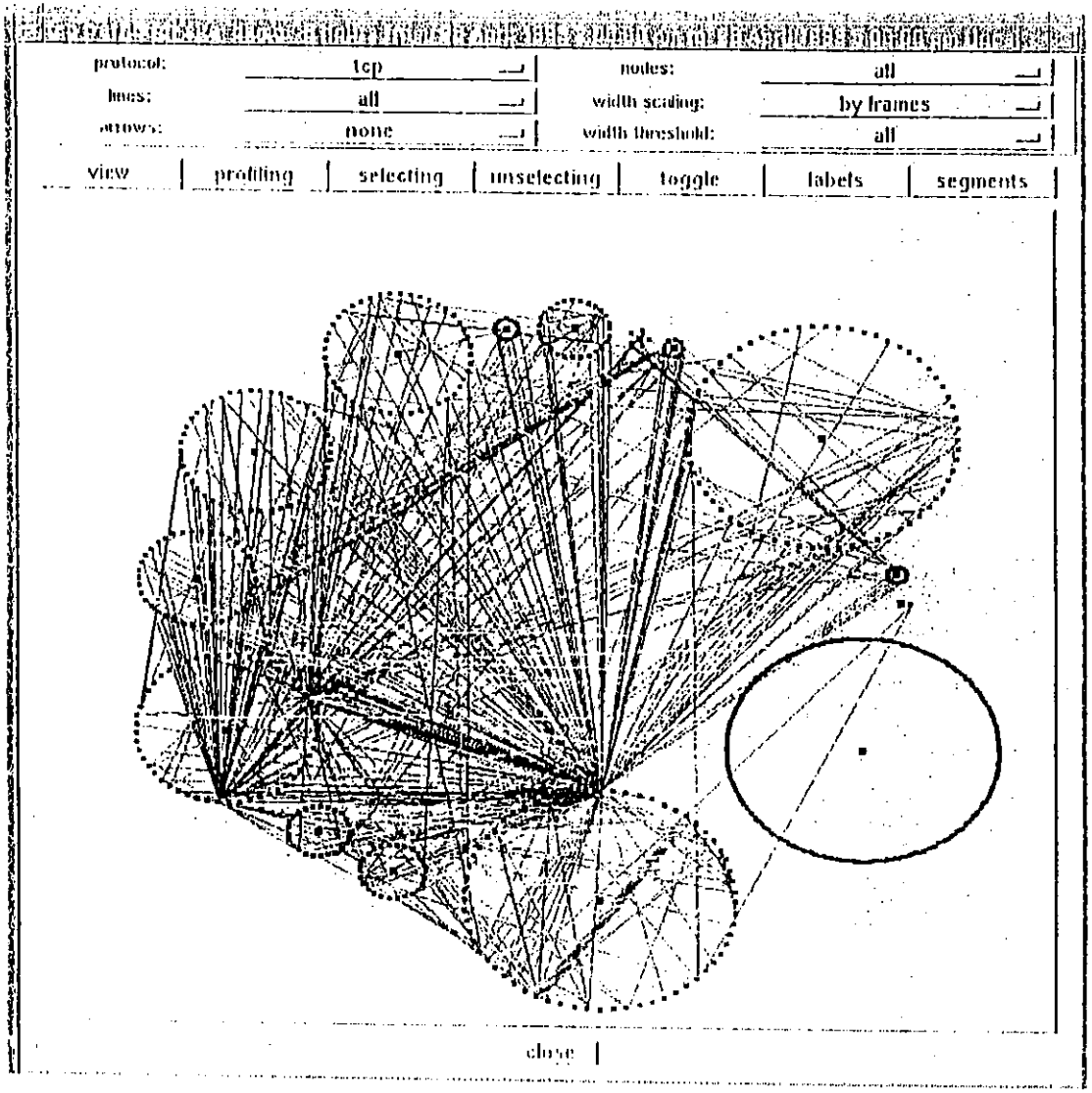
Eine optimale Plazierung ist erreicht, wenn
a) Komponentenpaare mit hohem Kommunikationsbedarf
im selben Netzsegment liegen, und
b) Komponentenpaare mit niedrigem Kommunikationsbedarf
in unterschiedlichen Netzsegmenten liegen

---

1) logische Nachbarschaft von Komponenten
2) physische Nachbarschaft von Komponenten

Plazierung

Beispiel

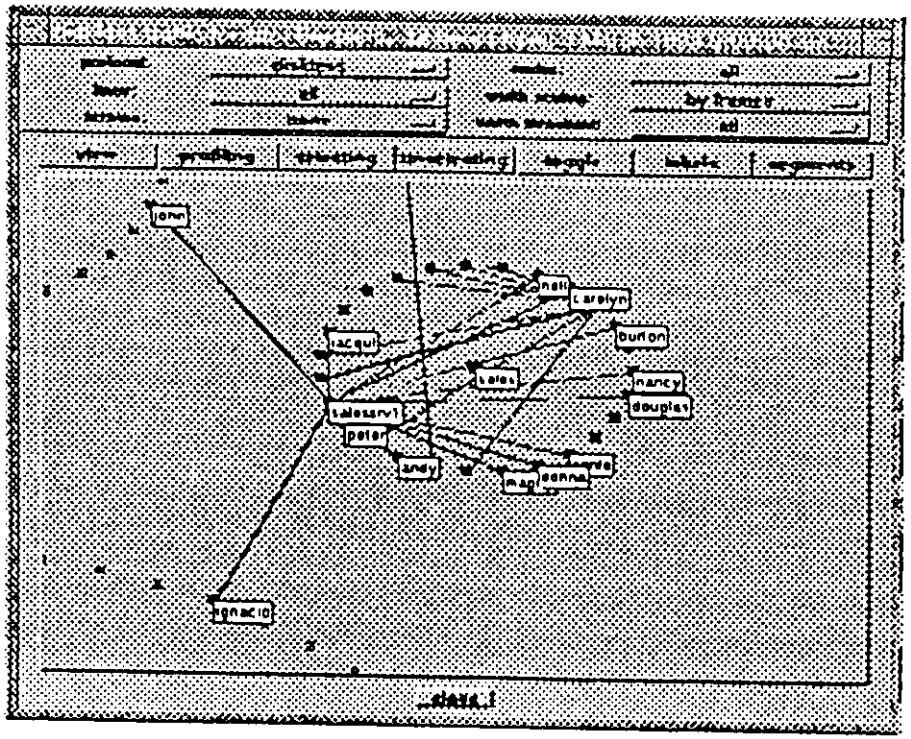


HP OpenView Traffic Expert

Plazierung

Beispiel

Traffic Pattern Viewer (zoomed)

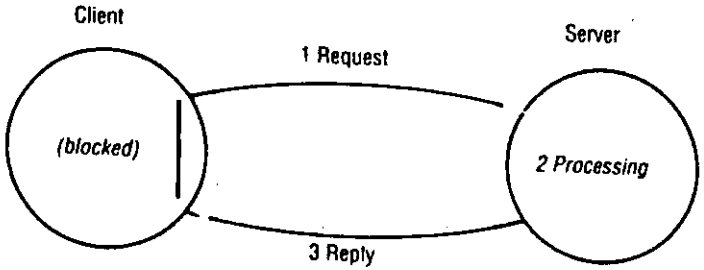


Kommunikation

in Verteilten Systemen

Client-Server-Modell

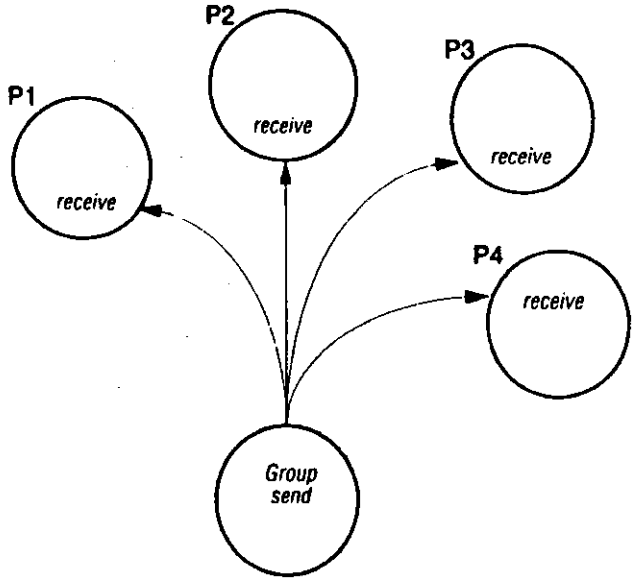
für Prozeß-Paare



typisch: Remote Procedure Call

Gruppen-Modell

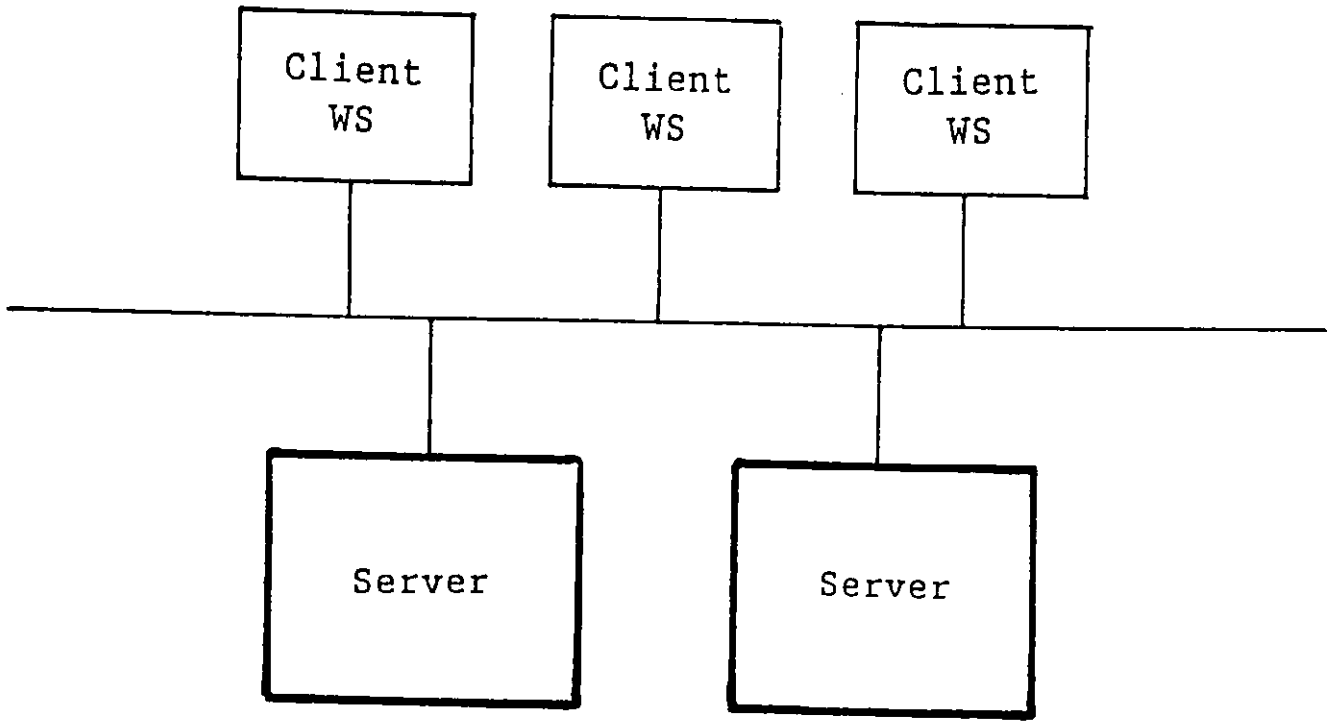
für kooperierende Prozeßgruppen



typisch: Multicast

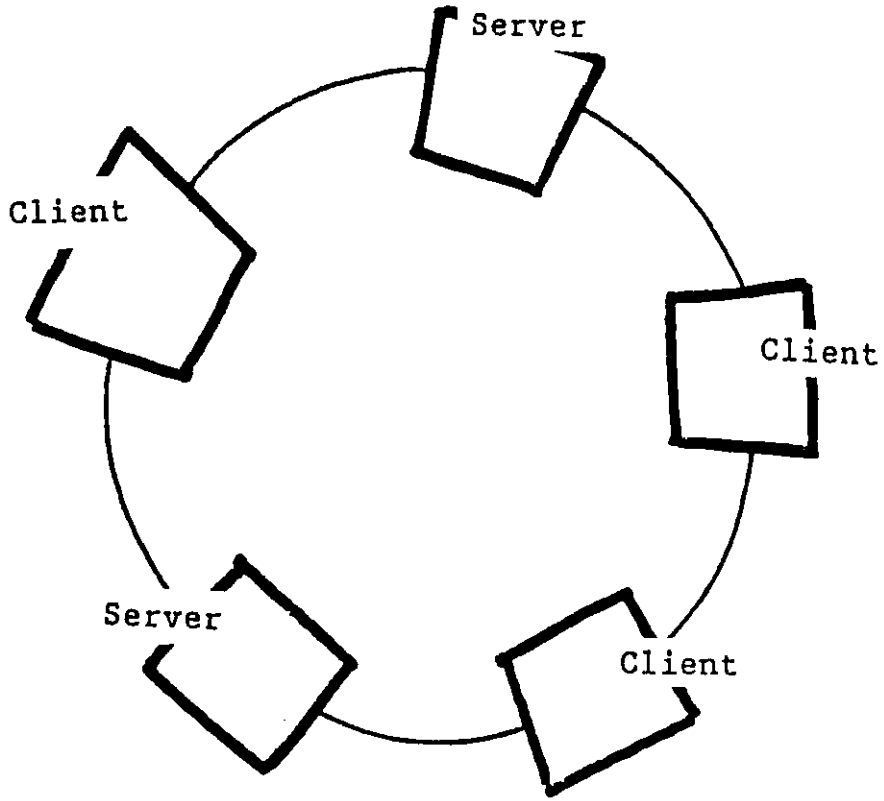
Client-Server-Szenario Nr. 1

"Hardware-Konfiguration"



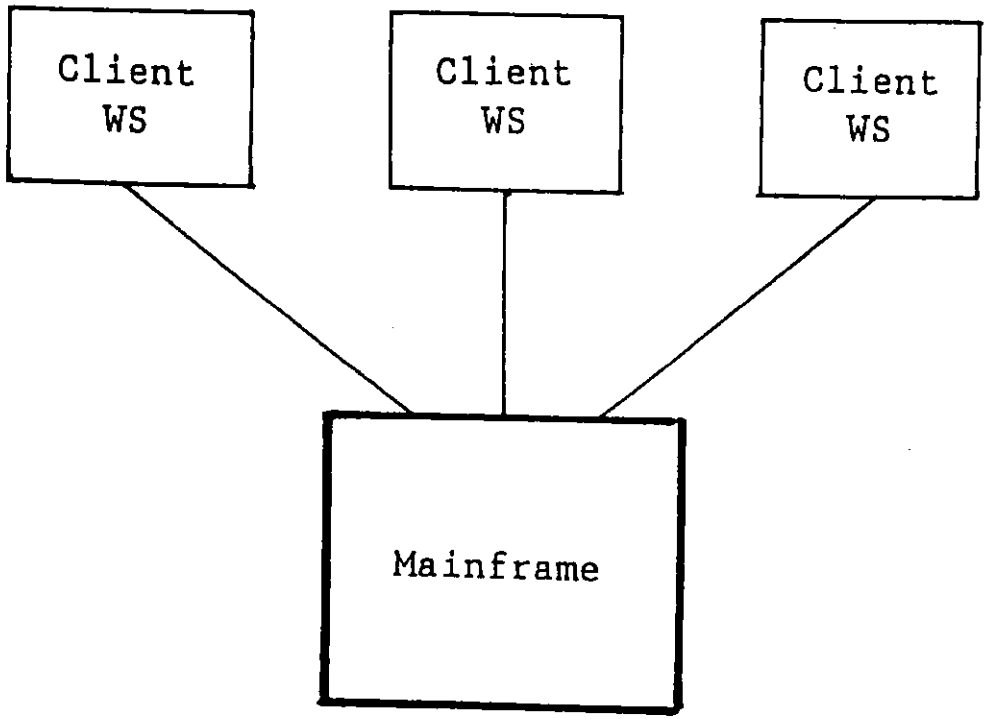
Client-Server-Szenario Nr. 2

"Integrierte Verteilte Verarbeitung"



Client-Server-Szenario Nr. 3

"Zentraler Host mit Workstations"



Server-Typen

Kontrolle gemeinsam genutzter Ressourcen:

- Disk-Server
- Print-Server
- File-Server
- Database-Server

Management-Services:

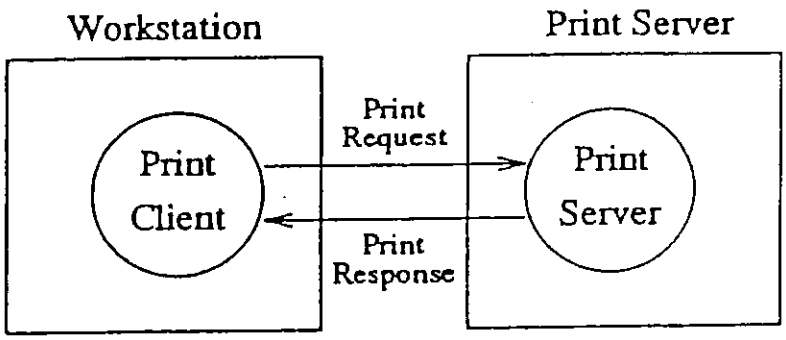
- Directory-Server
- Security-Server
- Resource-Allocation-Server

Benutzer-orientierte Dienste:

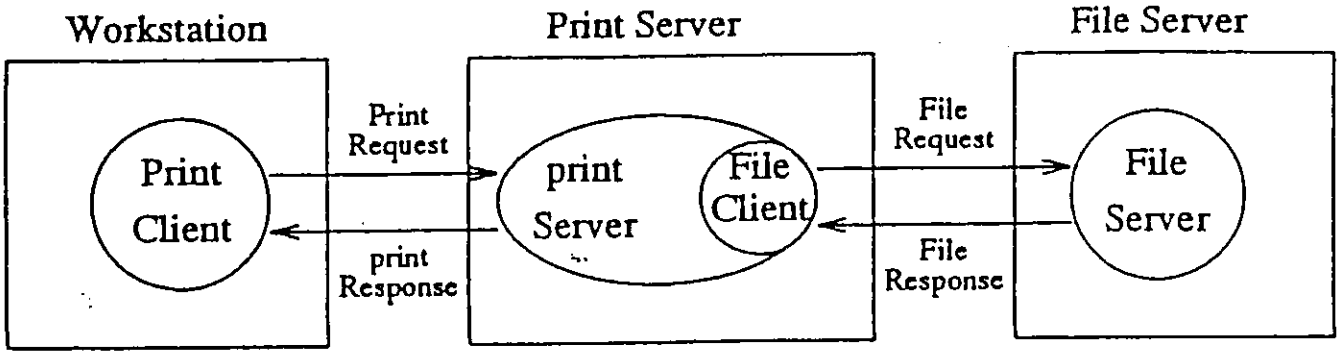
- "Gelbe-Seiten"-Server
- Mail-Server

Das Client-Server-Modell

Print-Client und Print-Server



Der Print-Server als Client des File-Server

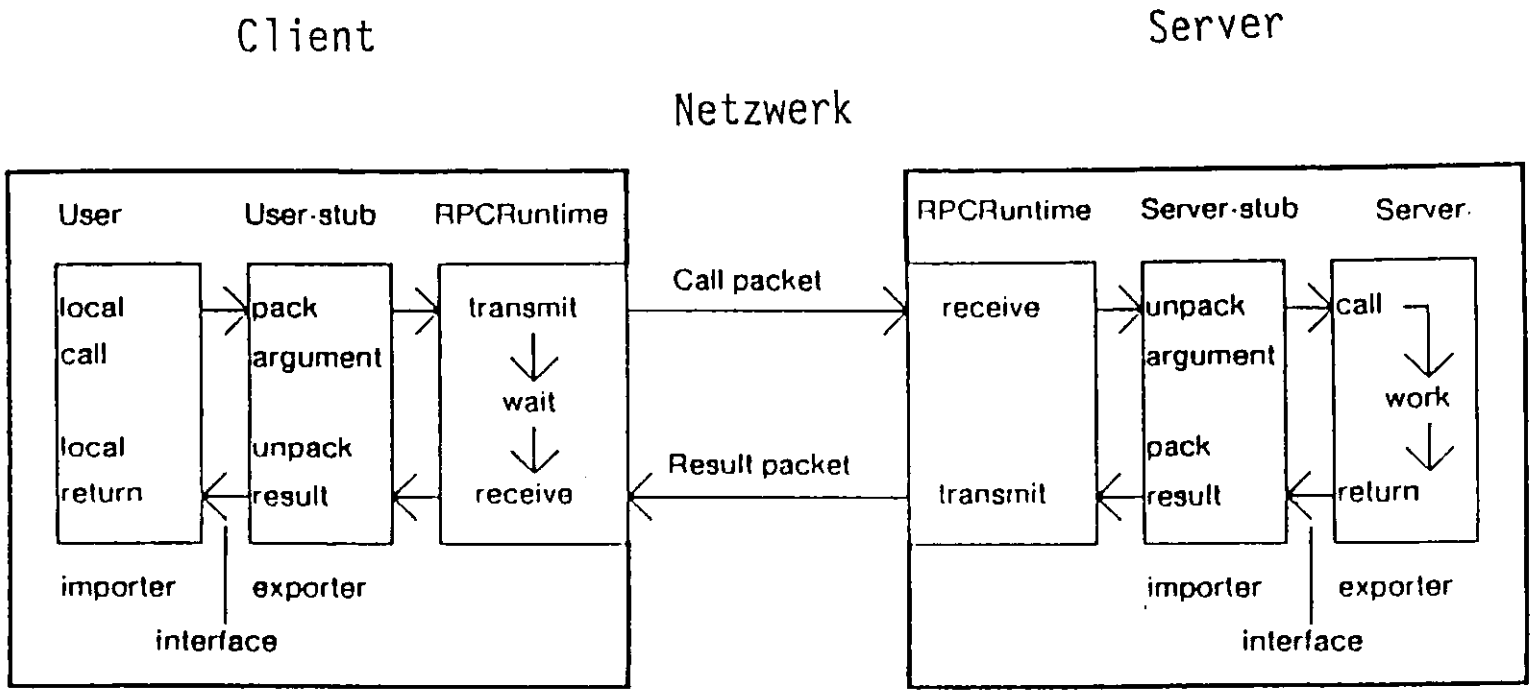


Client-Server-Dialog

- Lokale Kommunikation
- Geringes Datenvolumen (1-10 kB)
- Synchrone Abwicklung

Remote Procedure Call

Komponenten und Interaktionen



Remote Procedure Call

Fallunterscheidung

"Simple Call"

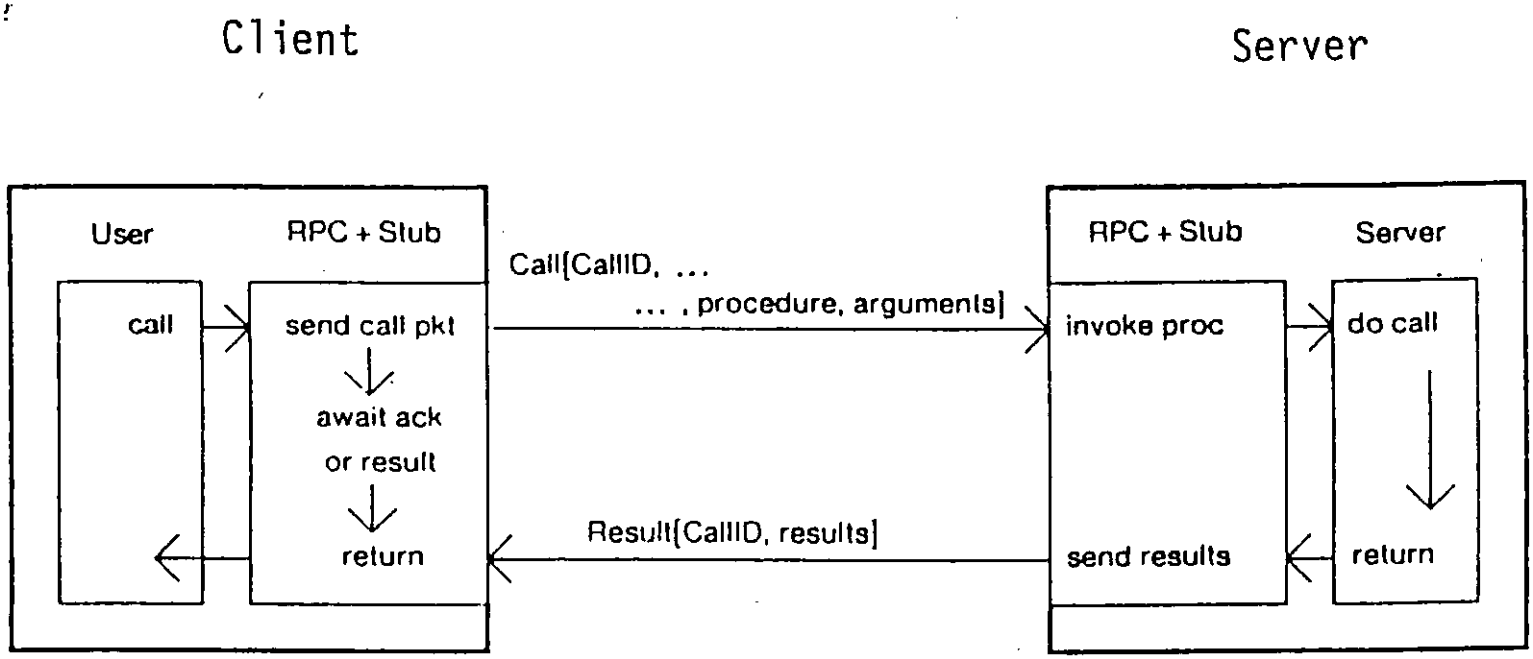
- Alle Parameter passen in ein einziges Datenpaket
- Die Calls folgen dicht aufeinander
- Die Abarbeitung der Calls dauert nur kurze Zeit

"Complicated Call"

- Die Parameter müssen in mehreren Datenpaketen übertragen werden
- Die Abstände zwischen den Calls sind groß
- Die Abarbeitung der Calls dauert lange

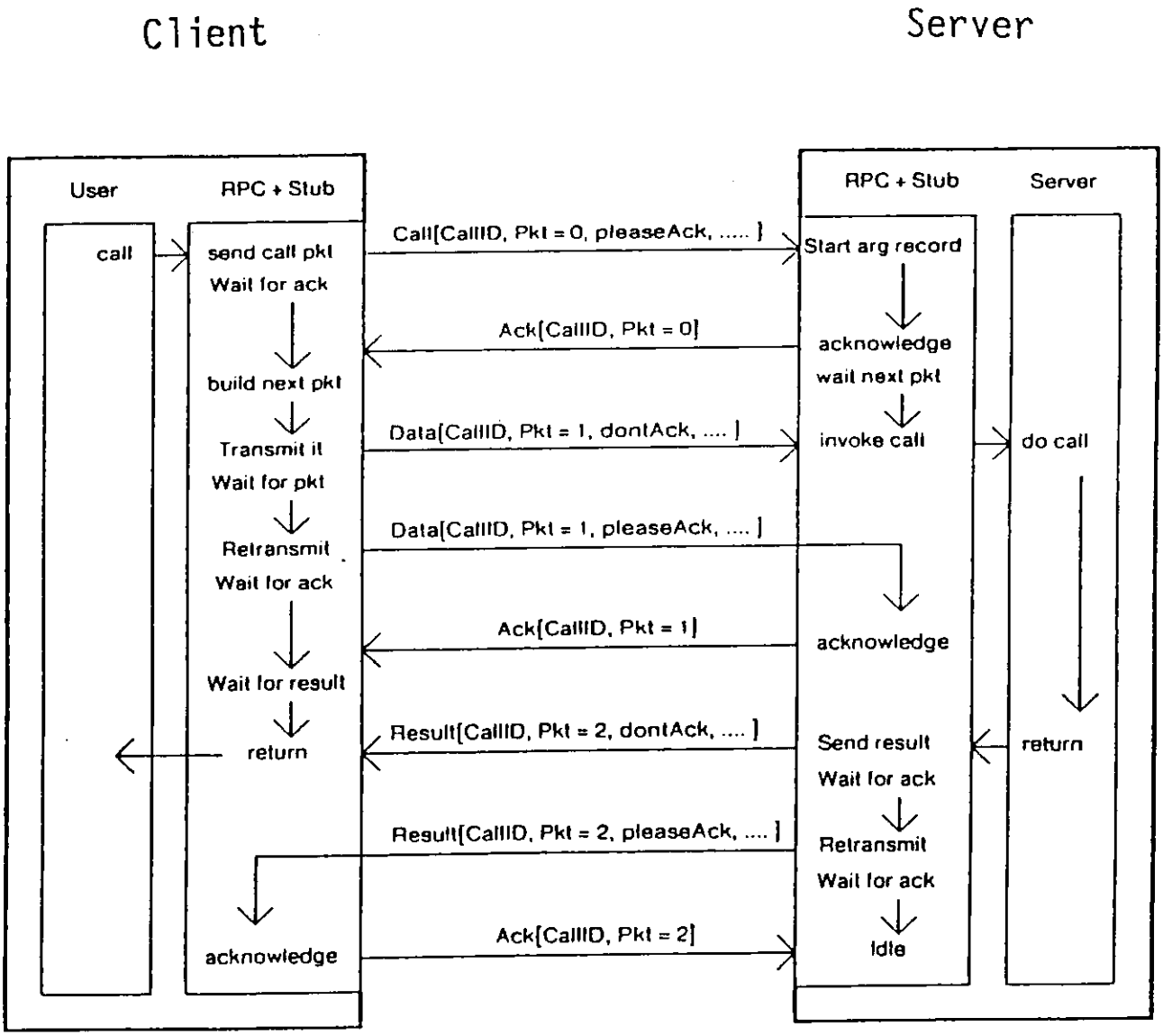
Remote Procedure Call

"Simple Call"



Remote Procedure Call

"Complicated Call"



RPC-Semantik

Ausnahme-Behandlung

Störfälle

- verlorengegangene Requests/Replies
- Server Crash

Störungserkennung

durch TimeOut

Störungsbehebung

durch Request-Wiederholung

Probleme

- unbekannte Anzahl von Server-Operationen
- nicht-idempotente Server-Operationen

RPC-Semantik

Sicherungsmaßnahmen

1) Request-Wiederholung

solange a) bis Reply eintrifft, oder
b) bis zweifelsfrei Server Crash festgestellt ist

2) Löschen von Request-Duplikaten

gegen wiederholte Ausführung einer Server-Operation
(realisiert über Message Identifier)

3) Reply-Wiederholung

"mit" und "ohne" Wiederholung der Server-Operation
"mit": bei Verlust einer Request-Nachricht
"ohne": bei Verlust einer Reply-Nachricht
(realisiert über "History")

RPC-Semantik

Sicherungsmaßnahmen

Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed;

Duplicate filtering: when retransmissions are used, whether to filter out duplicates at the server;

Retransmission of replies: whether to keep a history of reply messages to enable lost replies to be retransmitted without re-executing the server operations.

RPC-Aufrufsemantik

Delivery guarantees			*RPC call semantics*
Retry request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

RPC-Semantik

"Maybe"

- Keine Wiederholung des Request
- unsicher, ob die Operation überhaupt ausgeführt wurde

"At-least-once"

- Wiederholung des Request
- kein Löschen von Request-Duplikaten
- Mehrfachausführung der Operation möglich
- mindestens einmalige Ausführung, nach Reply

"At-most-once"

- Wiederholung des Request
- Löschen von Duplikaten
- keine Mehrfachausführung der Operation
- höchstens einmalige Ausführung

RPC-Orphans

"Verwaiste" Server nach Client-Crash

Durch "Orphans" verursachte Probleme:

- unnötiger Ressourcenverbrauch
- Blockieren von Files, etc.
- nach Reboot erhält Client veraltete Resultate

Vier Methoden, "Orphans" zu beenden:

1) Extermination

Nach Reboot, Auftrag an Servermaschinen, alle "Orphans" zu beenden (Voraussetzung: RPC-Log vor Ausführung)

2) Expiration

Server erhält Zeitquanten vom Client. Bei Ausbleiben eines Zeitquantums: Selbstbeendigung des Servers

3) Reincarnation

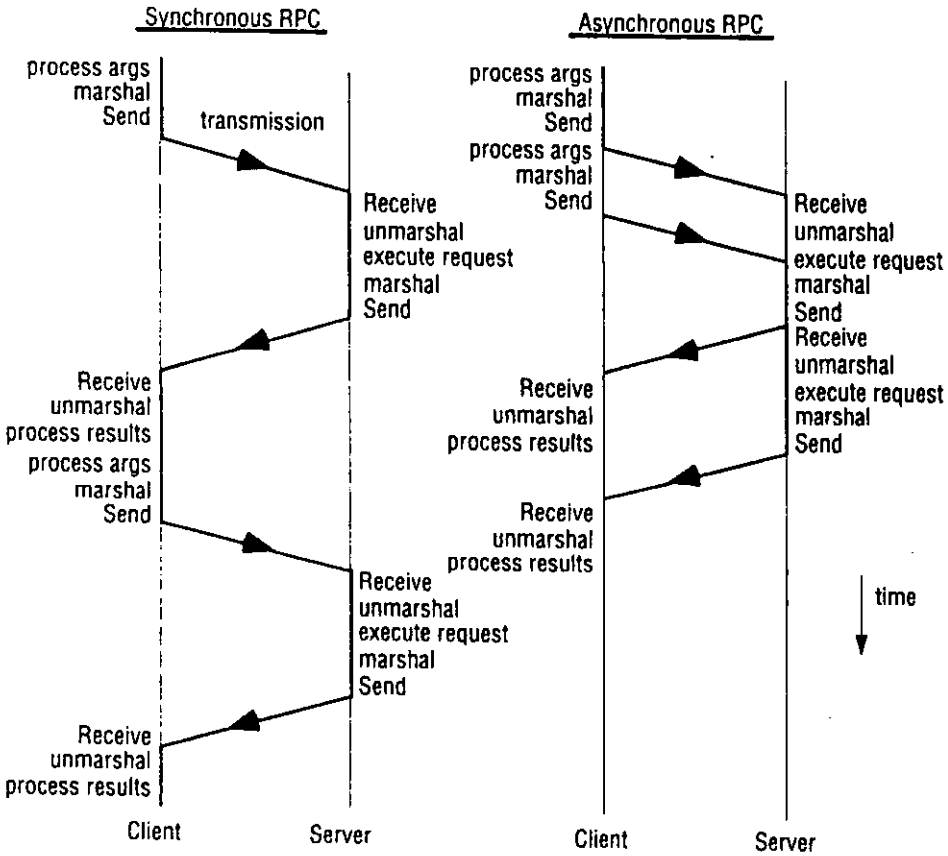
Nach Reboot, Ausrufen einer neuen "Zeitepoche", veraltete Replies sind an der "Epochennummer" zu erkennen

4) Gentle reincarnation

wie 3), zuvor jedoch, Versuch, den zum Server passenden Client zu lokalisieren

Asynchroner RPC

Nicht-blockierender RPC



Drei Varianten:

1) Ohne Reply
   Request → Freigabe des Client
2) Ohne Reply
   Requests sammeln → Sammel-request → Freigabe
3) Mit Reply
   Request → Freigabe ----→
   Reply empfangen

Beispiel ("X-11 Window System")

Ein Anwendungsprogramm (Client) sendet – ohne auf Replies zu warten – fortlaufend Bildinformationen (RPCs) an das X-Terminal (Server): Parallel-Lauf von Anwendung und Präsentation

RPC-Implementierung

Probleme

- heterogene Client/Server-Plattformen
- Interoperabilität
- Schnittstellen-Änderungen

Lösung

» Interface Definition Language «

+

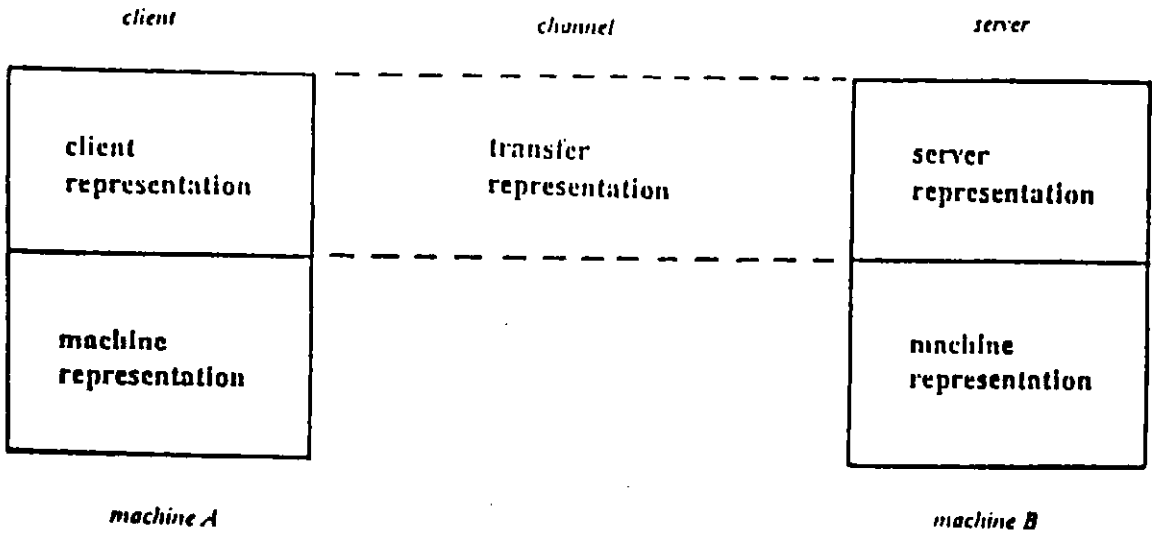
» IDL-Compiler «
(Stub*)-Generator)

---

*) Stub = En/Dekodier-Routinen zur Umwandlung von Anwendungsdatenstrukturen in Transfersyntax

Heterogenität

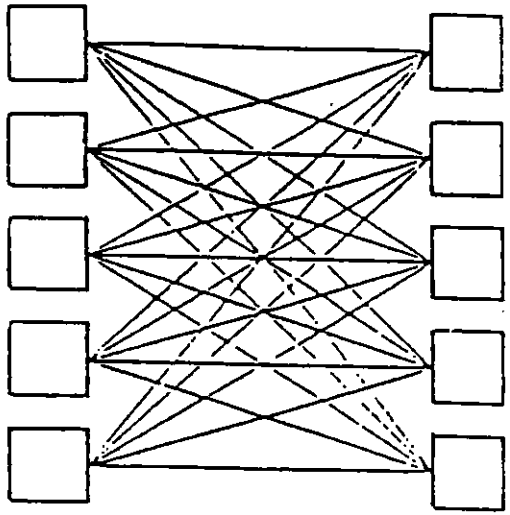
Syntax-Unterschiede



Heterogenität

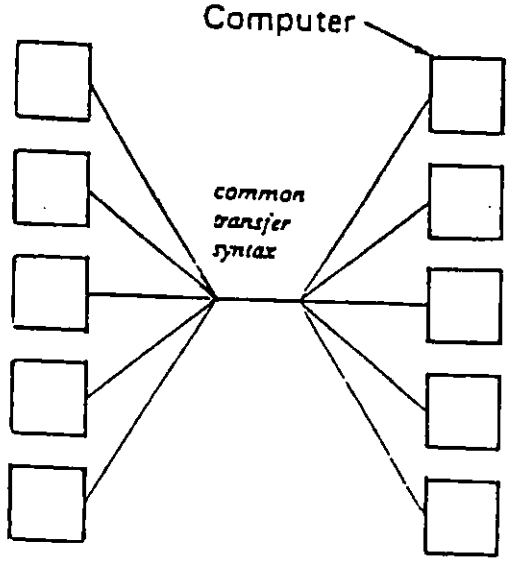
Syntax-Umwandlung

1. Methode



- 1 Konversion pro Datenaustausch
- aufwendig in heterogener Umgebung

2. Methode



- 2 Konversionen pro Datenaustausch
- geeignet für heterogene Umgebung (offenes System)

RPC-Spezifikation

Modularer Aufbau

abstract representation	IDL
internal representation	C
transfer representation	Bit/Byte Level

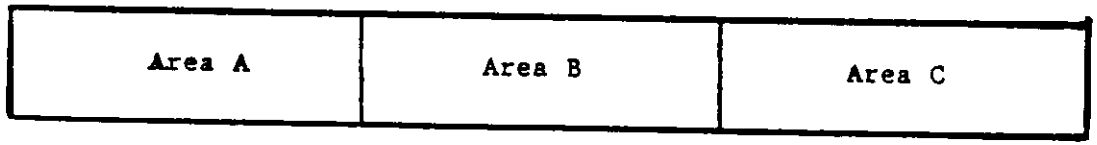
flexibel !

1) Alternative Darstellungen auf jeder Ebene möglich
2) Maschinen-unabhängig auf jeder Ebene

zu KFC-Spezifikation"

Nicht-nachahmenswertes Beispiel

Transfer Buffer:



Layout of Area A:

Byte	Description
Byte 1 - 2	offset of command sequer
Byte 3 - 4	length of command sequ
Byte 5 - 6	offset of operand tab
Byte 7 - 8	length of operand ta
Byte 9 - 10	offset of value ar
Byte 11 - 12	length of value a
Byte 13 - 14	offset of record
Byte 15 - 16	length of reco
Byte 17 - 20	offset of rec
Byte 21 - 22	free
Byte 23	flag to ind - x'01' - x'02' - x'0 - x' - x -

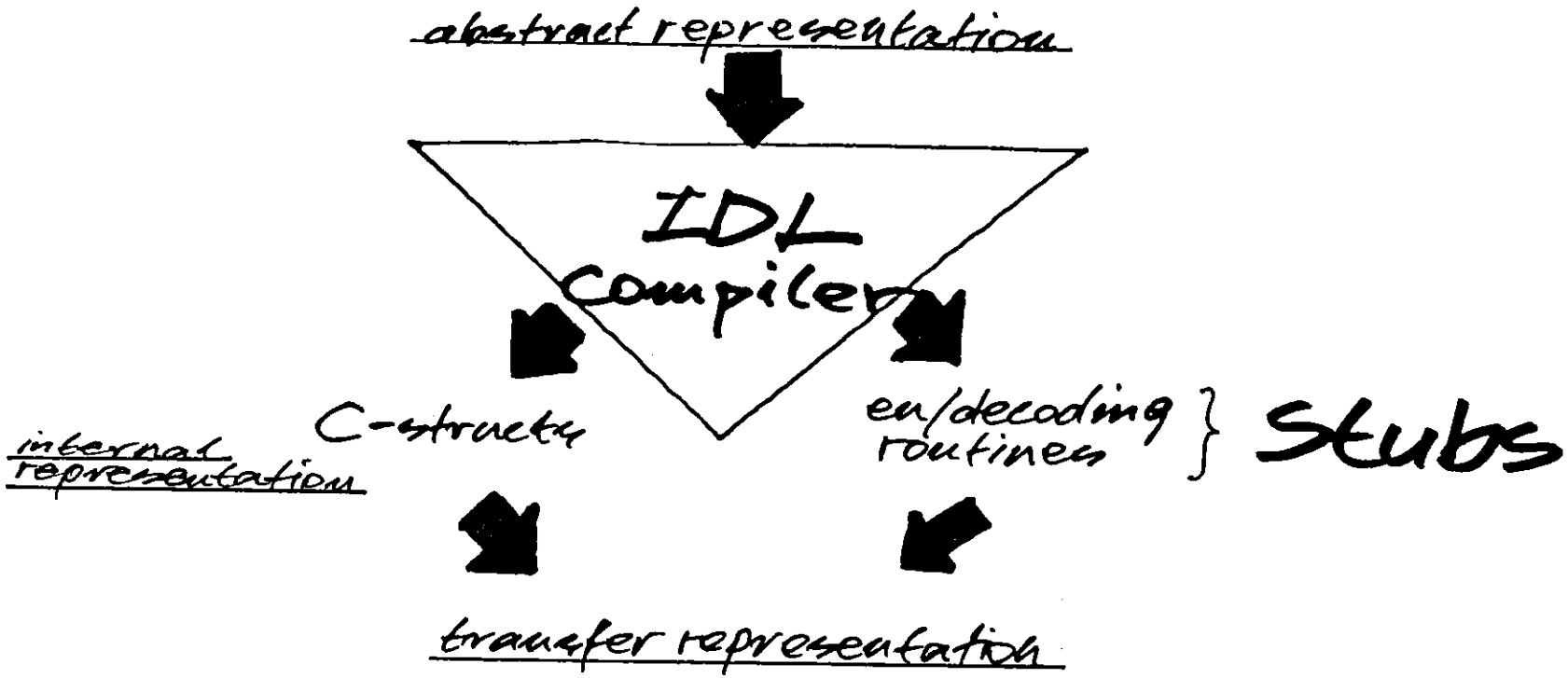
unflexibel !

1) Enthält gleichzeitig
- Protokoll-Spezifikation
- Anwendungsdatenstrukturen
- Transfersyntax

2) Maschinen-abhängig

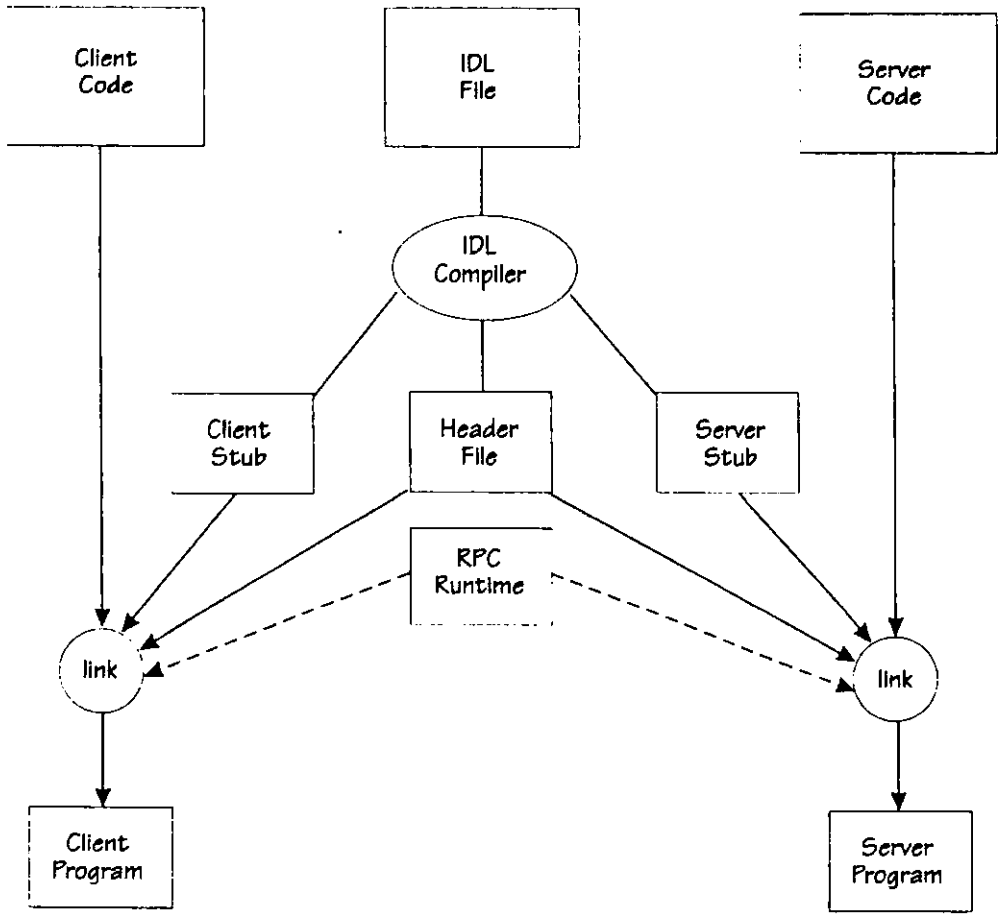
Stub Generator

Computer Aided Engineering



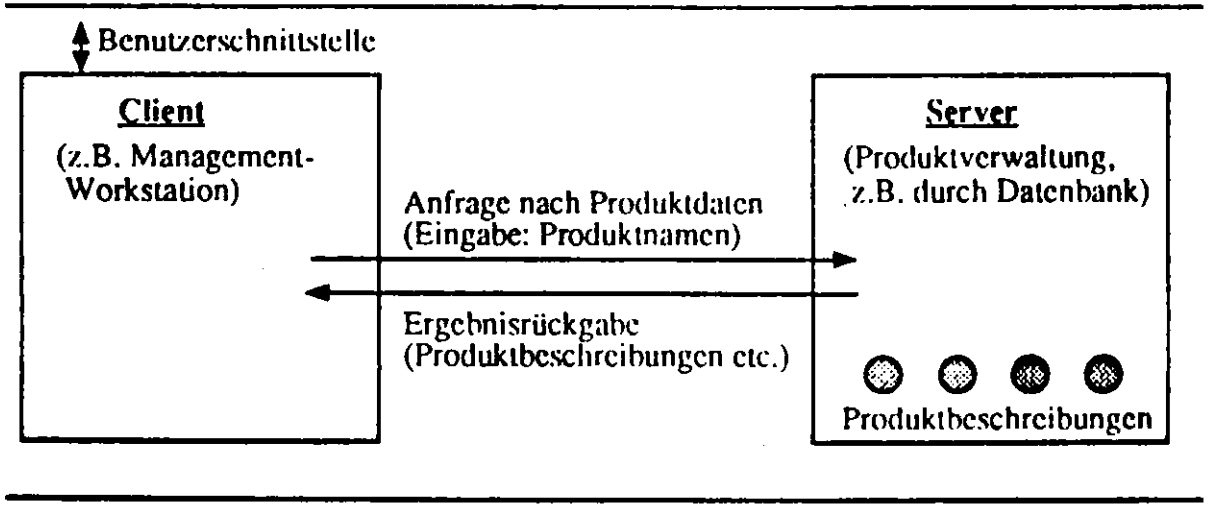
RPC

Implementierungs-Prozeß



RPC

Programmierbeispiel



Das Beispiel (s. Abb. 3-3) realisiert eine entfernte Anfrage nach Produktdaten, die durch einen Server verwaltet werden. Der Server erhält eine Liste von Produktnamen und liefert die entsprechenden Produktbeschreibungen dazu zurück. Außerdem berechnet er die Summe der mit den Produkten gespeicherten Entwicklungskosten und liefert diese ebenfalls als Rückgabeparameter an den Client.

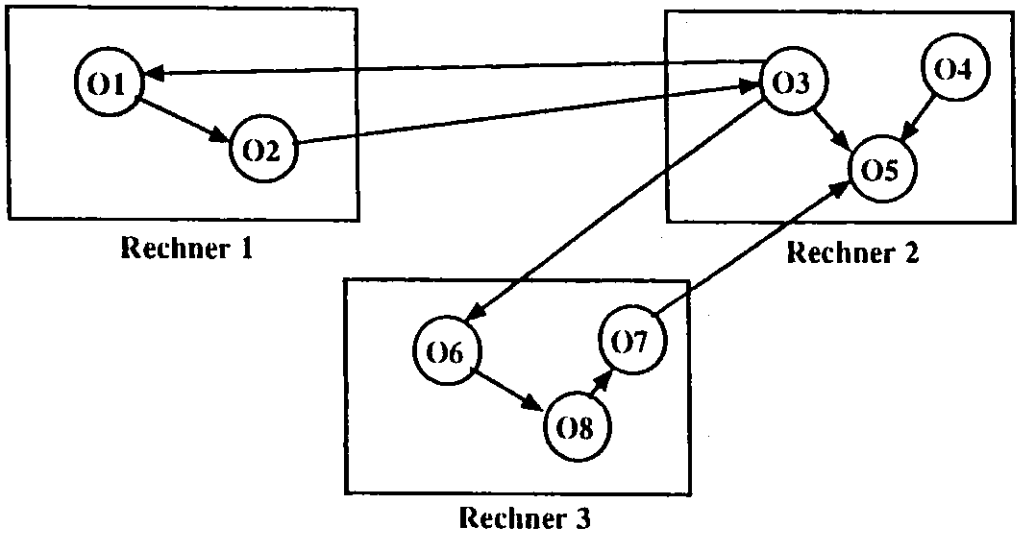
RPC-Schnittstelle

IDL - Definition

```
[
uuid(765c3b10-100a-135d-1568-040034c67831),
version(1.0),
pointer_default(ptr)
]
interface ProductData                          // Schnittstelle für Produktdaten
  {
  import "globaldef.idl";                      // Import allgemeiner IDL-Definitionen

  const long maxNoProducts = 10;               // Maximale Produktanzahl
  typedef [string] char *String;               // Datentyp für Character-Strings

  typedef struct {
     String productName;                       // Produktname
     String productAnnotation;                 // Textuelle Beschreibung
     String dateOfIssue;                       // Erscheinungsdatum
     long developmentCosts;                    // Gesamte Entwicklungskosten
  } ProductDescription;                        // Produktbeschreibung

  long productQuery (                          // Anfrage n. Produkten u. Entw.kosten
     [in] String productName[maxNoProducts],   // -> Produktnamen
     [out] ProductDescription *pd[maxNoProducts], // <- Produktbeschreibungen
     [out] long *totalDevelopmentCosts );      // <- Summe der Entwicklungskosten
}
```

Verteilte Objekte

Struktur einer verteilten objektorientierten Anwendung

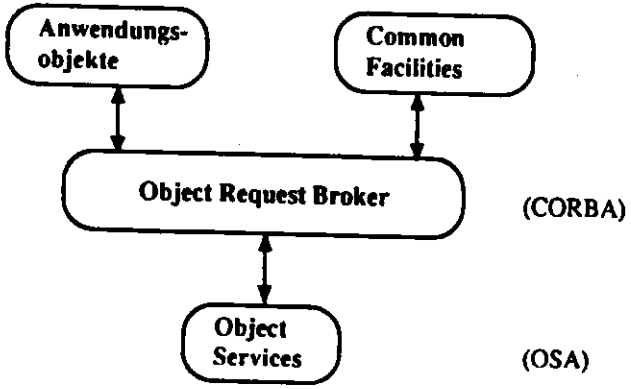


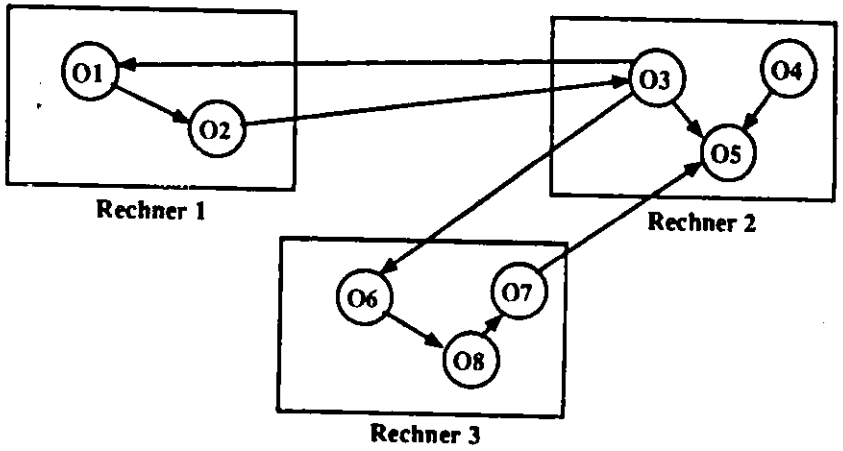
- *Objektverteilung:* Es ist möglich, Objekte relativ beliebig auf verschiedene Rechner zu verteilen, indem entfernte Objekterzeugungsoperationen aufgerufen werden. Alle Objekte sind global eindeutig gekennzeichnet und können sich daher auch rechnerübergreifend eindeutig referenzieren.
- *Lokationsunabhängige Aufrufe:* Unabhängig von der momentanen Lokation eines Objektes kann dieses von anderen Objekten aufgerufen werden - egal, ob sich das aufrufende Objekt auf dem gleichen oder auf einem anderen Rechner befindet.
- *Objektmigration:* Objekte können auch dynamisch zur Laufzeit zwischen Rechnern migriert, also verlagert werden. Dazu bieten sie eine spezielle Migrationsoperation an, die im Rahmen der Anwendung explizit aufgerufen werden kann. Dies ermöglicht z.B. die Zusammenführung kommunizierender Objekte oder auch die Lastverteilung im System.

CORBA

Common Object Request Broker Architecture

Object Management Architecture





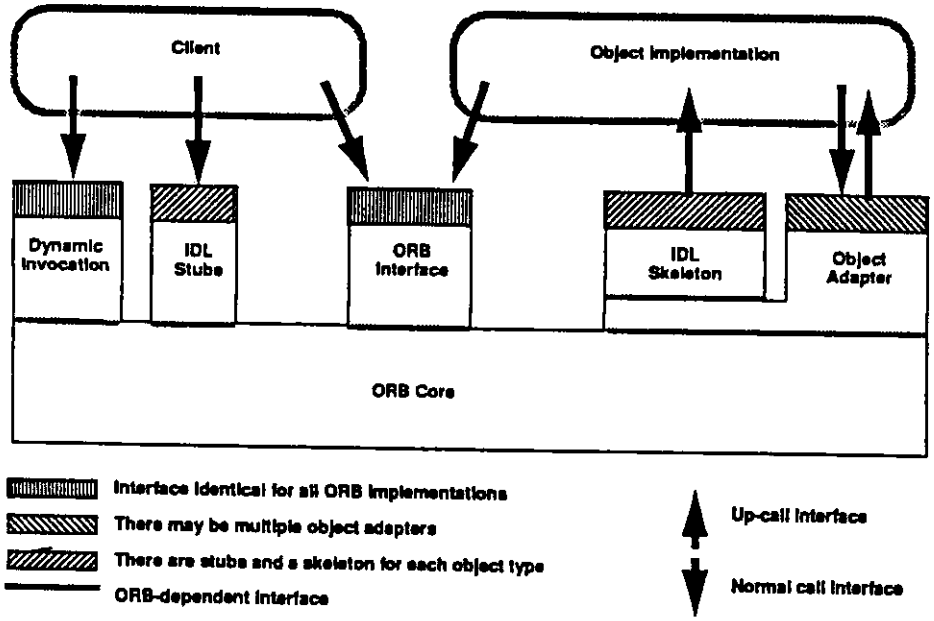
Struktur einer verteilten objektorientierten Anwendung

Object Services

Dienst	Funktionalität
Archive	Schnittstelle zwischen aktivem Speicher und Backup
Backup/Restore	Backup und Recovery von Datenobjekten
Change Management	Versions- und Konfigurationsverwaltung für Objekte
Concurrency Control	Steuerung/Kontrolle nebenläufiger Objektzugriffe
Data Interchange	Verteilter Objektaustausch (z.B. auf Basis von ASN.1)
Event Notification	Verwaltung/Signalisierung dynamischer Ereignisse
Externalization	Transformation von Objekten in flaches Speicherformat
Implementation Repository	Verwaltung/Speicherung von Objektimplementierungen
Installation and Activation	Verteilung, Aktivierung und Migration von Objekten
Interface Repository	Verwaltung/Speicherung von Objektschnittstellen
Licensing	Lizenzverwaltung
Lifecycle	Erzeugen, Löschen und Kopieren von Objekten
Naming	Namensverwaltung für Objekte
Operational Control	Monitoring des Objektverhaltens
Persistence	Persistente Objektverwaltung auf Hintergrundspeicher
Query	Anfragesprachen für den Objektzugriff
Relationships	Beziehungen zwischen zwei und mehr Objekten
Replication	Verteilte konsistente Objektreplikation
Security	Zugriffskontrolle für Objekte
Startup Services	Initialisierung und Terminierung von Objekten
Threads	Nebenläufige Prozesse
Time	Verteilte Zeitsynchronisation
Trading	Abbildung von Dienstanforderungen auf Dienste/Server
Transactions	Verteilte Transaktionsverwaltung

CORBA

Schnittstellen



Dynamic Invocation

Für alle Services einheitlich, separate (und einheitlich kodierte) Übertragung der Elementarteile eines Requests, zB. Objektspezifikation, Operation, Parameterstyp und -Wert.

IDL-Stubs und -Skeletons

statisch definierte Service-Schnittstellen

ORB-Interface

Allgemeine Services, core zB. Input/Output

Object Adapter

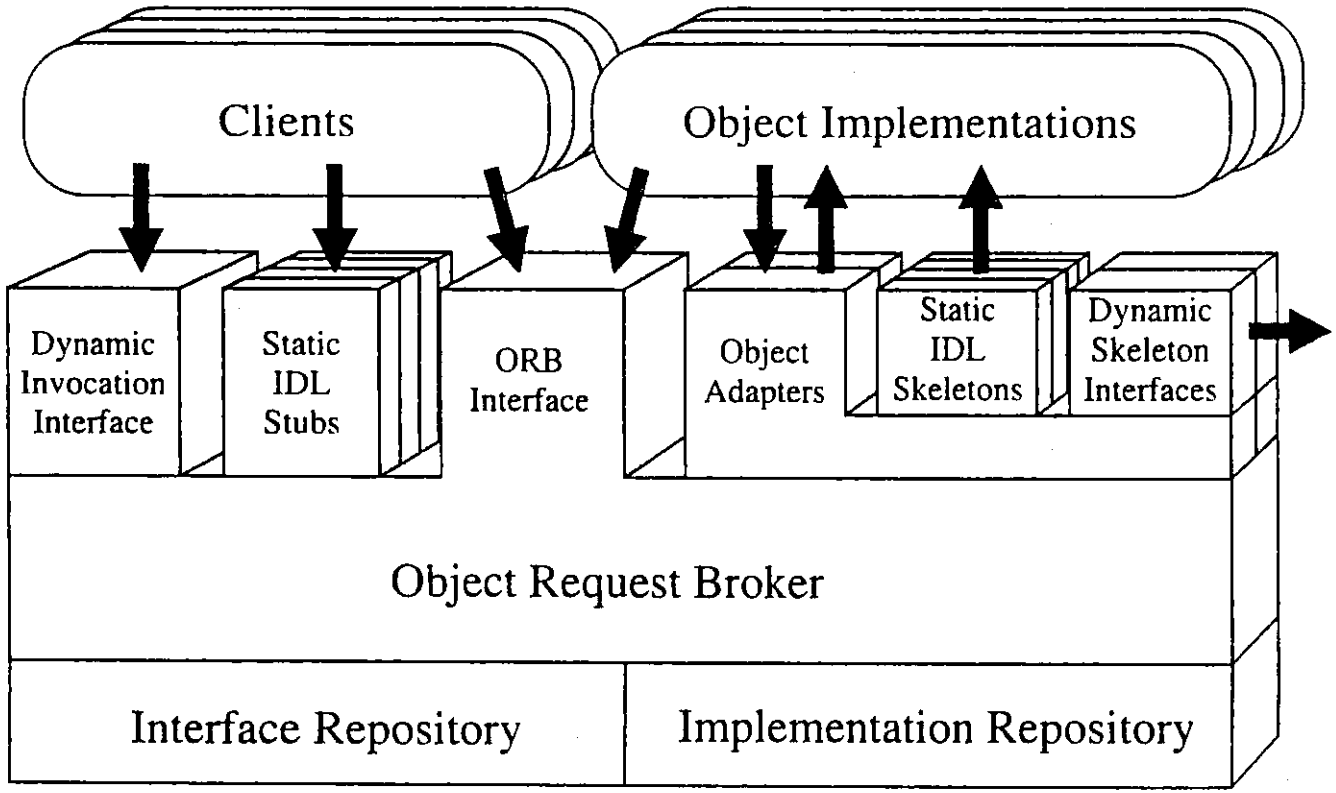
Anpassung der ORB-Schnittstellen an unterschiedliche Implementierungsstile (separate Server, Bibliotheken, Methoden des Programms, Verkapselte Anwendungen, O-O DB, ...).

Dienste:

- Methodenaufruf
- Sicherheit der Interaktionen
- Aktivierung/Deaktivierung von Objektimplementierungen
- Registrierung von Implementierungen

CORBA

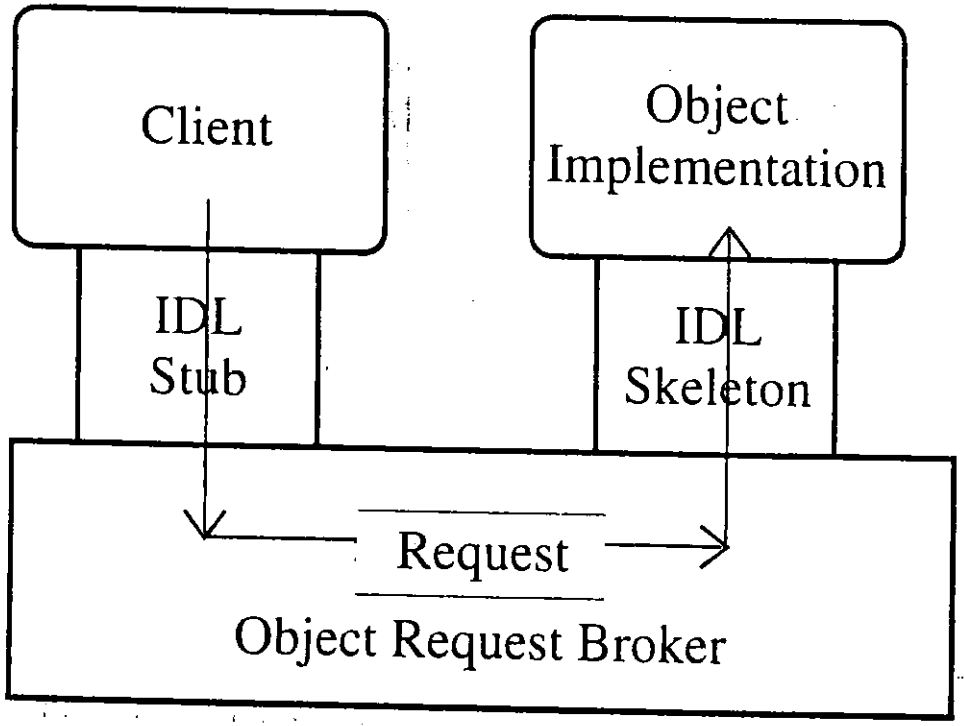
Gesamtarchitektur



Structure of the Object Request Broker. Interfaces between ORB components and its clients and object implementations (shown by arrows) are expressed in OMG IDL and standardized by the OMG. Interfaces between ORB components (where component boxes abut in the figure) are proprietary.

CORBA

A request passing from client to object implementation.



OMG IDL BY EXAMPLE

```
//POS Object IDL example
module POS {
        typedef string Barcode;

        Interface InputMedia {
            typedef string OperatorCmd;
            void  barcode_Input(in Barcode item);
            void  keypad_Input(in OperatorCmd cmd);
        };

        interface OutputMedia {
            boolean output_text(in string string_to_print );
        };

        interface POSTerminal {
            void  end_of_sale();
            void  print_POS_sales_summary();
    };
};
```

IDL Modules, Types, and Scoping

Interface

Operations

Exceptions

```
exception input_out_of_range { long dummy };
void operation1(in long arg1) raises (input_out_of_range);
```

Invocation Semantics and the Oneway Declaration

```
oneway void SendMyMessage (in string MyMessage);
```

Context Objects

```
context (context1, context2, ...)
```

Attribute Variables

```
Interface MyInterface {
    attribute float radius;
};
```

Inheritance

```
interface example1 {
    long operation1 (in long arg1);
};

interface example2:example1 {
    void operation2 (in long arg2, out long arg3);
};
```

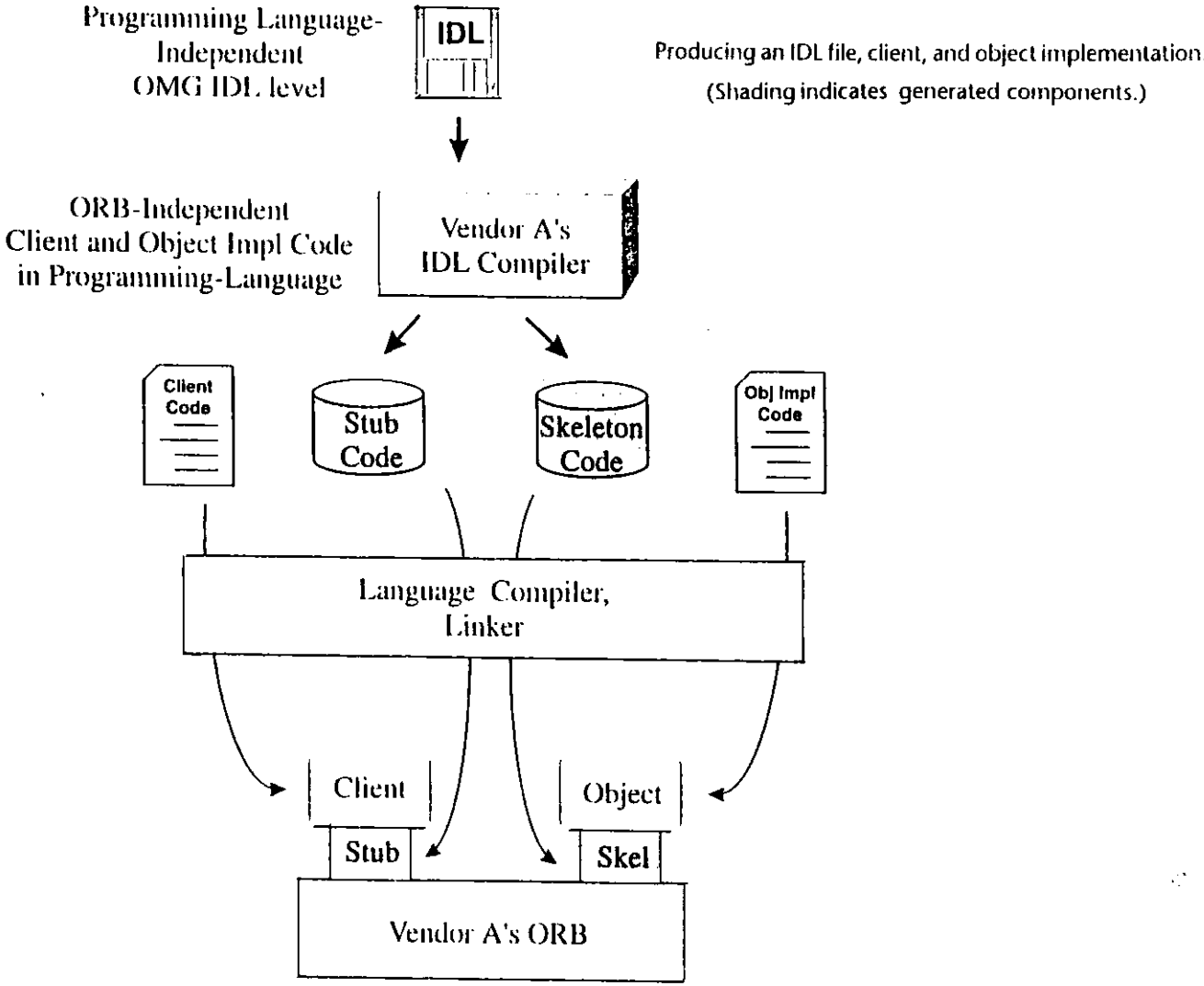
The Object Interface

inherited by every object

```
Interface Object {                                  // PIDL
    ImplementationDef get_implementation ();
    InterfaceDef get_interface ();
    Object duplicate ();
    void release ();

    Status create_request (...
    );
```

IDL

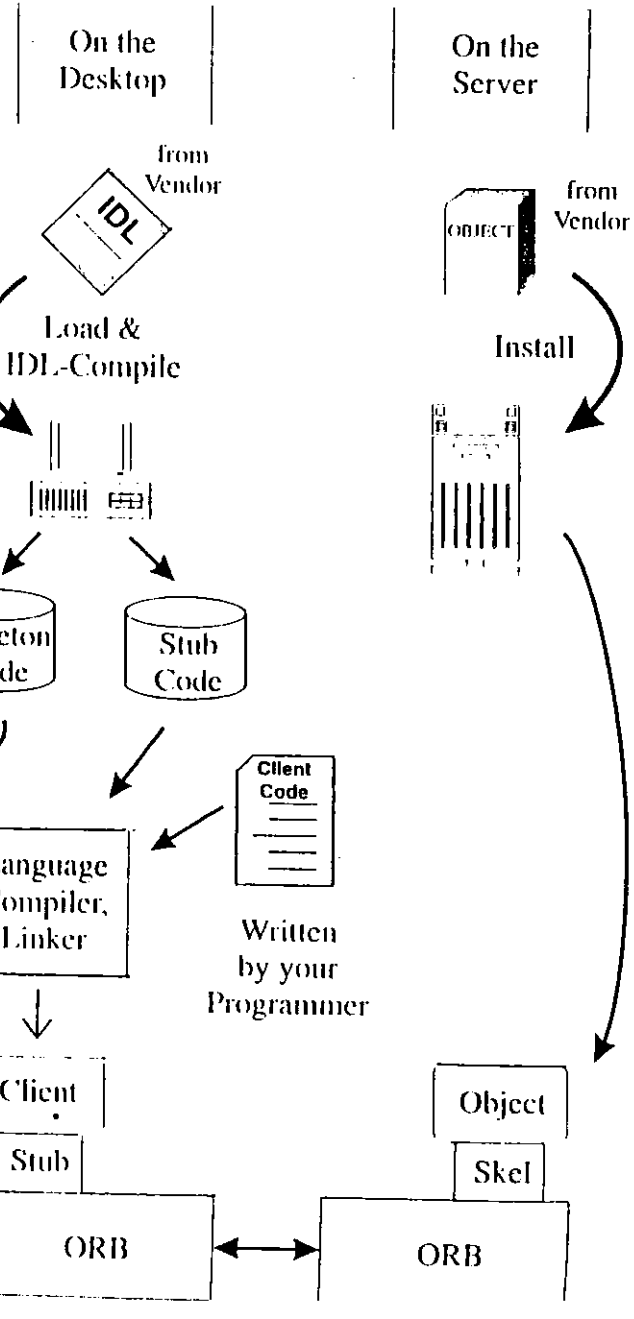
Interface Definition Language



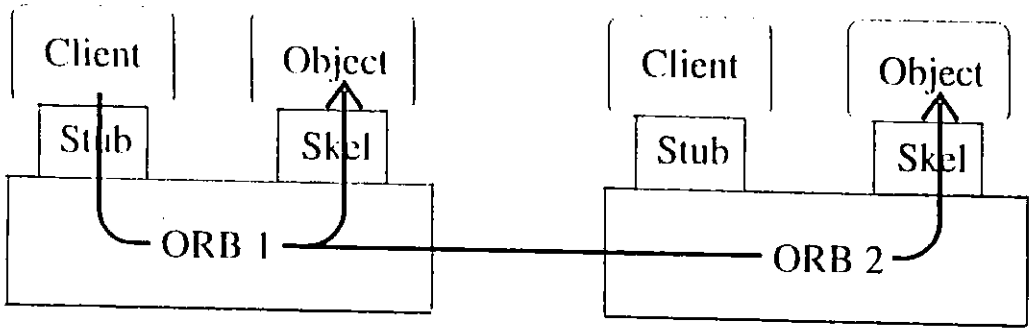
CORBA

Interoperability

Integrating a purchased object into your software environment.



Interoperability uses ORB-to-ORB communication.

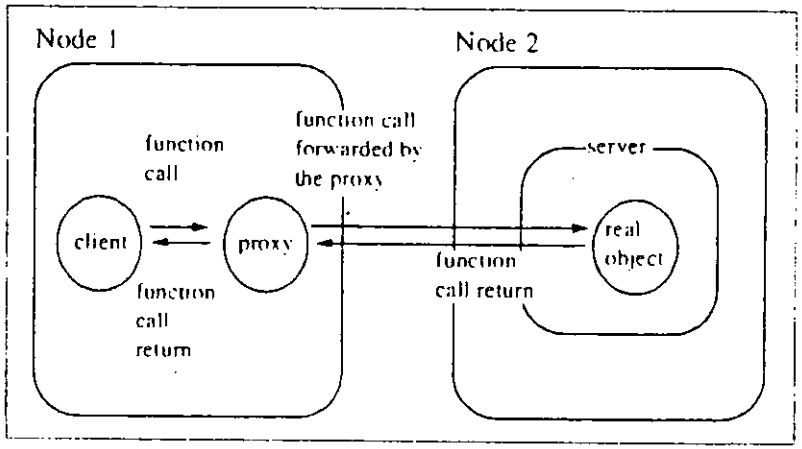


Proxy Object

ORBIX / Iona Remote Invocation

If the real target object is in fact remote, then the object reference for it automatically denotes a *proxy object* which implements the support necessary to send requests to the remote object: all requests made by the client will be forwarded automatically by the proxy to the remote object.

An operation call on a proxy

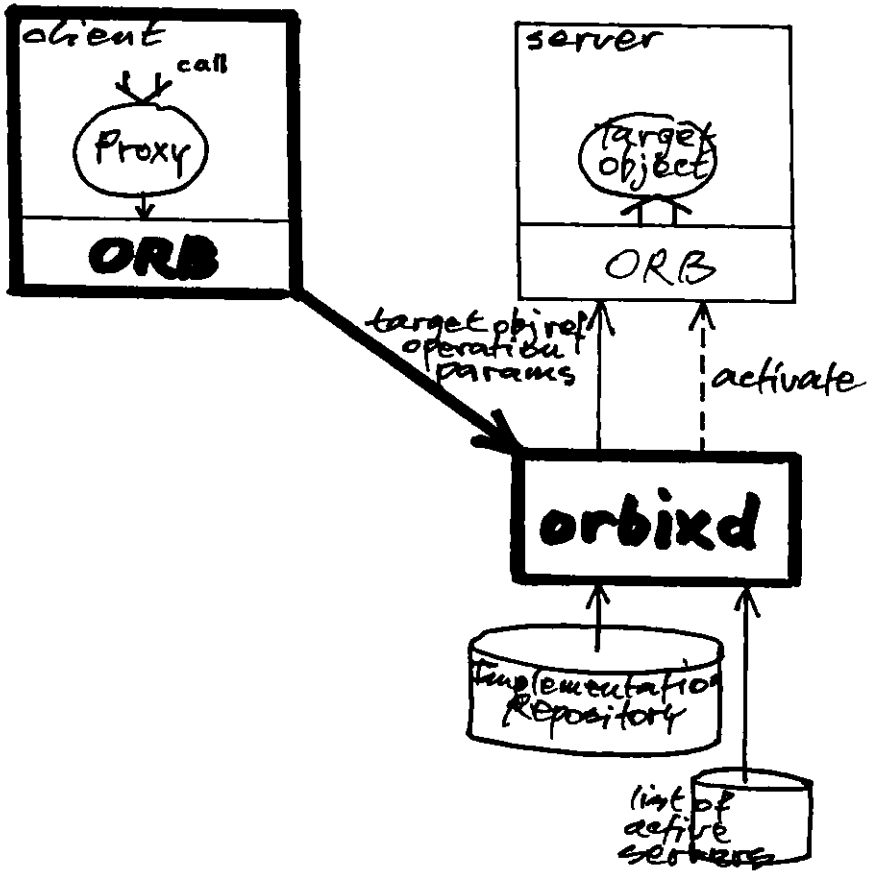


- proxy methods: IDL stubs
- proxy code generated by IDL compiler
- transparent to user
- "smart proxy"
  - derived from proxy
  - additional functionality for
    - caching
    - fault tolerance
    - load balancing

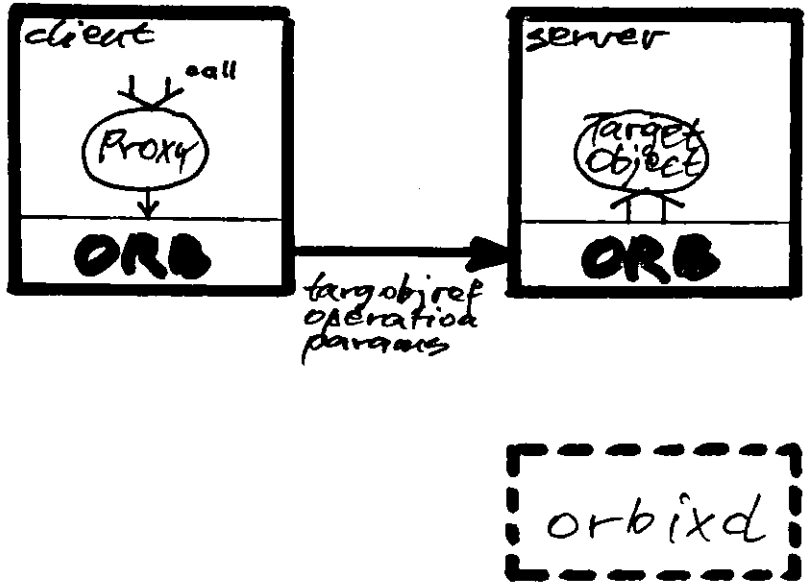
CORBIX

Operation Calls

First Call

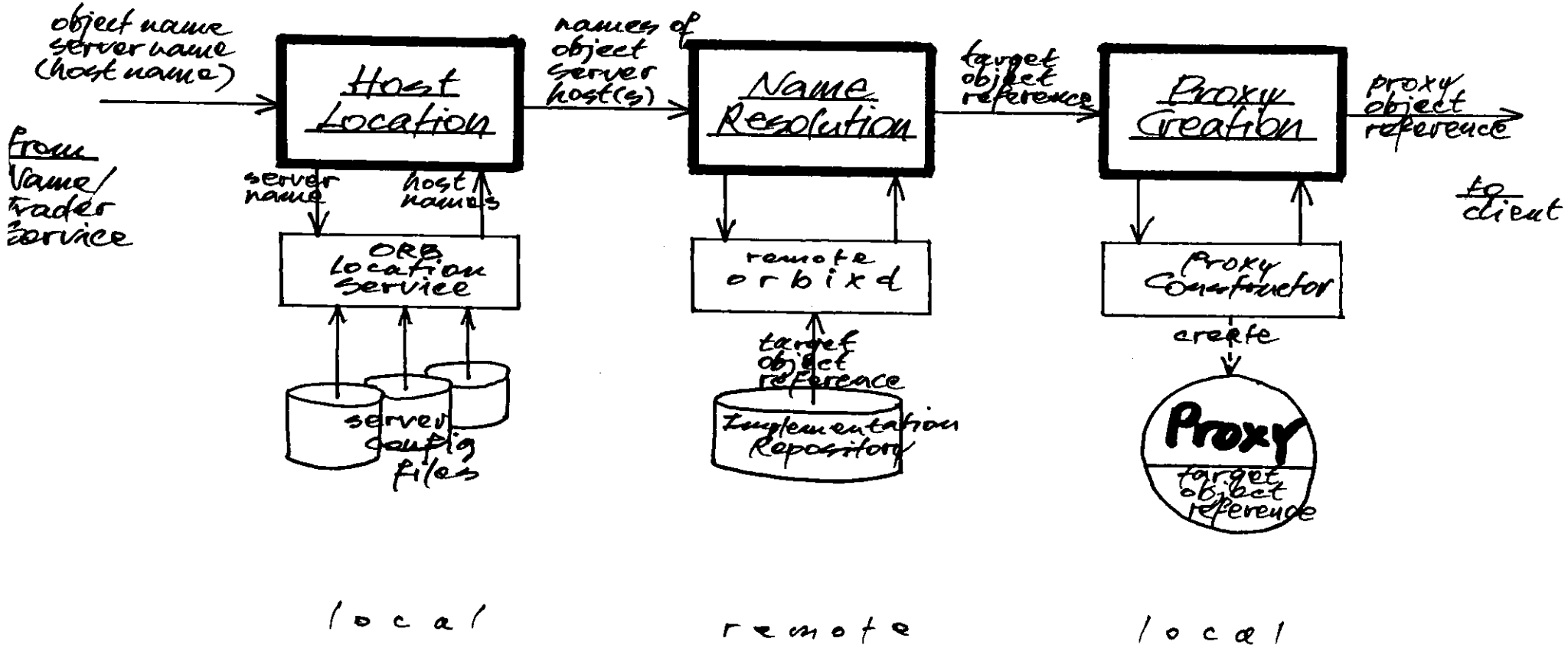


Subsequent Calls



ORBIX

Binding



Dynamic Invocation

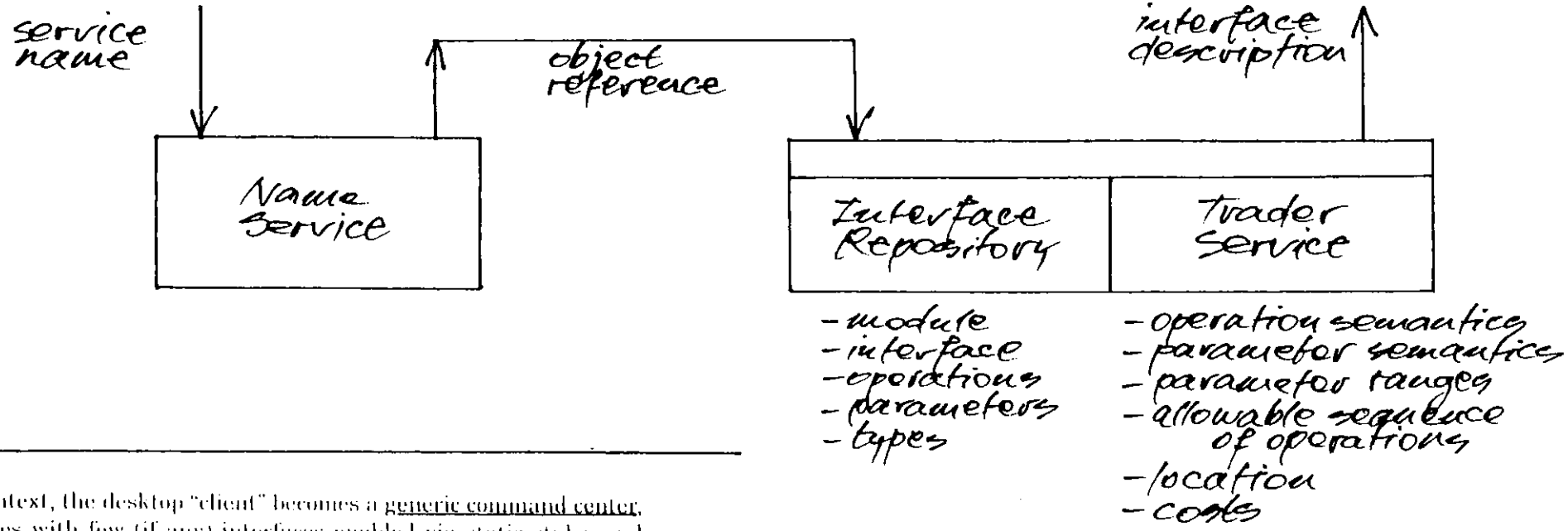
Objekt-Auswahl zur Laufzeit *)

There are four steps to a dynamic invocation:

1. Identify the object you want to invoke.
2. Retrieve its interface.
3. Construct the invocation.
4. Invoke the request, and receive the results.

1. Object Identification

2. Interface Retrieval



*) In this context, the desktop "client" becomes a generic command center, which activates with few (if any) interfaces enabled via static stubs, and proceeds to configure itself to invoke all of the actions its user wants via the DII.

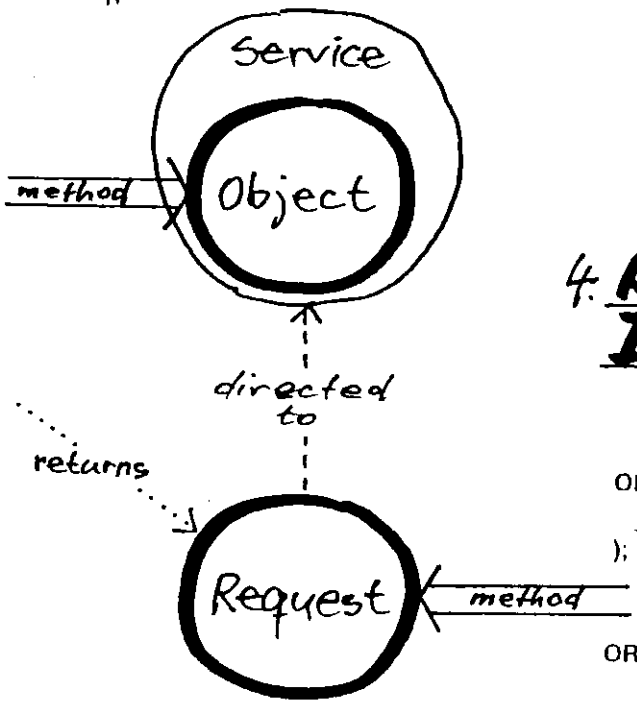
Dynamic Invocation

```
interface Object {
    ImplementationDef get_implementation ();
    InterfaceDef get_interface ();
    Object duplicate ();
    void release ();

    Status create_request (...
    );
};
```

3. Request Construction

```
ORBStatus create_request (
    in Context          ctx,          // context object for
                                      // operation
    in Identifier       operation,    // intended operation on
                                      // object
    in NVList           arg_list,     // args to operation
    inout NamedValue    result,       // operation result
    out Request         request,      // newly created request
    in Flags            req_flags     // request flags
);
```

```
struct NamedValue {
    Identifier      name;       // argument name
    any             argument;   // argument
    long            len;        // length/count of argument
                                // value
    Flags           arg_modes;  // argument mode flags
};
```



4. Request Invocation

```
ORBStatus invoke (
    in Flags        invoke_flags    //invocation flags
);

ORBStatus send (
    in Flags        invoke_flags    //invocation flags
);

ORBStatus get_response (
    in Flags        response_flags  //response flags
);                                    RESP_NO_WAIT
```

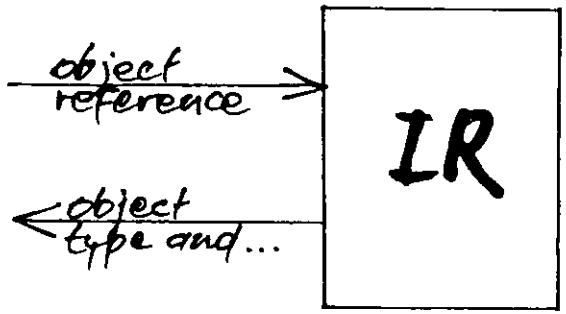
Interface Repository

Typ-Info über Objekte

Aufgaben

- **für ORB:**
  - To provide interoperability between different ORB implementations.
  - To provide type-checking of request signatures, whether a request was issued through the DII or through a stub.
  - To check the correctness of inheritance graphs.
- **für Clients und Benutzer:**
  - To manage installation and distribution of interface definitions around your network.
  - During the development process, for instance, to browse or modify interface definitions or other information stored in IDL.
  - Language compilers could compile stubs and skeletons directly from the IR instead of from the IDL files (all of the required information is contained in both formats).

Inhalte



- The module in which the interface was defined, if any.
- The name of the interface.
- The interface's attributes, and their definitions.
- The interface's operations, and their full definition, including parameter, context exception definitions.
- The inheritance specification of the interface.

Implementation Repository

Info über Objekt-Implementierungen

Aufgaben

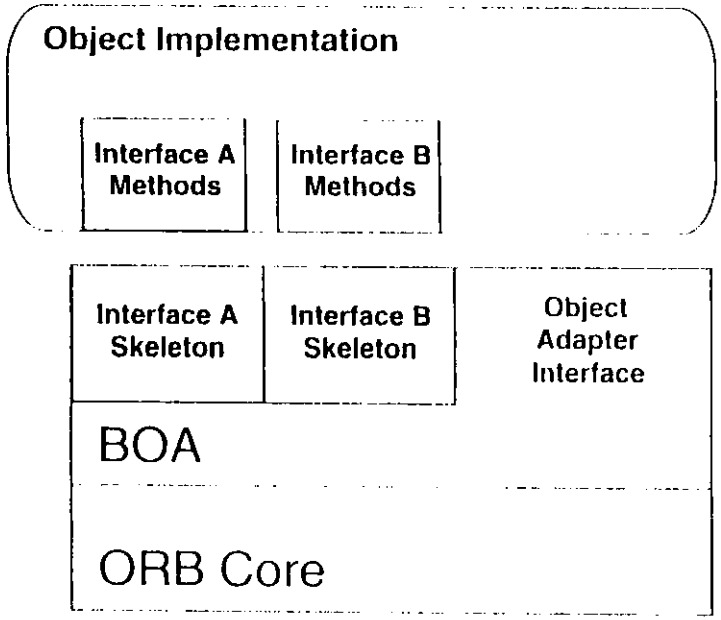
- Server-Registrierung (browsing)
- Server-Aktivierung (activation modes, mapping from server name to exe file path name)
- Zugriffs-Sicherung (owner, permissions; launching, function invocation)

Inhalte

- server name
- exe-file path name
- activation modes
- persistent server
- owner
- permissions
- activation orders

BOA

Basic Object Adapter



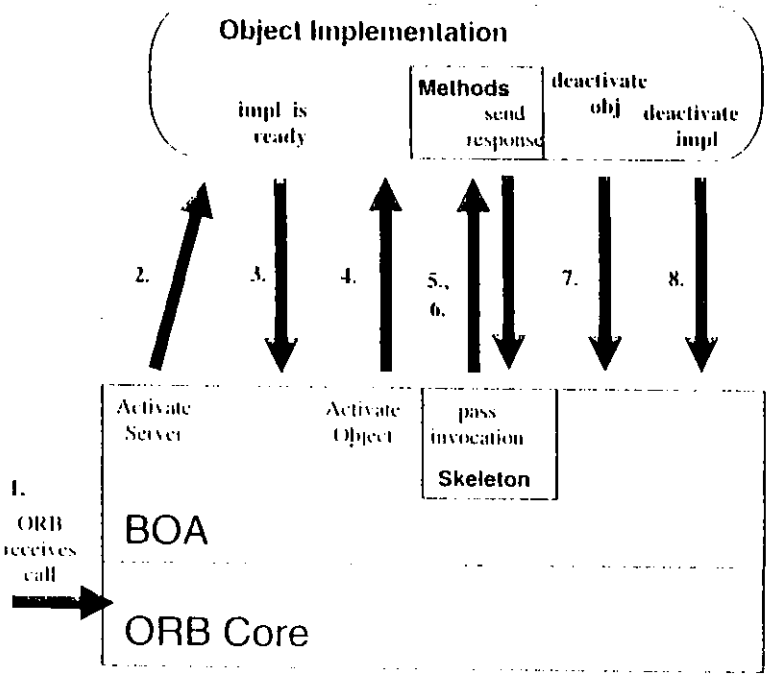
Structure of a typical object adapter. There is one Object Adapter Interface that serves all object implementations. There may be multiple skeletons per Object Implementation and multiple Object Implementations per Object Adapter.

Object Adapters, such as the BOA, are responsible for:

- registering implementations;
- generating and interpreting object references;
- mapping object references to their corresponding implementations;
- activating and deactivating object implementations;
- invoking methods, via a skeleton or the DSI; and
- coordinating interaction security, in cooperation with the forthcoming Security Object service.

BOA

Funktionsweise



Object activation, invocation, and deactivation by the BOA.

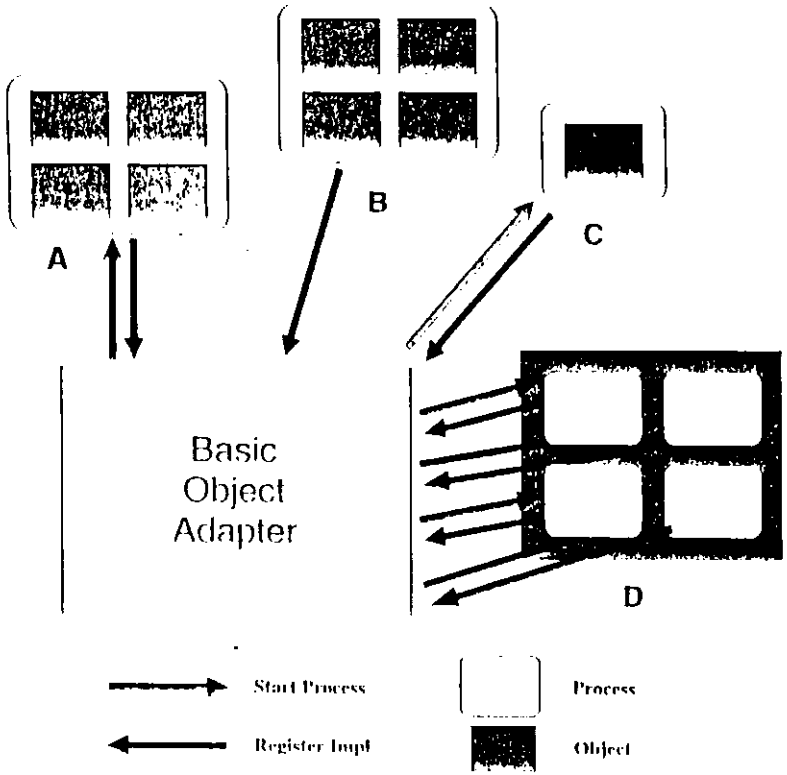
1. The ORB receives a request targeting an object in the server. The ORB checks its repository and determines that neither the server nor the object is currently active.
2. The ORB activates the server, using a system-dependent linkage. As part of the activation process, the server is passed the information it needs to communicate with the BOA. The ORB momentarily holds the original call to the object.
3. The server calls **impl_is_ready**, a call on the BOA, which knows that now the server is ready to activate objects.
4. The BOA calls the server's object activate routine for the target object, passing it the object reference. The server activates the object. If the object had previously been active and stored its state in persistent storage, that information is retrieved and the previous state restored. The key to the persistent store for that object (a filename, database key, or PID for the Persistent Object Service) had been maintained by the BOA, associated with the object reference, and retrieved by the object during startup using **get_id**.
5. The BOA passes the invocation to the object through the skeleton and receives the response, which it routes back to the client.
6. The BOA may receive additional requests on that object, which it passes through the skeleton as in step 5. The BOA may also receive calls for additional objects in the server, which process through steps 4 and then 5 for each new object.
7. The server may, for whatever reason (user request, usage monitoring, and so on) decide to shut down an object. It first calls the BOA routine **deactivate_obj**, specifying the target object, after which the BOA will no longer route calls to that object without reactivating it first. The object saves its state in persistent storage (if it has any such state) before shutting down.
8. Similarly, the server may shut down entirely. It first calls the BOA routine **deactivate_impl**, informing the BOA that it is no longer available to activate objects. On receiving the next request, the BOA will start over at step 1.

Activation Modes

primary server activation modes

Activation Policies:

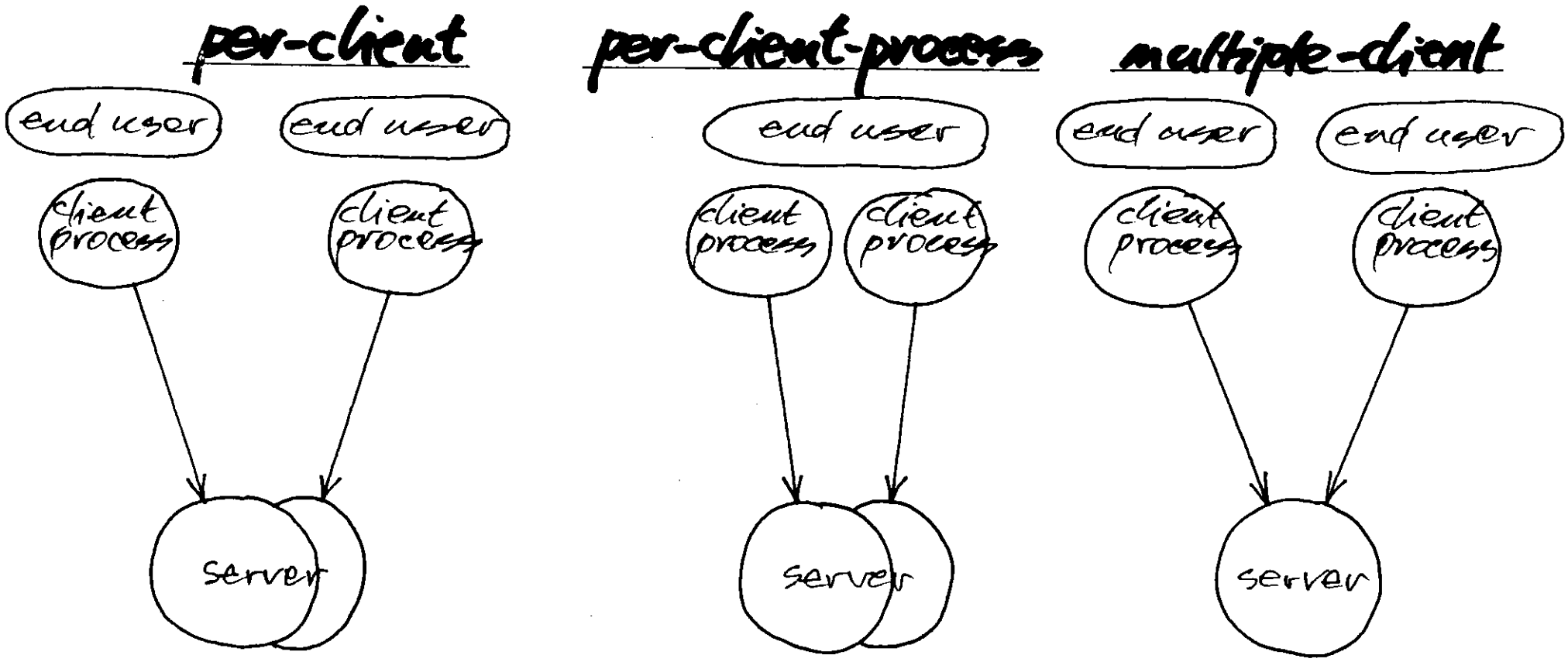
A, shared server; B, persistent server; C, unshared server; D, server per method



- *Shared server policy:* A server activated by the BOA encompassing multiple active objects.
- *Persistent server policy:* Like the shared server except that the server is activated outside of the BOA and registered in an installation procedure.
- *Unshared server policy:* Only one object of a given implementation at a time can be active on a server.
- *Server-per-method policy:* The BOA starts a separate server for each method invocation; the server fulfills the request and then terminates.

Activation Modes

secondary server activation modes



per client: in this mode, activations of the same server by different end users ("principals") will cause a different process to be created for each such end user.

multiple-client: in this mode, activations of the same server by different end users will share the same process, in accordance with whichever fundamental activation mode is selected. This is the default.

per client process: in this mode, activations of the same server by different client processes will cause a different process to be created for each such client process.

Basic Object Adapter

wesentliche Schwachstellen

- *Keine portable Verknüpfung zwischen Skeletons und Servants.* Es ist weder beschrieben, wie die Skeletons aussehen müssen noch wie die Servants mit den Skeletons assoziiert werden. Es sind zwar die Signaturen und die Rümpfe der Servant-Methoden definiert, aber nicht die Basisklassen, von denen die Servants abgeleitet sind.

- *Undefinierte Registrierung der Servants.* Die Implementierungen der Servants müssen auf Serverseite registriert werden, damit sie vom Objektadapter gefunden werden können. Das API zur Registrierung ist jedoch nicht spezifiziert, so dass es CORBA-Produkte gibt, bei denen die Registrierung explizit durch einen bestimmten Methodenaufruf erfolgen muss, und andere Produkte, bei denen die Registrierung implizit durch den Konstruktor des Servants erfolgt.

- *Multithreading ist nicht berücksichtigt.* Um einen hohen Durchsatz bei vielen konkurrierenden Clients zu gewährleisten, kann man i.A. nicht je Kommunikationsbeziehung einen eigenständigen Prozess auf Serverseite bereitstellen. Deshalb ist ein multithreaded CORBA-Server notwendig. In der BOA-Spezifikation wurde Multithreading jedoch nicht berücksichtigt und vollständig den Herstellern überlassen.

- *Bereitschaft für Client-Aufträge ist unpräzise.* Wenn ein Server hochgefahren wird, sind zunächst diverse Vorkehrungen wie z.B. die Erzeugung von Servants notwendig. Ab einem bestimmten Zeitpunkt ist ein Server bereit, Aufträge der Clients, d.h. Methodenaufrufe, durchzuführen. Dazu gibt es beim BOA-Ansatz die beiden Methoden `impl_is_ready` und `obj_is_ready`, die jeweils unter spezifischen Bedingungen aufgerufen werden können. An dieser Stelle ist die BOA-Spezifikation jedoch sehr vage und unpräzise, so dass die verschiedenen Hersteller zwangsläufig verschiedene Implementierungen produziert haben.

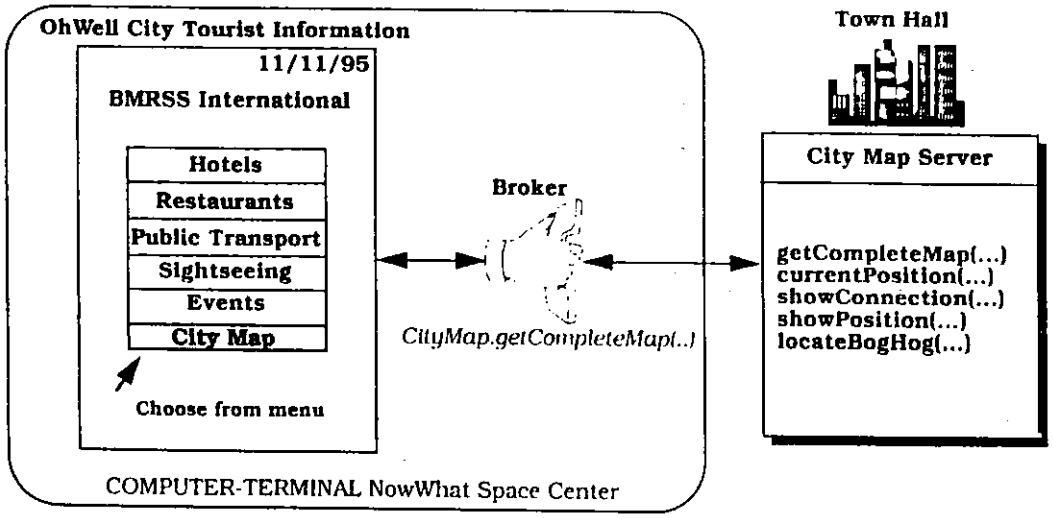
Lösung:

POA

portable Object Adapter

Broker

The *Broker* architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.



Example Suppose we are developing a city information system (CIS) designed to run on a wide area network. Some computers in the network host one or more services that maintain information about events, restaurants, hotels, historical monuments or public transportation. Computer terminals are connected to the network. Tourists throughout the city can retrieve information in which they are interested from the terminals using a World Wide Web (WWW) browser.

Broker

Context Your environment is a distributed and possibly heterogeneous system with independent cooperating components.

Problem Building a complex software system as a set of decoupled and interoperating components, rather than as a monolithic application, results in greater flexibility, maintainability and changeability.

However, when distributed components communicate with each other, some means of inter-process communication is required. If components handle communication themselves, the resulting system faces several dependencies and limitations. For example, the system becomes dependent on the communication mechanism used, clients need to know the location of servers, and in many cases the solution is limited to only one programming language.

Services for adding, removing, exchanging, activating and locating components are also needed. Applications that use these services should not depend on system-specific details to guarantee portability and interoperability, even within a heterogeneous network.

From a developer's viewpoint, there should essentially be no difference between developing software for centralized systems and developing for distributed ones. An application that uses an object should only see the interface offered by the object. It should not need to know anything about the implementation details of an object, or about its physical location.

Use the Broker architecture to balance the following *forces*:

- Components should be able to access services provided by others through remote, location-transparent service invocations.
- You need to exchange, add, or remove components at run-time.
- The architecture should hide system- and implementation-specific details from the users of components and services.

Broker

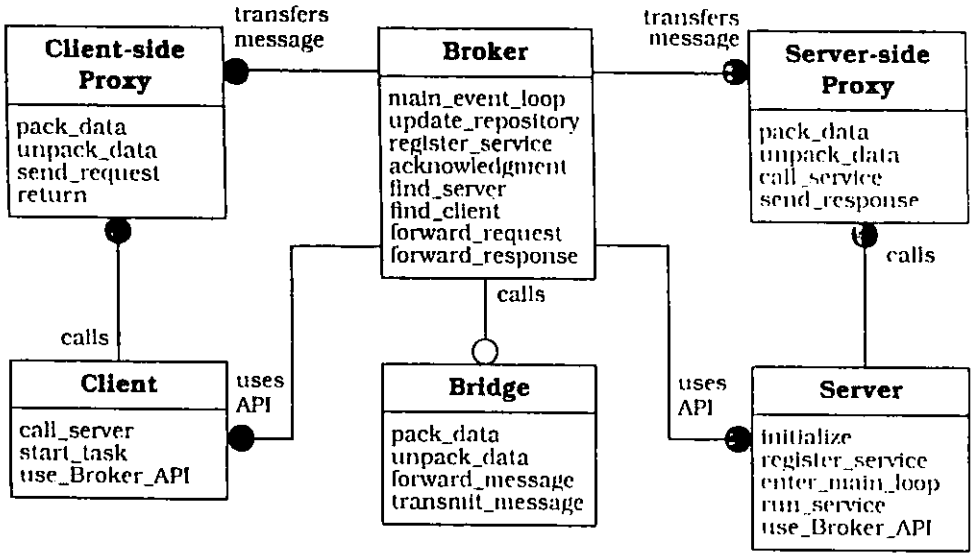
Solution Introduce a *broker* component to achieve better decoupling of clients and servers. Servers register themselves with the broker, and make their services available to clients through method interfaces. Clients access the functionality of servers by sending requests via the broker. A broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

By using the Broker pattern, an application can access distributed services simply by sending message calls to the appropriate object, instead of focusing on low-level inter-process communication. In addition, the Broker architecture is flexible, in that it allows dynamic change, addition, deletion, and relocation of objects.

The Broker pattern reduces the complexity involved in developing distributed applications, because it makes distribution transparent to the developer. It achieves this goal by introducing an object model in which distributed services are encapsulated within objects. Broker systems therefore offer a path to the integration of two core technologies: distribution and object technology. They also extend object models from single applications to distributed applications consisting of decoupled components that can run on heterogeneous machines and that can be written in different programming languages.

Broker

Structure The Broker architectural pattern comprises six types of participating components: *clients*, *servers*, *brokers*, *bridges*, *client-side proxies* and *server-side proxies*.



Broker

Class	Collaborators
Server	• Server-side Proxy • Broker
Responsibility • Implements services. • Registers itself with the local broker. • Sends responses and exceptions back to the client through a server-side proxy.	

Class	Collaborators
Client	• Client-side Proxy • Broker
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	

A *server*[10] implements objects that expose their functionality through interfaces that consist of operations and attributes. These interfaces are made available either through an interface definition language (IDL) or through a binary standard.

➥ The servers in our CIS example comprise WWW servers that provide access to HTML (Hypertext Markup Language) pages. WWW servers are implemented as httpd daemon processes (hypertext transfer protocol daemon) that wait on specific ports for incoming requests. When a request arrives at the server, the requested document and any additional data is sent to the client using data streams. The HTML pages contain documents as well as CGI (Common Gateway interface) scripts for remotely-executed operations on the network host—the remote machine from which the client received the HTML-page. A CGI script may be used to allow the user fill out a form and submit a query, for example a search request for vacant hotel rooms. To display animations on the client's WWW browser, Java 'applets' are integrated into the HTML documents.

Clients are applications that access the services of at least one server. To call remote services, clients forward requests to the broker. After an operation has executed they receive responses or exceptions from the broker.

➥ In the context of the Broker pattern, the clients are the available WWW browsers. They are not directly connected to the network. Instead, they rely on Internet providers that offer gateways to the Internet, such as Compuserve. WWW browsers connect to these workstations, using either a modem or a leased line. When connected they are able to retrieve data streams from httpd servers, interpret this data and initiate actions such as the display of documents on the screen or the execution of Java applets. ❑

Broker

A *broker* is a messenger that is responsible for the transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client. A broker must have some means of locating the receiver of a request based on its unique system identifier. A broker offers APIs (Application Programming Interfaces) to clients and servers that include operations for registering servers and for invoking server methods.

When a request arrives for a server that is maintained by the local broker[11], the broker passes the request directly to the server. If the server is currently inactive, the broker activates it. All responses and exceptions from a service execution are forwarded by the broker to the client that sent the request. If the specified server is hosted by another broker, the local broker finds a route to the remote broker and forwards the request using this route. There is therefore a need for brokers to interoperate.

Depending on the requirements of the whole system, additional services—such as *name services*[12] or *marshaling support*[13]—may be integrated into the broker.

Class Broker	**Collaborators**
Responsibility • (Un-)registers servers. • Offers APIs. • Transfers messages. • Error recovery. • Interoperates with other brokers through bridges. • Locates servers.	• Client • Server • Client-side Proxy • Server-side Proxy • Bridge

➥ A broker in our CIS example is the combination of an Internet gateway and the Internet infrastructure itself. Every information exchange between a client and a server must pass through the broker. A client specifies the information it wants using unique identifiers called URLs (Universal Resource Locators). By using these identifiers the broker is able to locate the required services and to route the requests to the appropriate server machines. When a new server machine is added, it must be registered with the broker. Clients and servers use the gateway of their Internet provider as an interface to the broker. ❑

12. Name services provide associations between names and objects. To resolve a name, a name service determines which server is associated with a given name. In the context of Broker systems, names are only meaningful relative to a *name space*.

13. Marshaling is the semantic-invariant conversion of data into a machine-independent format such as ASN.1 (Abstract Syntax Notation) or ONC XDR (eXternal Data Representation). Unmarshaling performs the reverse transformation.

11. In this pattern description we distinguish between *local* and *remote* brokers. A local broker is running on the machine currently under consideration. A remote broker is running on a remote network node.

Broker

Class	Collaborators
Client-side Proxy	• Client • Broker
Responsibility	
• Encapsulates system-specific functionality. • Mediates between the client and the broker.	

Class	Collaborators
Server-side Proxy	• Server • Broker
Responsibility	
• Calls services within the server. • Encapsulates system-specific functionality. • Mediates between the server and the broker.	

Client-side proxies represent a layer between clients and the broker. This additional layer provides transparency, in that a remote object appears to the client as a local one. In detail, the proxies allow the hiding of implementation details from the clients such as:

- The inter-process communication mechanism used for message transfers between clients and brokers.
- The creation and deletion of memory blocks.
- The marshaling of parameters and results.

In many cases, client-side proxies translate the object model specified as part of the Broker architectural pattern to the object model of the programming language used to implement the client.

Server-side proxies are generally analogous to Client-side proxies. The difference is that they are responsible for receiving requests, unpacking incoming messages, unmarshaling the parameters, and calling the appropriate service. They are used in addition for marshaling results and exceptions before sending them to the client.

When results or exceptions are returned from a server, the Client-side proxy receives the incoming message from the broker, unmarshals the data and forward it to the client.

➥ In our CIS example the WWW browsers and httpd servers such as Netscape provide built-in capabilities for communicating with the gateway of the Internet provider, so we do not need to worry about proxies in this case. ❑

Broker

Bridges[14] are optional components used for hiding implementation details when two brokers interoperate. Suppose a Broker system runs on a heterogeneous network. If requests are transmitted over the network, different brokers have to communicate independently of the different network and operating systems in use. A bridge builds a layer that encapsulates all these system-specific details.

There are two different kinds of Broker systems: those using direct communication and those using indirect communication. To achieve better performance, some broker implementations only establish the initial communication link between a client and a server, while the rest of the communication is done directly between participating components—messages, exceptions and responses are transferred between client-side proxies and server-side proxies without using the broker as an intermediate layer. This direct communication approach requires that servers and clients use and understand the same protocol. In this pattern description we focus on the Indirect Broker variant, where all messages are passed through the broker. The Client-Dispatcher-Server pattern (323) describes the important aspects of the direct variant of the Broker pattern.

Class Bridge	***Collaborators*** • Broker • Bridge
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	

➥ Bridges are not required in our CIS example, because all httpd servers and WWW browsers implement the protocols necessary for remote data exchange such as `http` (hypertext transfer protocol) or `ftp` (file transfer protocol). ❑

➥ Our CIS example implements the indirect communication variant, because browsers and servers can only collaborate using Inter-net gateways. There is one place in CIS however where we use the direct communication variant instead—Java applets loaded from the network may connect directly to the WWW server from which they came using a socket connection. ❑

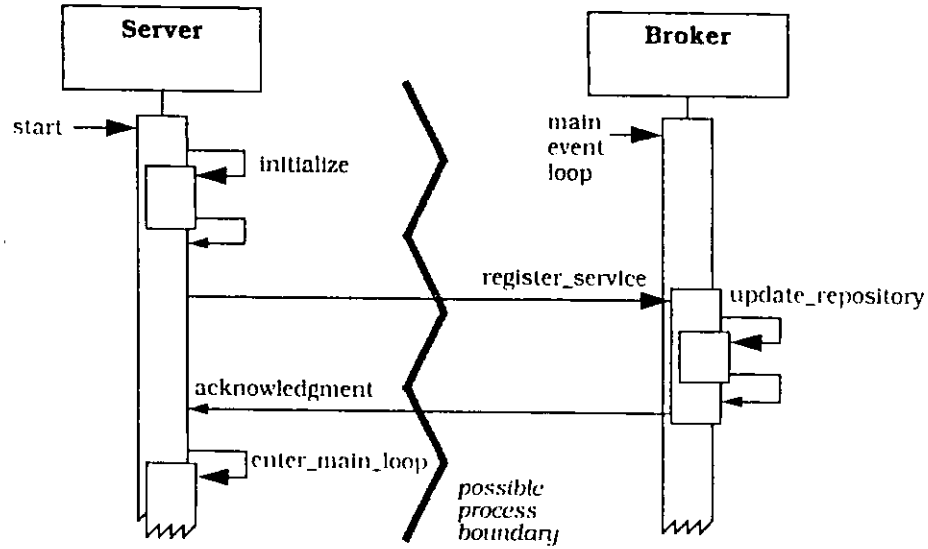
14. We call these components *Bridges* following the terminology of the OMG in the CORBA 2 specification.

Broker

Dynamics This section focuses on the most relevant scenarios in the operation of a Broker system.

Scenario I illustrates the behavior when a server registers itself with the local broker component:

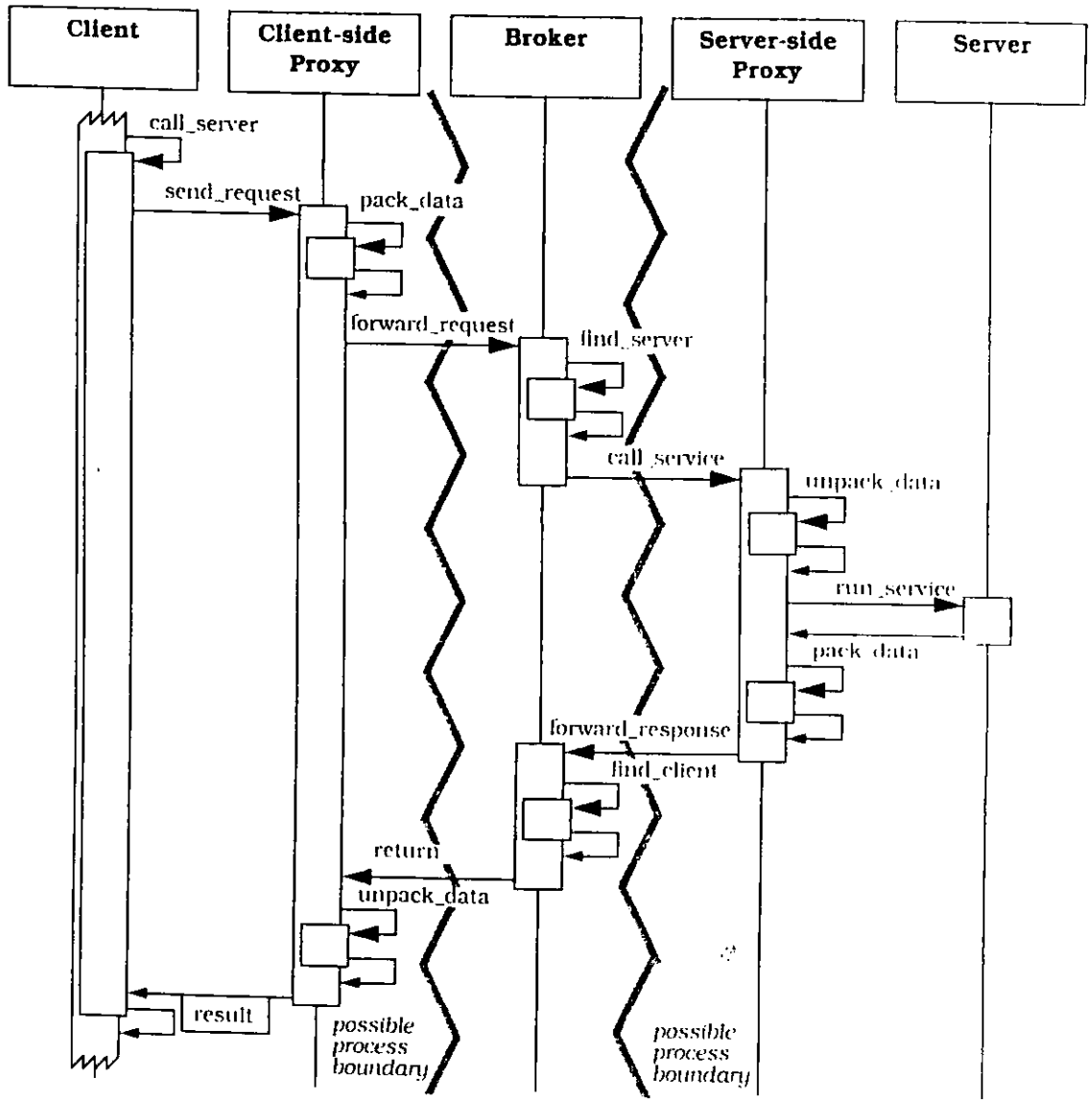
- The broker is started in the initialization phase of the system. The broker enters its event loop and waits for incoming messages.
- The user, or some other entity, starts a server application. First, the server executes its initialization code. After initialization is complete, the server registers itself with the broker.
- The broker receives the incoming registration request from the server. It extracts all necessary information from the message and stores it into one or more repositories. These repositories are used to locate and activate servers. An acknowledgment is sent back.
- After receiving the acknowledgment from the broker, the server enters its main loop waiting for incoming client requests.



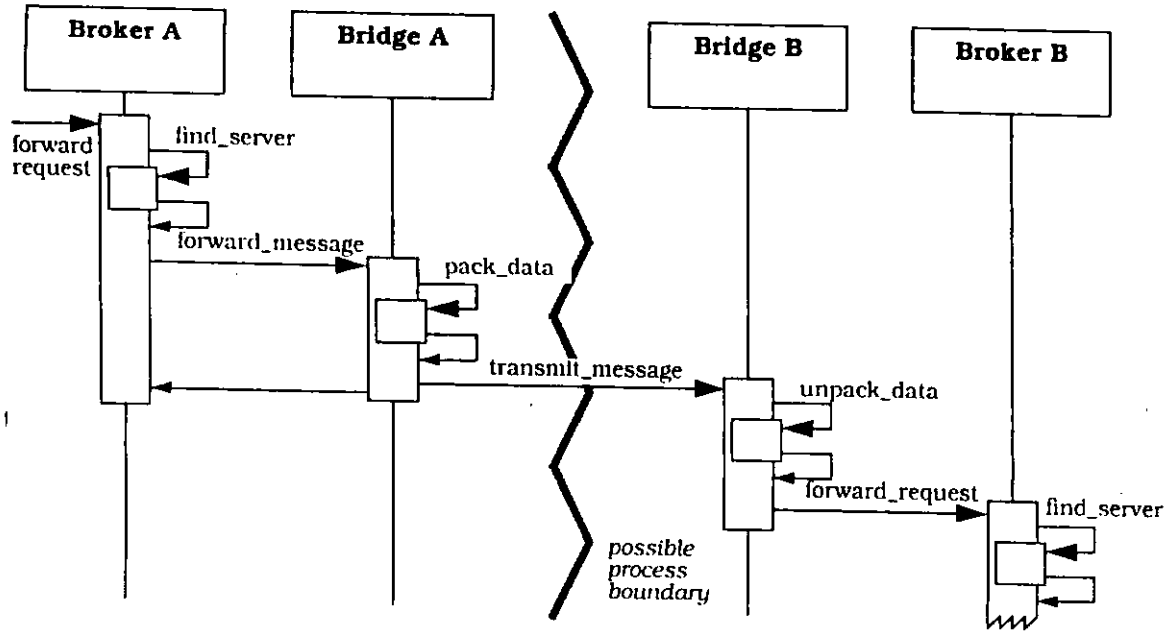
Broker

Scenario II illustrates the behavior when a client sends a request to a local server. In this scenario we describe a synchronous invocation, in which the client blocks until it gets a response from the server. The broker may also support asynchronous invocations, allowing clients to execute further tasks without having to wait for a response.

- The client application is started. During program execution the client invokes a method of a remote server object.
- The client-side proxy packages all parameters and other relevant information into a message and forwards this message to the local broker.
- The broker looks up the location of the required server in its repositories. Since the server is available locally, the broker forwards the message to the corresponding server-side proxy. For the remote case, see the following scenario.
- The server-side proxy unpacks all parameters and other information, such as the method it is expected to call. The server-side proxy invokes the appropriate service.
- After the service execution is complete, the server returns the result to the server-side proxy, which packages it into a message with other relevant information and passes it to the broker.
- The broker forwards the response to the client-side proxy.
- The client-side proxy receives the response, unpacks the result and returns to the client application. The client process continues with its computation.



Broker



Scenario III illustrates the interaction of different brokers via bridge components:

- Broker A receives an incoming request. It locates the server responsible for executing the specified service by looking it up in the repositories. Since the corresponding server is available at another network node, the broker forwards the request to a remote broker.
- The message is passed from Broker A to Bridge A. This component is responsible for converting the message from the protocol defined by Broker A to a network-specific but common protocol understood by the two participating bridges. After message conversion, Bridge A transmits the message to Bridge B.
- Bridge B maps the incoming request from the network-specific format to a Broker B-specific format.
- Broker B performs all the actions necessary when a request arrives, as described in the first step of this scenario.

Adapter

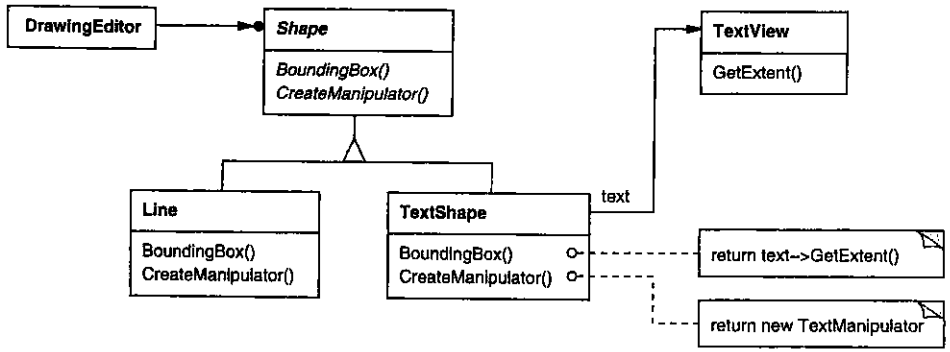
Entwurfsmuster

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Motivation

Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.



Classes for elementary geometric shapes like LineShape and PolygonShape are rather easy to implement, because their drawing and editing capabilities are inherently limited. But a TextShape subclass that can display and edit text is considerably more difficult to implement, since even basic text editing involves complicated screen update and buffer management. Meanwhile, an off-the-shelf user interface toolkit might already provide a sophisticated TextView class for displaying and editing text. Ideally we'd like to reuse TextView to implement TextShape, but the toolkit wasn't designed with Shape classes in mind. So we can't use TextView and Shape objects interchangeably.

How can existing and unrelated classes like TextView work in an application that expects classes with a different and incompatible interface? We could change the TextView class so that it conforms to the Shape interface, but that isn't an option unless we have the toolkit's source code. Even if we did, it wouldn't make sense to change TextView; the toolkit shouldn't have to adopt domain-specific interfaces just to make one application work.

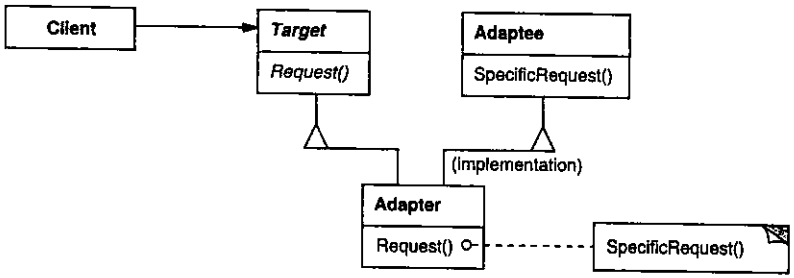
Instead, we could define TextShape so that it *adapts* the TextView interface to Shape's. We can do this in one of two ways: (1) by inheriting Shape's interface and TextView's implementation or (2) by composing a TextView instance within a TextShape and implementing TextShape in terms of TextView's interface.

Adapter

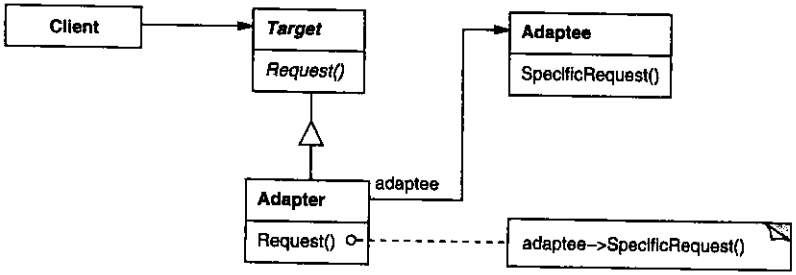
Entwurfsmuster

Structure

A class adapter uses multiple inheritance to adapt one interface to another:



An object adapter relies on object composition:



Participants

- **Target** (Shape)
  - defines the domain-specific interface that Client uses.
- **Client** (DrawingEditor)
  - collaborates with objects conforming to the Target interface.
- **Adaptee** (TextView)
  - defines an existing interface that needs adapting.
- **Adapter** (TextShape)
  - adapts the interface of Adaptee to the Target interface.

Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Consequences

- Class and object adapters have different trade-offs. A class adapter

  - adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
  - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
  - introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

- An object adapter

  - lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
  - makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Proxy

Surrogate

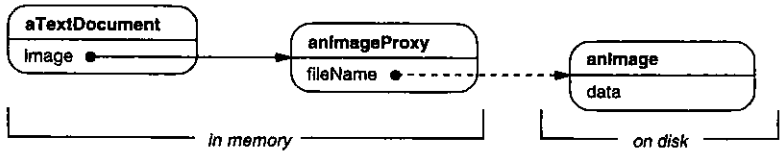
Intent

Provide a surrogate or placeholder for another object to control access to it.

Motivation

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened.

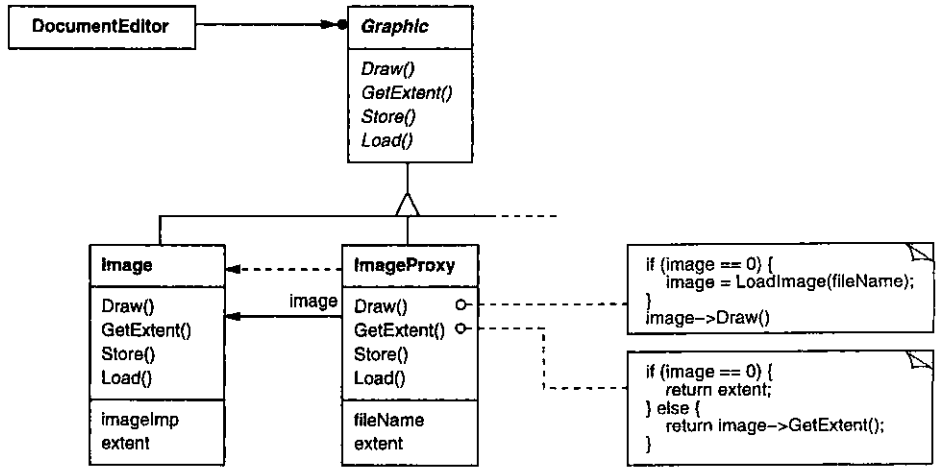
The solution is to use another object, an image **proxy**, that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.



The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation. The proxy forwards subsequent requests directly to the image. It must therefore keep a reference to the image after creating it.

Let's assume that images are stored in separate files. In this case we can use the file name as the reference to the real object. The proxy also stores its **extent**, that is, its width and height. The extent lets the proxy respond to requests for its size from the formatter without actually instantiating the image.

The following class diagram illustrates this example in more detail.



The document editor accesses embedded images through the interface defined by the abstract Graphic class. ImageProxy is a class for images that are created on demand. ImageProxy maintains the file name as a reference to the image on disk. The file name is passed as an argument to the ImageProxy constructor.

Applicability

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP [Add94] uses the class NXProxy for this purpose. Coplien [Cop92] calls this kind of proxy an "Ambassador."
2. A **virtual proxy** creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.
3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system [CIRM93] provide protected access to operating system objects.
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include
   - counting the number of references to the real object so that it can be freed automatically when there are no more references (also called **smart pointers** [Ede92]).
   - loading a persistent object into memory when it's first referenced.
   - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Dereference

Operator →

Aufgabe:

Der *dereference*-Operator -> kann als unärer Postfix-Operator überladen werden. Bei einer gegebenen Klasse

```
class Ptr {
    // ...
    X* operator->();
};
```

können Objekte der Klasse **Ptr** verwendet werden, um auf Elemente einer Klasse **X** auf eine Weise zuzugreifen, die dem Zugriff auf Pointer ähnelt;

Syntax:

```
void f(Ptr p)
{
    p->m = 7; // (p.operator->())->m = 7
}
```

Die Transformation des Objekts **p** in den Pointer **p.operator->()** ist vom Element **m** unabhängig. Dies ist gemeint, wenn man den **operator->()** als Postfix-Operator bezeichnet. Es wird jedoch keine neue syntaktische Einbindung eingeführt, so daß hinter -> weiterhin ein Element-Name erwartet wird:

```
void g(Ptr p)
{
    X* q1 = p->;  // Syntaxfehler
    X* q2 = p.operator->(); // ok
}
```

Äquivalenzen:

Durch zusätzliches Überladen von [] und * :

```
class  X {
    Y* p;
public:
    Y* operator->() { return p; }
    Y& operator*() { return *p; }
    Y& operator[](int i) { return p[i]; }
};
```

erreicht man, daß (wie üblich) gilt:

$$p\to m == (*p).m == p[0].m$$

Dereference

Operator →

Das Überladen von -> dient im Wesentlichen dazu, sogenannte *smart pointers* zu implementieren, also Objekte, die sich wie ein Pointer verhalten, bei jedem Zugriff jedoch zusätzliche Aktionen ausführen können.

Prinzip

A) ~~Ersetze~~ "Zeiger p auf Objekt vom Typ X" durch "Objekt sp vom Typ SmartPointer"

B) Redefiniere für Klasse SmartPointer den Operator "->" mit Returnwert vom Typ X*"

---

d.h., der Operator "->" wandelt das Objekt sp um in den Zeigerwert X*.

Bei der Umwandlung können viele sinnvolle ("smarte") Nebenaktionen durchgeführt werden, wie z.B. Speicher-Zugriffe und Transformationen

Dereference

Beispiel 1

Beispielsweise kann man eine Klasse **RecPtr** definieren, mit der auf Objekte einer Klasse **Rec** zugegriffen werden kann, die auf Platte gespeichert sind. Der Konstruktor für **RecPtr** nimmt als Argument einen Namen, der dazu dient, das Objekt auf der Platte zu finden. **RecPtr::operator->()** bringt das Objekt in den Hauptspeicher, wenn ein Zugriff über **RecPtr** erfolgt; **RecPtrs** Destruktor wird das modifzierte Objekt dann eventuell auf die Platte zurückschreiben:

```
class RecPtr {
    Rec* in_core_adress;
    const char* identifier;
    // ...
public:
    RecPtr(const char* p)
        : identifier(p) { in_core_adress = 0; }
    ~RecPtr()
        { write_to_disc(in_core_address,identifier); }
    Rec* operator->();
};
```

```
Rec* RecPtr::operator->()
{
    if (in_core_adress == 0)
        in_core_adress = read_from_disc(identifier);
    return in_core_adress;
}
```

Ein Anwendungsbeispiel:

```
main(int argc, const char* argv[])
{
    for (int i = argc; i; i--) {
        RecPtr p(argv[i]);
        p->update();
    }
}
```

Dereference Beispiel 2

One might define a Text class to store and `display()` each discrete story text segment and accompanying picture.

```
class Text {
public:
    void display();
    // ...
private:
    String *text;
    Picture *pic;
};

class Story {
public:
    Text *operator->();
    int get_choice();
    void get_text_segment();
    // ...
private:
    // holds array of possible
    // next story text segments
    Text *text_array[];
};

Text*
Story::operator->() {
    if (text_array == 0)
        get_text_segment();
    return text_array[ get_choice() ];
}
```

```
// constructor displays first text segment
Story story;

main() {
    while (story.continue()) {
        // overloaded operator->()
        // returning a Text* object
        story->display();
        story.get_choice();
        // ...
    }
}
```

The expression

```
story->display();
```

is resolved in two steps. First, the overloaded instance of the operator is invoked. Second, its return value — the Text pointer — is bound to the actual pointer operator and the `Text::display()` member function is invoked.

Dereference

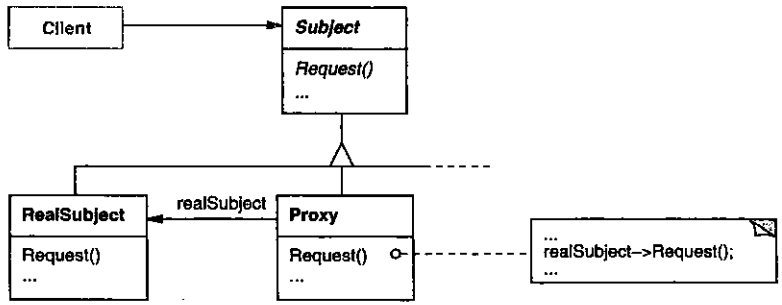
Ausführung des Operators "->"
Semantik

```
perform (x -> mem())
    {
      if (x.isPointer)
          x->mem();

      else
      if (x.has->operator)
          perform ((x.operator->())->mem());

      else
          ERROR;
    }
```

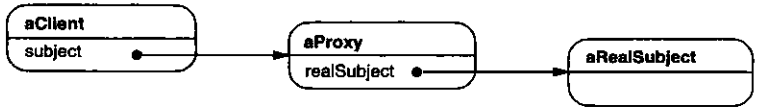
Proxy

Entwurfsmuster

Structure



Here's a possible object diagram of a proxy structure at run-time:



Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Participants

- **Proxy** (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilities depend on the kind of proxy:
    - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
    - *protection proxies* check that the caller has the access permissions required to perform a request.
- **Subject** (Graphic)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject** (Image)
  - defines the real object that the proxy represents.

Implementation

The Proxy pattern can exploit the following language features:

1. *Overloading the member access operator in C++.* C++ supports overloading `operator->`, the member access operator. Overloading this operator lets you perform additional work whenever an object is dereferenced. This can be helpful for implementing some kinds of proxy; the proxy behaves just like a pointer.

   The following example illustrates how to use this technique to implement a virtual proxy called `ImagePtr`.

```
class Image;
extern Image* LoadAnImageFile(const char*);
    // external function

class ImagePtr {
public:
    ImagePtr(const char* imageFile);
    virtual ~ImagePtr();

    virtual Image* operator->();
    virtual Image& operator*();
private:
    Image* LoadImage();
private:
    Image* _image;
    const char* _imageFile;
};

ImagePtr::ImagePtr (const char* theImageFile) {
    _imageFile = theImageFile;
    _image = 0;
}

Image* ImagePtr::LoadImage () {
    if (_image == 0) {
        _image = LoadAnImageFile(_imageFile);
    }
    return _image;
}
```

   The overloaded `->` and `*` operators use `LoadImage` to return `_image` to callers (loading it if necessary).

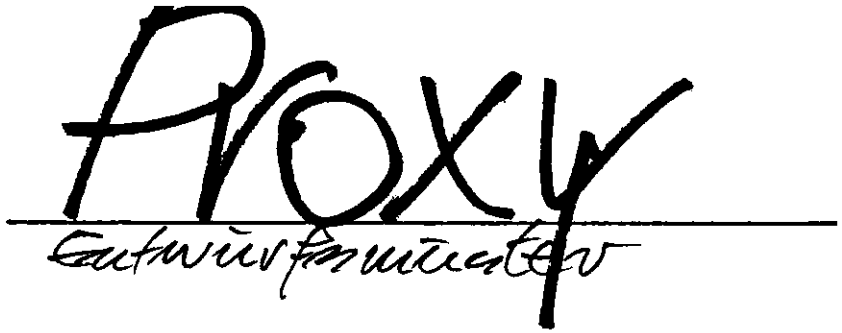
```
Image* ImagePtr::operator-> () {
    return LoadImage();
}

Image& ImagePtr::operator* () {
    return *LoadImage();
}
```

   This approach lets you call `Image` operations through `ImagePtr` objects without going to the trouble of making the operations part of the `ImagePtr` interface:

```
ImagePtr image = ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
    // (image.operator->())->Draw(Point(50, 100))
```

3. *Proxy doesn't always have to know the type of real subject.* If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubject classes uniformly. But if Proxies are going to instantiate RealSubjects (such as in a virtual proxy), then they have to know the concrete class.

Another implementation issue involves how to refer to the subject before it's instantiated. Some proxies have to refer to their subject whether it's on disk or in memory. That means they must use some form of address space-independent object identifiers. We used a file name for this purpose in the Motivation.



Sample Code

A virtual proxy. The `Graphic` class defines the interface for graphical objects:

```
class Graphic {
public:
    virtual ~Graphic();

    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
protected:
    Graphic();
};
```

The `Image` class implements the `Graphic` interface to display image files. `Image` overrides `HandleMouse` to let users resize the image interactively.

```
class Image : public Graphic {
public:
    Image(const char* file);  // loads image from a file
    virtual ~Image();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();
    virtual void Load(istream& from);
    virtual void Save(ostream& to);
private:
    // ...
};
```

`ImageProxy` has the same interface as `Image`:

```
class ImageProxy : public Graphic {
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
protected:
    Image* GetImage();
private:
    Image* _image;
    Point _extent;
    char* _fileName;
};
```

The constructor saves a local copy of the name of the file that stores the image, and it initializes `_extent` and `_image`:

```
ImageProxy::ImageProxy (const char* fileName) {
    _fileName = strdup(fileName);
    _extent = Point::Zero;  // don't know extent yet
    _image = 0;
}

Image* ImageProxy::GetImage() {
    if (_image == 0) {
        _image = new Image(_fileName);
    }
    return _image;
}
```

The implementation of `GetExtent` returns the cached extent if possible; otherwise the image is loaded from the file. `Draw` loads the image, and `HandleMouse` forwards the event to the real image.

```
const Point& ImageProxy::GetExtent () {
    if (_extent == Point::Zero) {
        _extent = GetImage()->GetExtent();
    }
    return _extent;
}

void ImageProxy::Draw (const Point& at) {
    GetImage()->Draw(at);
}

void ImageProxy::HandleMouse (Event& event) {
    GetImage()->HandleMouse(event);
}
```

The `Save` operation saves the cached image extent and the image file name to a stream. `Load` retrieves this information and initializes the corresponding members.

```
void ImageProxy::Save (ostream& to) {
    to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
    from >> _extent >> _fileName;
}
```

Finally, suppose we have a class `TextDocument` that can contain `Graphic` objects:

```
class TextDocument {
public:
    TextDocument();

    void Insert(Graphic*);
    // ...
};
```

We can insert an `ImageProxy` into a text document like this:

```
TextDocument* text = new TextDocument;
// ...
text->Insert(new ImageProxy("anImageFileName"));
```

Broker

Direct versus Indirect Communication

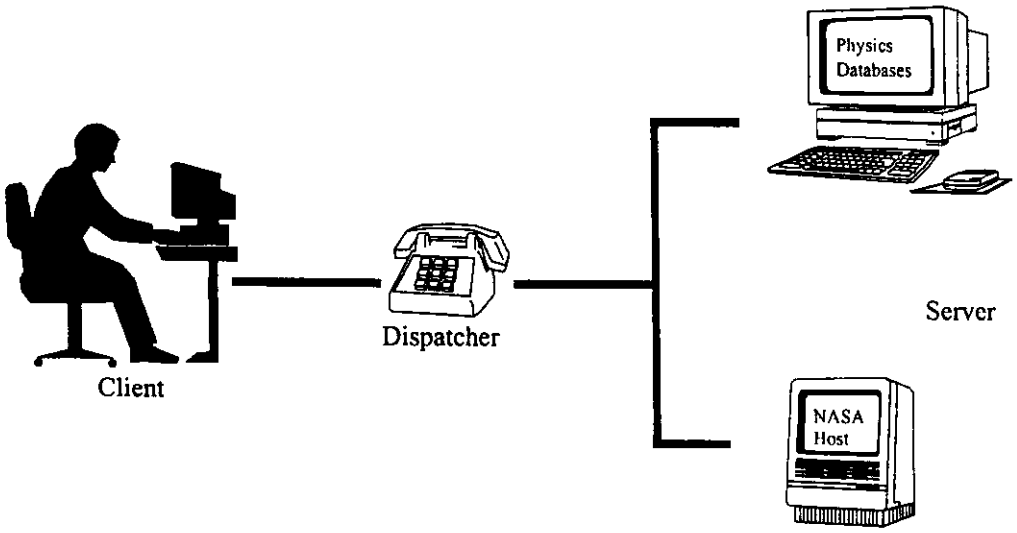
There are two different kinds of Broker systems: those using direct communication and those using indirect communication. To achieve better performance, some broker implementations only establish the initial communication link between a client and a server, while the rest of the communication is done directly between participating components—messages, exceptions and responses are transferred between client-side proxies and server-side proxies without using the broker as an intermediate layer. This direct communication approach requires that servers and clients use and understand the same protocol. In this pattern description we focus on the Indirect Broker variant, where all messages are passed through the broker. The Client-Dispatcher-Server pattern (323) describes the important aspects of the direct variant of the Broker pattern.

Client-Dispatcher-Server

Entwurfsmuster

The *Client-Dispatcher-Server* design pattern introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.

Example Imagine we are developing a software system ACHILLES for the retrieval of new scientific information. The information providers are both on our local network and distributed over the world. To access an individual information provider, it is necessary to specify its location and the service to be executed. When an information provider receives a request from a client application, it runs the appropriate service and returns the requested information to the client.



Context A software system integrating a set of distributed servers, with the servers running locally or distributed over a network.

Problem When a software system uses servers distributed over a network it must provide a means for communication between them. In many cases a connection between components may have to be established before the communication can take place, depending on the available communication facilities. However, the core functionality of the components should be separate from the details of communication mechanisms. Clients should not need to know where servers are located. This allows you to change the location of servers dynamically, and provides resilience to network or server failures.

We have to balance the following *forces*:

- A component should be able to use a service independent of the location of the service provider.
- The code implementing the functional core of a service consumer should be separate from the code used to establish a connection with service providers.

C-D-S

Client-Dispatcher-Server

tructure

The task of a *client* is to perform domain-specific tasks. The client accesses operations offered by servers in order to carry out its processing tasks. Before sending a request to a server, the client asks the dispatcher for a communication channel. The client uses this channel to communicate with the server.

A *server* provides a set of operations to clients. It either registers itself or is registered with the dispatcher by its name and address. A server component may be located on the same computer as a client, or may be reachable via a network.

Class Client	***Collaborators***
Responsibility	• Dispatcher • Server
• Implements a system task. • Requests server connections from the dispatcher. • Invokes services of servers.	

Class Server	***Collaborators***
Responsibility	• Client • Dispatcher
• Provides services to clients. • Registers itself with the dispatcher.	

The *dispatcher* offers functionality for establishing communication channels between clients and servers. To do this, it takes the name of a server component and maps this name to the physical location of the server component. The dispatcher establishes a communication link to the server using the available communication mechanism and returns a communication handle to the client. If the dispatcher cannot initiate a communication link with the requested server, it informs the client about the error it encountered.

To provide its name service, the dispatcher implements functions for registering and locating servers.

Class Dispatcher	***Collaborators***
Responsibility	• Client • Server
• Establishes communication channels between clients and servers. • Locates servers. • (Un-)Registers servers. • Maintains a map of server locations.	

The **OMG Corba** (Common Object Request Broker Architecture) specification [OMG92] uses the principles of the Client-Dispatcher-Server design pattern for refining and instantiating the Broker architectural pattern (99)

C-D-S

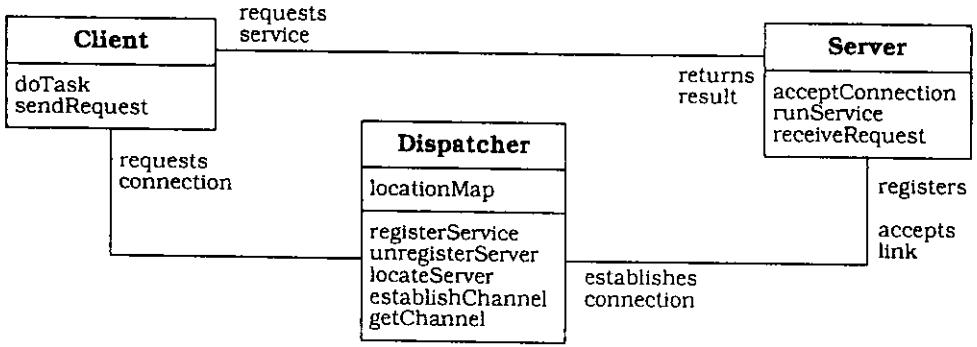
Client-Dispatcher-Server

Solution Provide a *dispatcher* component to act as an intermediate layer between *clients* and *servers*. The dispatcher implements a name service that allows clients to refer to servers by names instead of physical locations, thus providing location transparency. In addition, the dispatcher is responsible for establishing the communication channel between a client and a server.

Add servers to the application that provides services to other components. Each server is uniquely identified by its name, and is connected to clients by the dispatcher.

Clients rely on the dispatcher to locate a particular server and to establish a communication link with the server. In contrast to traditional Client-Server computing, the roles of clients and servers can change dynamically.

The static relationships between clients, servers and the dispatcher are as follows:

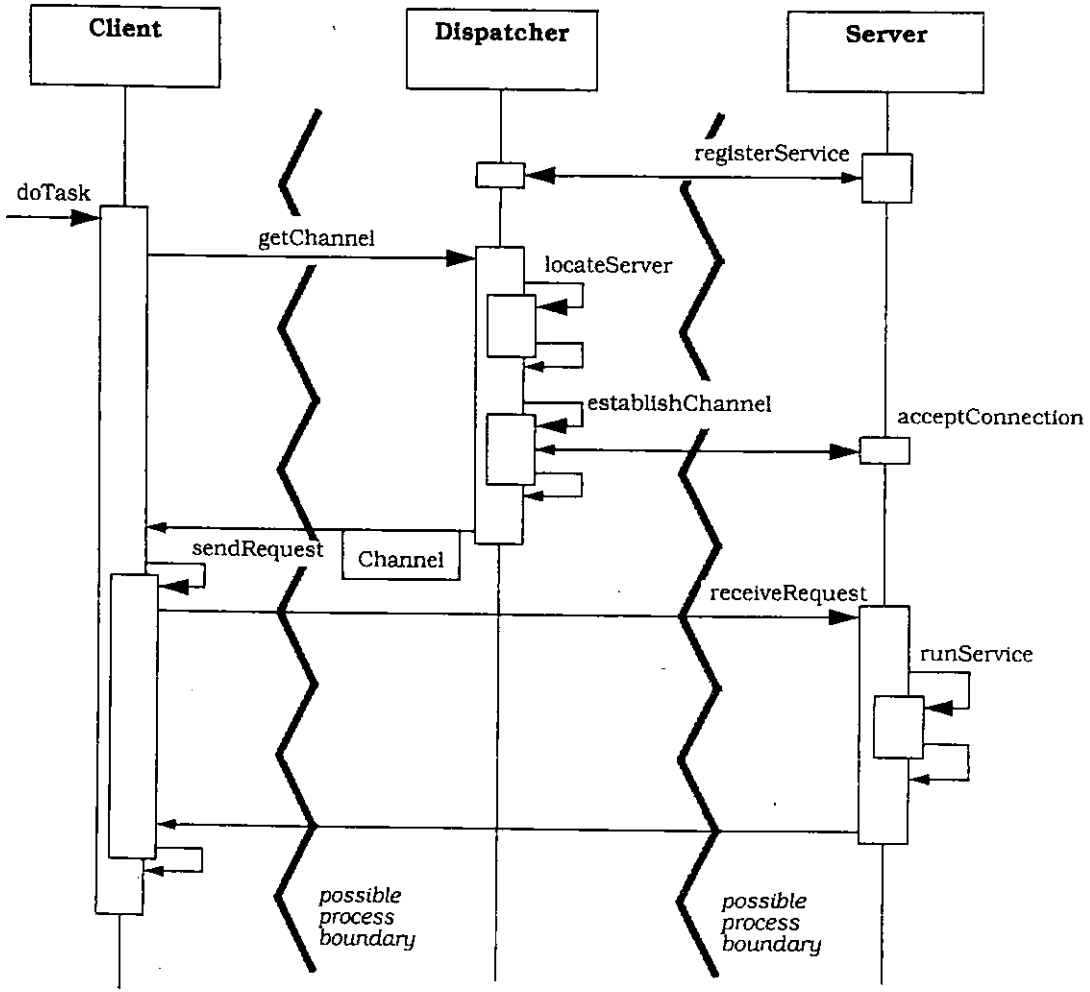


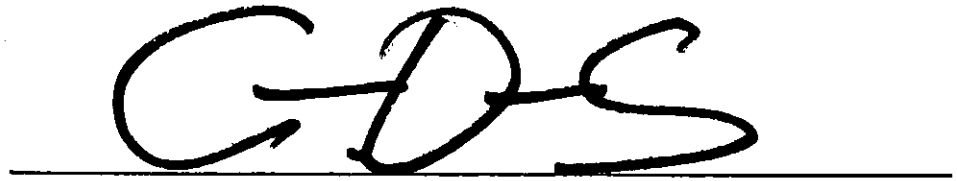
C-D-S

Client-Dispatcher-Server

Dynamics A typical scenario for the Client-Dispatcher-Server design pattern includes the following phases:

- A server registers itself with the dispatcher component.
- At a later time, a client asks the dispatcher for a communication channel to a specified server.
- The dispatcher looks up the server that is associated with the name specified by the client in its registry.
- The dispatcher establishes a communication link to the server. If it is able to initiate the connection successfully, it returns the communication channel to the client. If not, it sends the client an error message.
- The client uses the communication channel to send a request directly to the server.
- After recognizing the incoming request, the server executes the appropriate service.
- When the service execution is completed, the server sends the results back to the client.





Client-Dispatcher-Server

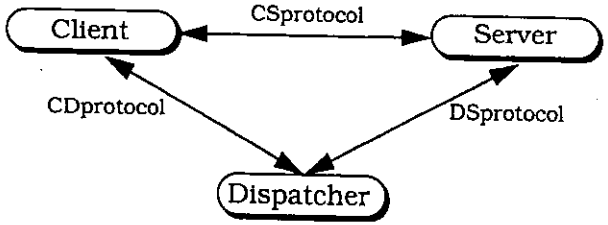
Implementation

To implement a Client-Dispatcher-Server structure, apply the following steps. You do not necessarily need to follow the steps in the order given, because some of them are interrelated.

1 *Separate the application into servers and clients.* Define which components should be implemented as servers, and identify the clients that will access these servers. Since clients may also act as servers, and vice-versa—their roles are not predefined and may change at run-time.

2 *Decide which communication facilities are required.* Select communication facilities for the interaction between clients and the dispatcher, between servers and the dispatcher and between clients and servers. For example, if the dispatcher and the clients accessing it are on the same machine, shared memory is the fastest method of inter-process communication. In this example, clients may communicate with the dispatcher using shared memory, but the servers and the dispatcher, as well as clients and servers, could communicate using sockets.

3 *Specify the interaction protocols between components.* Consider the following diagram:



4 *Decide how to name servers.* The four-byte Internet IP address scheme is not applicable, because it does not provide location transparency. If IP addresses were used, a client would depend on the concrete location of the server. You need to introduce names that uniquely identify servers but do not carry any location information. For example, use strings such as 'ServerX' or predefined constants such as ID_SERVER_X. These location-independent names are mapped to physical locations by the dispatcher (see step 5).

5 *Design and implement the dispatcher.* Determine how the protocols you introduced in step 3 should be implemented using available communication facilities.

With some communication mechanisms the available communication channels may be a limited resource. For example, the number of socket descriptors is constrained by the size of descriptor tables in the operating system. There are several ways round this. For example, each server may allocate its own socket, limiting the number of possible servers. When a client request arrives, the dispatcher returns the server's socket descriptor to the client. Alternatively, the dispatcher could temporarily store client requests in an internal message queue. It would then provide a socket port where servers can ask whether new requests have arrived. When a service request arrives, the server opens a new socket and passes the new socket descriptor to the dispatcher. The dispatcher then forwards the information to the client. After the interaction between client and server is completed, the server closes its socket descriptor.

A dispatcher includes a repository for mapping server names to their physical locations. The representation of server locations depends on the underlying mechanism you use for Client-Server communication.

Client-Dispatcher-Server

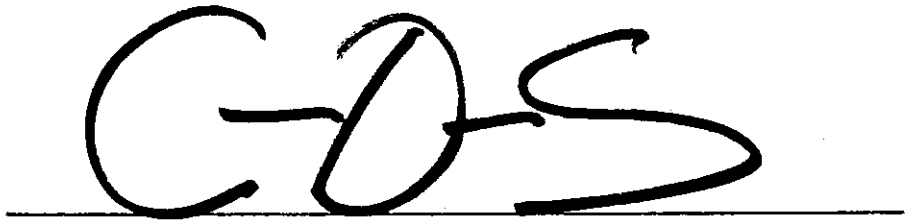
Variants

Distributed Dispatchers. Instead of using a single dispatcher component in a network environment, distributed dispatchers may be introduced. In this variant, when a dispatcher receives a client request for a server on a remote machine, it establishes a connection with the dispatcher on the target node. The remote dispatcher initiates a connection with the requested server and sends the communication channel back to the first dispatcher. The channel is then returned to the client. Another possibility is to allow clients to communicate directly with the dispatcher on the remote machine. This constrains location transparency, however, since clients must know the network node of each server they want to access.

Client-Dispatcher-Server with communication managed by clients. In this variant, instead of establishing a communication channel to servers, a dispatcher may only return the physical server location to the client. It is then the responsibility of the client to manage all communication activities with the server. You can use this variant to increase overall performance, or because the available communication facilities do not require you to establish an explicit communication link.

Client-Dispatcher-Server with heterogeneous communication. It is not always possible to implement the communication between clients and servers using only one communication mechanism. Some servers may use sockets, while others use named pipes. This leads to a variant of the Client-Dispatcher-Server pattern in which the dispatcher is capable of supporting more than one communication mechanism. In this variant, each server register itself with the dispatcher and specifies the communication mechanism it supports. When a client requests a communication channel to a particular server, the dispatcher establishes the communication using to the communication facility the server specified.

Client-Dispatcher-Service. In this variant, clients address *services* and not servers. When the dispatcher receives a request, it looks up which servers provide the specified service in its repository, and establishes a connection to one of these service providers. If it fails to establish the connection, it may try to access another server providing the same service instead, if one is available.



Client-Dispatcher-Server

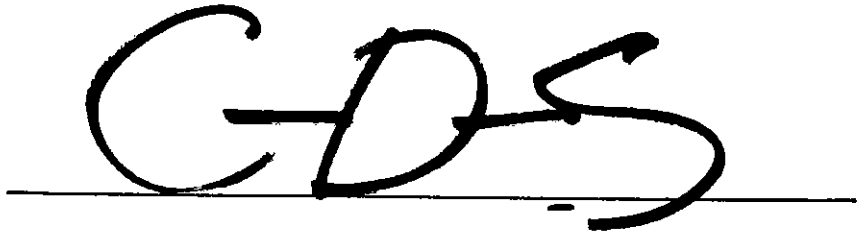
➥ The following sample Java code demonstrates the Client-Dispatcher-Service variant. All clients, servers and the dispatcher exist in the same address space.

The class `Dispatcher` uses a hash table of vectors as a name service repository. An entry in the hash table is available for each service name. Each entry consists of the vector of all servers providing the same kind of service. A server registers with the dispatcher by specifying a service name and the new server instance. When a client asks the dispatcher for a specific service, the dispatcher looks up all available servers in its repository. It randomly selects one of them and returns the server reference to the client.

```
class Dispatcher {
    Hashtable registry = new Hashtable();
    Random rnd = new Random(123456); // for random access

    public void register (String svc, Service obj) {
        Vector v = (Vector) registry.get(svc);
        if (v == null) {
            v = new Vector();
            registry.put(svc, v);
        }
        v.addElement(obj);
    }
    public Service locate(String svc) throws NotFound {
        Vector v = (Vector) registry.get(svc);
        if (v == null) throw new NotFound();
        if (v.size() == 0) throw new NotFound();
        int i = rnd.nextInt() % v.size();
        return (Service) v.elementAt(i);
    }
}
```

The abstract class `Service` represents the available server objects. It registers server objects with the dispatcher automatically when the constructor is executed.

```
abstract class Service {
    String nameOfService; // service name
    String nameOfServer; // server name
    public Service(String svc, String srv) {
        nameOfService = svc;
        nameOfServer  = srv;
        CDS.disp.register(nameOfService,this);
    }
    abstract public void service(); // service provided
}
```



Client-Dispatcher-Server

```
class PrintService extends Service {
    public PrintService(String svc, String srv) {
        super(svc,srv);
    }
    public void service() { // test output
        System.out.println("Service " + nameOfService
                + " by " + nameOfServer);
        // here the service code would be implemented
    }
}
```

Clients ask the dispatcher for object references, then use these references to invoke the appropriate method implementations.

```
class Client {
    public void doTask()
    {   Service s;
        try { s = CDS.disp.locate("printSvc");
              s.service();
        }
        catch (NotFound n) {
              System.out.println("Not available");
        }
        try { s = CDS.disp.locate("printSvc");
              s.service();
        }
        catch (NotFound n) {
              System.out.println("Not available");
        }
        try { s = CDS.disp.locate("drawSvc");
              s.service();
        }
        catch (NotFound n) {
              System.out.println("Not available");
        }
    }
}
```

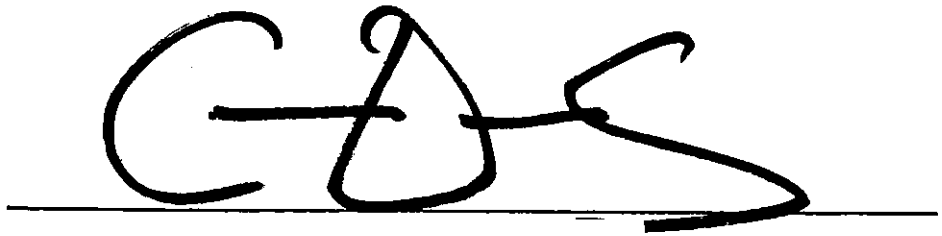
The class `CDS` defines the main program of the application. It instantiates the dispatcher, some servers and a client. It then invokes the event loop of the client:

```
public class CDS {
    public static Dispatcher disp = new Dispatcher();
    public static void main(String args[]) {
        Service s1 = new PrintService("printSvc","srv1");
        Service s2 = new PrintService("printSvc","srv2");
        Client client = new Client();
        client.doTask();
    }
}
```

When the program is started, the following output is displayed:

```
Service printSvc by srv2
Service printSvc by srv1
Not available
```

When the user starts the application, the static method `main` of the class `CDS` is invoked. Two services `S1` and `S2` register with the dispatcher `disp` under the same name. The client is then created and started by calling `client.doTask()`. The client asks the dispatcher to locate the service 'PrintSvc' twice, and once to locate the service 'DrawSvc'. The dispatcher returns the service objects registered with a particular name by using a random number generator. The first service invocations of the client therefore refer to different service objects in the sample output. Since the service 'DrawSvc' is not available, an error occurs when the client asks the dispatcher to locate an appropriate server. ❑



Client-Dispatcher-Server

Consequences

The Client-Dispatcher-Server design pattern has several **benefits**:

Exchangeability of servers. In the Client-Dispatcher-Server design pattern a software developer can change servers or add new ones without modifications to the dispatcher component or the clients becoming necessary. If a new implementation of a server is available, the server first unregisters itself. It then registers itself again with the new implementation.

Location and migration transparency. Clients do not need to know where servers are located—they do not depend on any location information. As a consequence, servers may be dynamically migrated to other machines. This does not work, of course, in the event of the server being migrated while it is connected to a client.

Re-configuration. The developer can defer decisions about which network nodes servers should run until the start-up time of the system, or even to run-time. The Client-Dispatcher-Server design pattern therefore allows you to prepare a software system for later conversion to a distributed system.

Fault tolerance. When network or server failures occur, new servers can be activated at a different network node without any impact to clients. This makes the system more robust and fault-tolerant.

The Client-Dispatcher-Server design pattern imposes some **liabilities**:

Lower efficiency through indirection and explicit connection establishment. The performance of a Client-Dispatcher-Server pattern depends on the overhead introduced by the dispatcher, due to its activities in locating and registering servers and explicitly establishing the connection. The alternative to this approach is to get rid of the dispatcher by hard-coding server locations into the clients. This leads to several disadvantages, however. For example, the clients would then depend directly on the server locations, thus loosing the exchangeability of servers.

Sensitivity to change in the interfaces of the dispatcher component. Because the dispatcher plays the central role, the software system is sensitive to changes in the interface of the dispatcher.

File Service

Schichtenaufbau eines konventionellen File-Systems

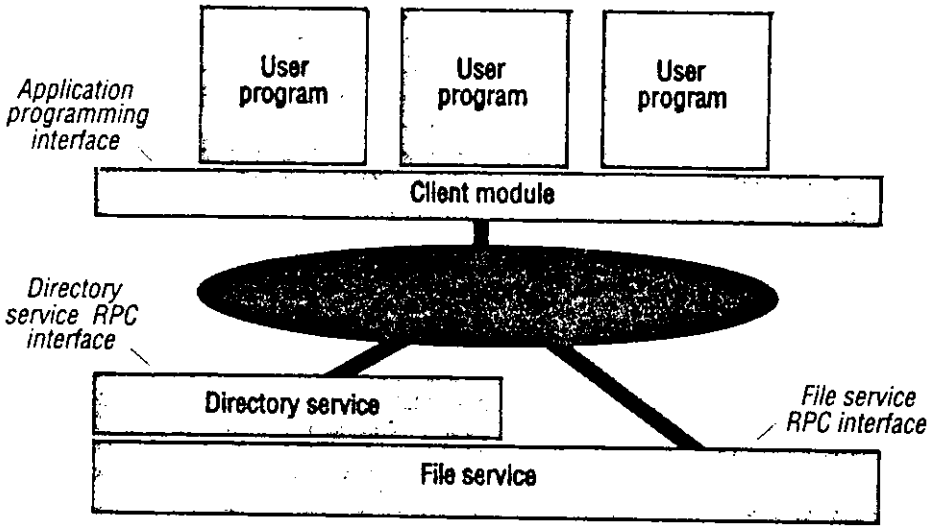
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

Anforderungen an einen verteilten FS

- **Transparenz**
  - Zugriff
  - Ort
  - Parallelität / Konkurrenz
  - Fehler
  - Performance
  - Replikation
  - Migration
- **Offenheit**
  HW + SW - Heterogenität
- **Skalierbarkeit**
  schrittweise Anpassung an Änderungen der Netze, der Benutzerkreises, der Last

File Service

Komponenten



Flat File Service

- Operationen auf File-Inhalten
- global eindeutige File-Bezeichner (UFID, unique file identifiers)
- UFIDs, erzeugt durch CREATE-Op.

Directory Service

- Abbildung zw. File-Namen und UFIDs
- UFID-Eintrag durch Client-Modul
- bei hierarchischen File-Namen: Verweise auf unter/übergeordnete Directories

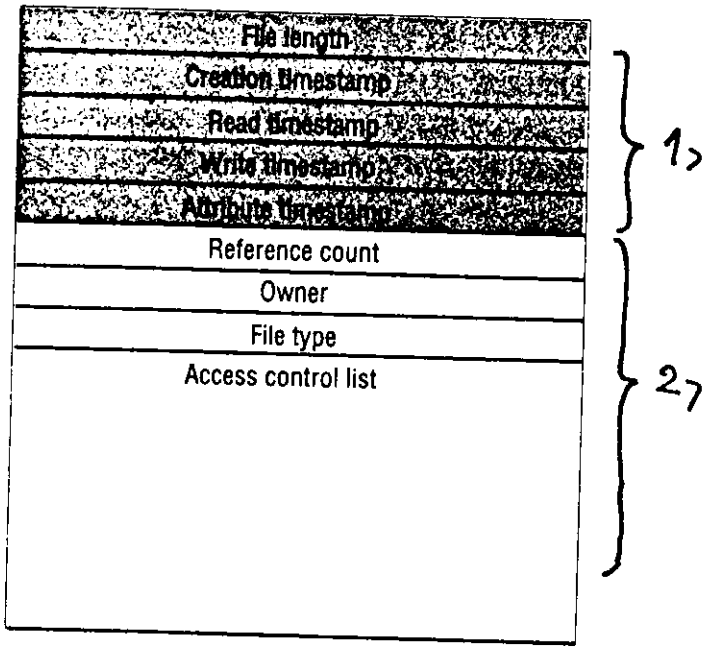
Client Module

- API für Flat File und Directory Service
- Verkapselung von RPC-Aufrufen und UFIDs
- Caching von File-Blöcken
- Emulation von Standard-File-Schnittstellen (zB. UNIX-File-System)

Flat File Service

Merkmale

File-Attribute



- File = Daten + Attribute
- Daten = strukturloser Bytestrom
- keine Open/Close-Operationen
- Read/Write mit Position
- Fehlertoleranz durch
  - a) idempotente Read/Write-Operationen (at-least-once-Semantik)
  - b) zustandsloser Server (Positionszeiger auf Client-Seite)

---

1) verwaltet durch Flat File Service
2) verwaltet durch Directory Service

Flat File Service

RPC - Schnittstelle

Read(File, i, n) → (Data) — REPORTS (BadPosition)
If $1 \leq i \leq Length(File)$:
Reads a sequence of up to *n* items in *File* starting at item *i* and returns it in *Data*.
If $i > Length(File)$:
Returns the empty sequence, reports an error.

Write(File, i, Data) — REPORTS (BadPosition)
If $1 \leq i \leq Length(File)+1$:
Writes a sequence of *Data* to *File*, starting at item *i*, extending the file if necessary.
If $i > Length(File)+1$: null operation, reports an error.

Create() → File
Creates a new file of length 0 and delivers a UFID for it.

Truncate(File, l)
If $l < Length(File)$: shortens the file to length *l*; else does nothing.

Delete(File)
Removes the file from the file store.

GetAttributes(File) → Attr
Returns the file attributes for the file.

SetAttributes(File, Attr)
Sets the file attributes (only those attributes that are not shaded in Figure 7.3).

Directory Service

Merkmale

- Basisoperationen auf linearen, nicht-hierarchischen Directory-Files
- Übersetzung von Namen in UFIDs nach vorheriger Überprüfung der Zugriffsrechte
- Rückgabe der UFIDs zusammen mit verschiedenen "Permissions"
- Übersetzung von Namen in UFIDs = "zustandslose" Ersatzform für Open-Operation
- Integration unterschiedlicher Directory-Dienste (Namensräume, Kontrollregimes)
- Auflösung von hierarchischen Pfadnamen durch iterative Anwendung der Lookup-Operation

Directory Service

RPC-Schnittstelle

Lookup(Dir, Name, AccessMode, UserID) → (File)
— REPORTS (NotFound, NoAccess)
Locates the text name in the directory and returns the relevant UFID; reports an error if it cannot be found or if the client making the request is not authorized to access the file in the manner specified by *AccessMode*.

AddName(Dir, Name, File, UserID) — REPORTS(NameDuplicate)
If *Name* is not in the directory:
Adds the *(Name, File)* pair to the directory and updates the attribute record accordingly.
If *Name* is already in the directory: reports an error.

UnName(Dir, Name) — REPORTS(NotFound)
If *Name* is in the directory:
The entry containing *Name* is removed from the directory.
If *Name* is not in the directory: reports an error.

ReName(Dir, OldName, NewName) — REPORTS(NotFound)
If *Name* is in the directory:
The entry containing *Name* gets the new name.
If *Name* is not in the directory: reports an error.

GetNames(Dir, Pattern) → NameSeq
Returns all of the text names in the directory that match the regular expression given by *Pattern*.

File Service

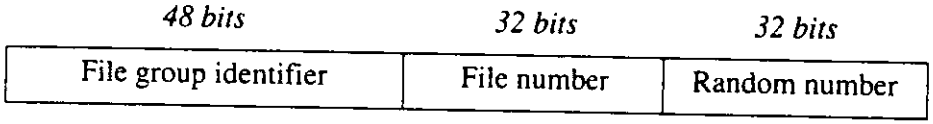
Implementierungs-Techniken

1) File Groups

- Verteilung der Files erfolgt gruppenweise, dadurch vereinfachte Lokalisierung von Files
- UFID enthält global-eindeutigen File Group Identifier

2) UFID-Konstruktion

- Anforderungen: global eindeutig und fälschungssicher



(durch 'random': dünn besetzter UFID-Raum)

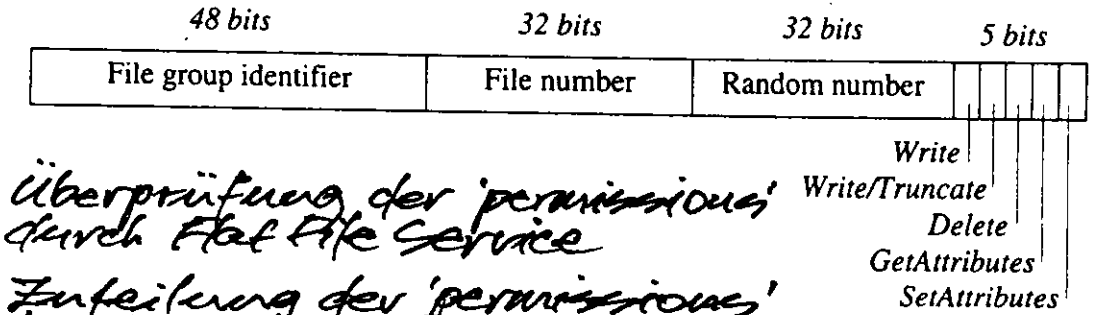
- enthält keine Adressinformation; Umsetzung nötig in Disk-Block-Adressen (durch Flat File Service)

File Service

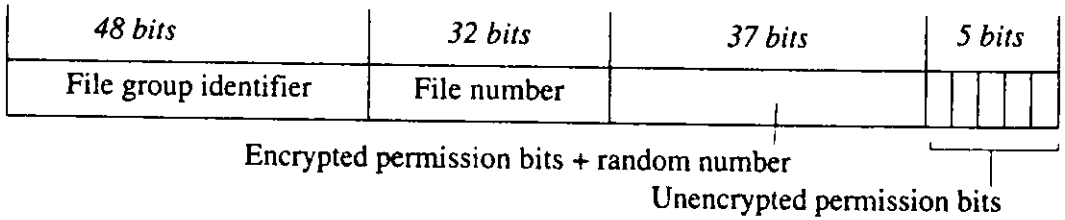
Implementierungs-Techniken

3) Zugriffskontrolle

- UFID ergänzen um 'permission flags'



- Überprüfung der 'permissions' durch Flat File Service
- Zuteilung der 'permissions' bei Directory-Zugriff (access control list)
- Verschlüsselung von 'random' und 'permissions' zum Schutz gegen Änderungen



- Änderung der 'permissions' macht UFID ungültig

File Service

Implementierungs-Techniken

Zu 3) Zugriffskontrolle

- "secret key"-Verschlüsselung zu unsicher
- "Einweg"-Verschlüsselung durch Directory-Service
- "Einweg"-Verschlüsselung der unverschlüsselten "permissions" durch Flat File Service mit anschließendem Vergleich

4) File-Lokalisierung

- im FFS: Übersetzung von UFIDs in Server-Standort (PortId) und File-Adresse
- Tabelle <FileGroupId, PortId> global verfeilt
- zusätzliche FFS-Operation

GetServerPort(FileGroupID) → PortID — REPORTS(UnknownFileGroup)
If *FileGroupID* is known, delivers the port identifier of a server holding the file group, else reports an error.

(mit Caching durch Client-Modul)

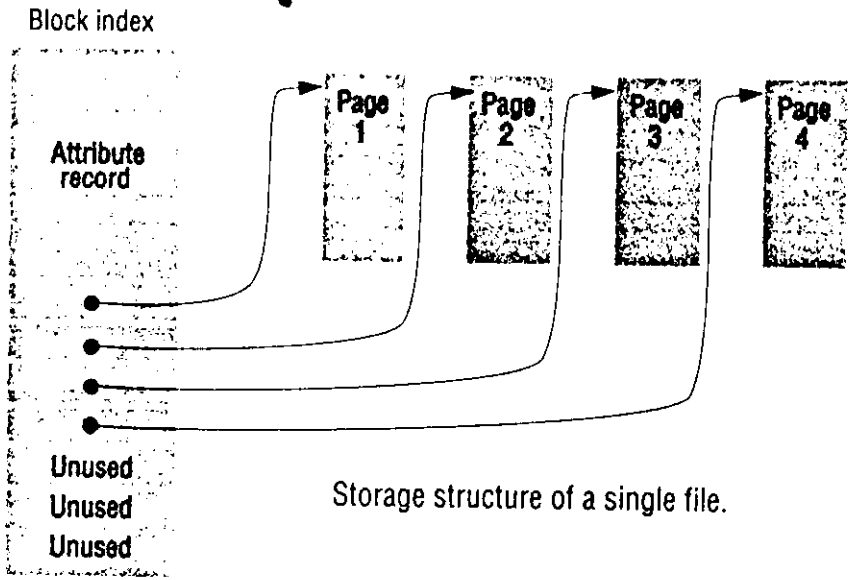
File Service

Implementierungs-Techniken

zu 4) File-Lokalisierung

- Übersetzung von <fileGroupId, PortId> in <Block-Index-Adresse> durch Server

5) File-Repräsentation



Storage structure of a single file.

File Service

Implementierungs-Techniken

Zu 5) File-Repräsentation

- nicht-zusammenhängende Speicherung
- 'Block-Index' verweist auf File-Blöcke
- Manipulation von Blöcken und Block-Index durch 'Block-Modul' (bei Write, Delete, ...)

6) Server Cache

- Caching von häufig adressierten Disk-File-Blöcken im Arbeitsspeicher
- Array aus <Block, Blockzeiger>
- Block-Modul-Operation 'GetBlock': Zugriff zunächst auf Cache; falls erfolglos, Platten-Zugriff über Blockzeiger und Einlagern in Cache
- Block-Modul-Operation 'PutBlock': Konsistenz-erhaltung durch gleichzeitiges Update von Cache und Disk
- Bei Cache-Überlauf: Ersetzungsstrategien (z.B. least-recently-used)

File Service

Implementierungs-Techniken

7) Client Cache

- Server-Cache dient der Server-Performance
- Client-Cache erspart Client-Server-Kommunikation
- Caching für File-Blöcke, File-Attribute und Directory-Information
- implementiert innerhalb des Client-Moduls
- gleichzeitige Updates von Cache-Kopie und Original (wie Server-Cache)
- Transfer von vollständigen File-blöcken trotz Fein-positionierung über Read/Write-Parameter c und n
- **Konsistenz-Problem** bei Mehrfachzugriff auf dasselbe File. Update durch einzelnen Client führt zur Inkonsistenz sämtlicher Cache-Kopien.

→ zentrales Designproblem !

ANDREW

Ein Verteiltes File-System *)

Hauptentwurfsziel

Skalierbarkeit (Unterstützung von bis zu 10 000 Workstations)

Entwurfsstrategie

1) **"whole file serving"** (der gesamte File-Inhalt wird vom Server zum Client übertragen)

2) **"whole file caching"**

- der gesamte File-Inhalt wird im Client-Cache gespeichert
- permanenter Cache auf Platte (überdauert Reboot)
- Reservierung einer gesamten File-Partition (ausreichend für mehrere Hundert Files)

---

*) Bestandteil von OSF/DCE

ANDREW

Benutzer-Profil *)

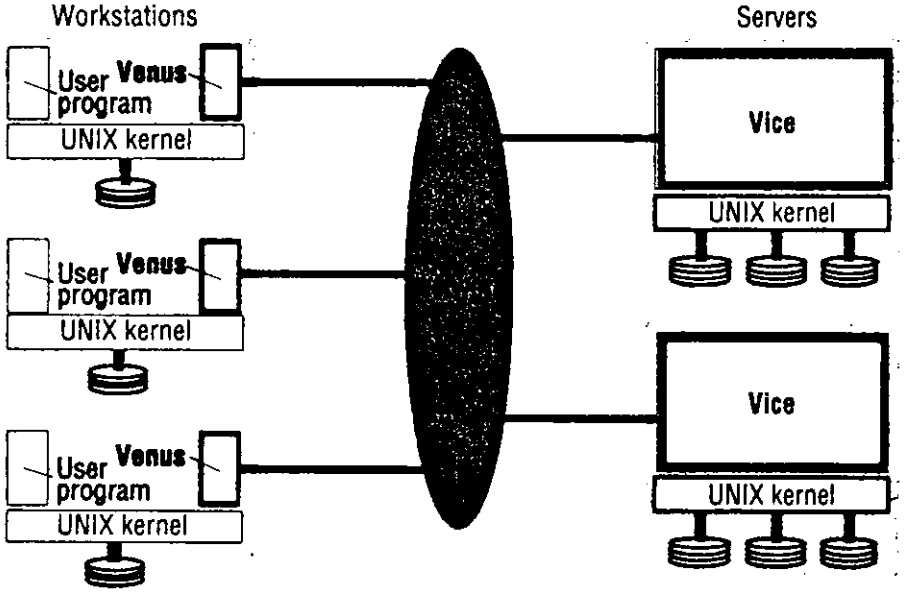
- meist kleine Files (< 10 KB)
- mehr Lese- als Schreiboperationen (Faktor 6)
- meist sequentieller Zugriff
- Schreiben und Lesen meist nur durch einen einzigen Benutzer
- bei konkurrierendem Zugriff erfolgt eine Modifikation nur durch einen einzigen Benutzer
- Filezugriffe erfolgen gehäuft

*) typisch für UNIX-Filesysteme

ANDREW

Implementierung

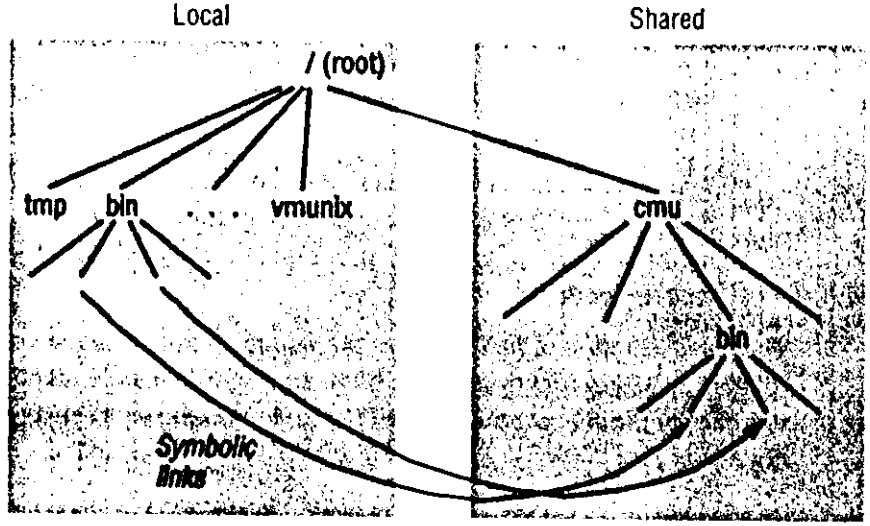
Prozesse



Vice: entspricht Flat File Service

Venus: entspricht Client Module; beinhaltet zusätzlich die Übersetzung von Pfadnamen in AFIDs

Namensraum



- konventionelle UNIX-Directory-Hierarchie mit speziellem Subdirectory "cmu" für sämtliche "shared files".
- Standard Files (in /bin, /lib, ...) existieren lokal nur als symbolische Links auf die "shared" Originale
- temporäre Files und Start-up-Routinen sind "/local"

Cache-Konsistenz

"callback promise"

- Mechanismus zur Sicherstellung von Updates auf "cached copies", nachdem das Original von einem (anderen) Client verändert und geschlossen wurde ("callback").
- Implementiert als Datenstruktur (token), die zusammen mit dem File im Cache gespeichert wird. Zwei Zustände: valid, cancelled
- Bei Öffnen eines Files: Überprüfung der "callback promise". Ggfs. Übertragen einer aktualisierten Cache-Kopie.
- "Cache validation request" nach Client-Reboot, ebenso nach Time-out.

Fetch(fid) → *attr, data*
Returns the attributes (status) and, optionally, the contents of file identified by the *fid* and records a callback promise on it.

Store(fid, attr, data)
Returns the attributes and, optionally, the contents of a specified file and records a callback promise on it.

Create() → *fid*
Creates a new file and records a callback promise on it.

Remove(fid)
Deletes the specified file.

SetLock(fid, mode)
Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.

ReleaseLock(fid)
Unlocks the specified file or directory.

RemoveCallback(fid)
Informs server that a Venus process has flushed a file from its cache.

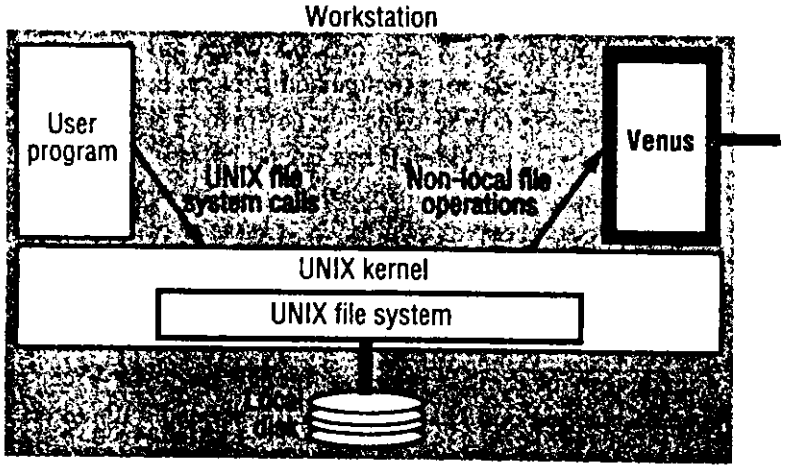
BreakCallback(fid)
This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

Note: Directory and administrative operations (*Rename, Link, Makedir, Removedir, GetTime, CheckToken*, and so on) are not shown.

ANDREW

Implementierung

UNIX-Kernelerweiterung



- **Client-Seite**
  Abfangen von File-Systemcalls durch den Kernel und ggfs. Weiterleiten an Venus
- **Server-Seite**
  File-Identifier ersetzt konventionellen UNIX-File-Deskriptor ("zustandslose" Operationen)

File-Identifier

- Files sind gruppiert in sog. "volumes"
- Alle Files und Directories werden eindeutig identifiziert über 96-Bit-IDs:

32 bits	32 bits	32 bits
Volume number	File handle	Uniquifier

- enthält keine Zugriffskontrollbits (→ Security-System "Kerberos")
- Auflösung der Pfadnamen in FIDs durch iterativen Zugriff auf Vice-Directories

ANDREW

Implementierung der File-System-Calls

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
	Open the local file and return the file descriptor to the application.	Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

Microkernel

The *Microkernel* architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

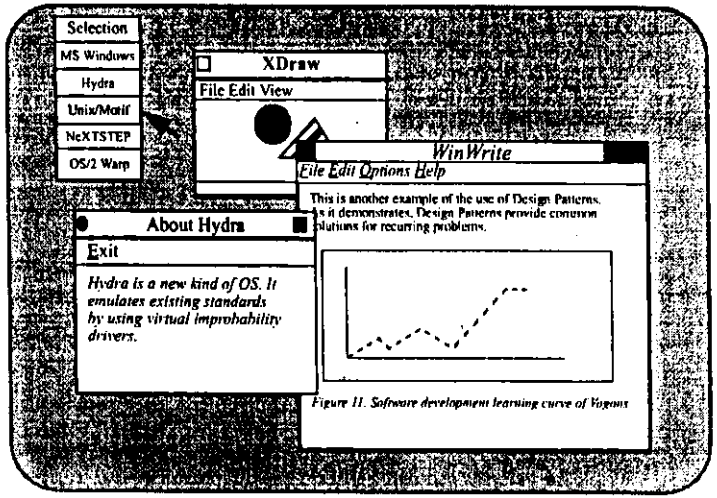


Figure 11. Software development learning curve of Vogons

Example Suppose we intend to develop a new operating system for desktop computers called Hydra. Our development team has elaborated a list of design goals to achieve this. One requirement is that this innovative operating system must be easily portable to the relevant hardware platforms, and must be able to accommodate future developments easily. It must also be able to run applications written for other popular operating systems such as NeXTSTEP, Microsoft Windows and UNIX System V. A user should be able to choose which operating system he wants from a pop-up menu before starting an application. Hydra will display all the applications currently running within its main window:

To emulate all these operating systems, Hydra will integrate special servers that implement specific views of Hydra's functional core. A view denotes a layer of abstraction built on top of the core functionality. The emulation of Microsoft Windows by a server process is an example of such a view.

Since several new technologies such as multimedia, pen-based computing and the World Wide Web are likely to increase in importance, Hydra should be designed for their easy integration, as well as for adaptation, evolution and enhancement of its overall functionality.

Microkernel

Context The development of several applications that use similar programming interfaces that build on the same core functionality.

Problem Developing software for an application domain that needs to cope with a broad spectrum of similar standards and technologies is a non-trivial task. Well-known examples are *application platforms* such as operating systems and graphical user interfaces[20].

The following *forces* therefore need particular consideration when designing such systems:

- The application platform must cope with continuous hardware and software evolution.
- The application platform should be portable, extensible and adaptable to allow easy integration of emerging technologies.
- The applications in your domain need to support different, but similar, application platforms.
- The applications may be categorized into groups that use the same functional core in different ways, requiring the underlying application platform to emulate existing standards.

An application platform that provides the functional core of a domain is an exclusive resource for its clients. To avoid performance problems and to guarantee scaleability, your solution must take an additional force into account:

- The functional core of the application platform should be separated into a component with minimal memory size, and services that consume as little processing power as possible.

Microkernel

Solution Encapsulate the fundamental services of your application platform in a *microkernel* component. The microkernel includes functionality that enables other components running in separate processes to communicate with each other. It is also responsible for maintaining system-wide resources such as files or processes. In addition, it provides interfaces that enable other components to access its functionality.

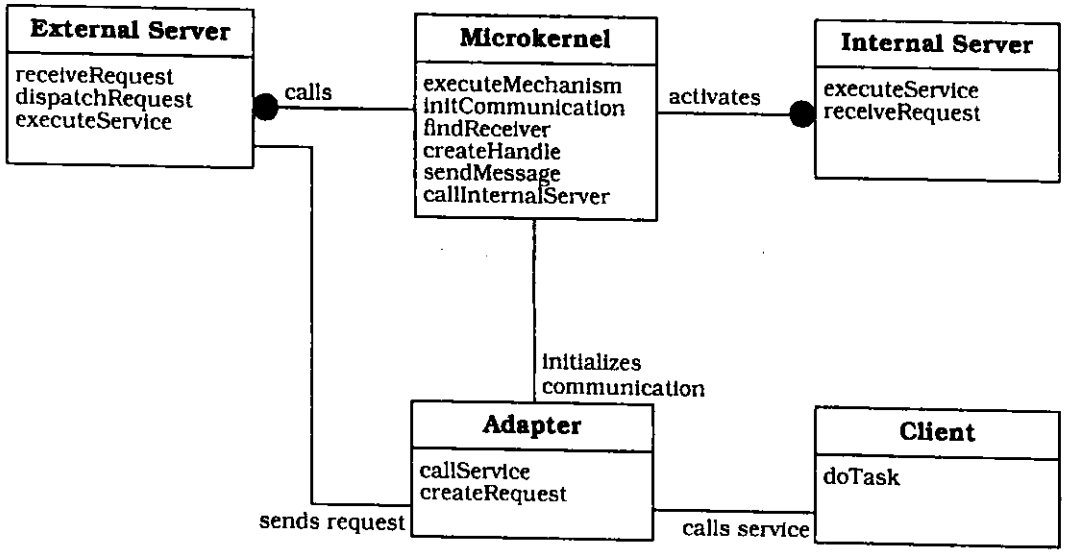
Core functionality that cannot be implemented within the microkernel without unnecessarily increasing its size or complexity should be separated in *internal servers*.

External servers implement their own view of the underlying microkernel. To construct this view, they use the mechanisms available through the interfaces of the microkernel. Every external server is a separate process that itself represents an application platform. Hence, a Microkernel system may be viewed as an application platform that integrates other application platforms.

Clients communicate with external servers by using the communication facilities provided by the microkernel.

Microkernel

Structure The following OMT diagram shows the static structure of a Microkernel system. Its central component, the microkernel, collaborates with external servers, internal servers and adapters. Each client is associated with an adapter used as a bridge between the client and its external server. Internal servers are only accessible by the microkernel component.



Microkernel

Class Microkernel	Collaborators • Internal Server
Responsibility • Provides core mechanisms. • Offers communication facilities. • Encapsulates system dependencies. • Manages and controls resources.	

The *microkernel* represents the main component of the pattern. It implements central services such as communication facilities or resource handling. Other components build on all or some of these basic services. They do this indirectly by using one or more interfaces that comprise the functionality exposed by the microkernel.

Many system-specific dependencies are encapsulated within the microkernel. For example, most of the hardware-dependent parts are hidden from other participants. Clients of the microkernel only see particular views of the underlying application domain and the platform specifics.

The microkernel is also responsible for maintaining system resources such as processes or files. It controls and coordinates the access to these resources.

In summary, a microkernel implements *atomic* services, which we refer to as *mechanisms*. These mechanisms serve as a fundamental base on which more complex functionality, called *policies*, are constructed.

Microkernel

Class Internal Server	***Collaborators*** • Microkernel
Responsibility • Implements additional services. • Encapsulates some system specifics.	

An *internal server*—also known as a *subsystem*—extends the functionality provided by the microkernel. It represents a separate component that offers additional functionality. The microkernel invokes the functionality of internal servers via service requests. Internal servers can therefore encapsulate some dependencies on the underlying hardware or software system. For example, device drivers that support specific graphics cards are good candidates for internal servers.

One of the design goals should be to keep the microkernel as small as possible to reduce memory requirements. Another goal is to provide mechanisms that execute quickly, to reduce service execution time. Additional and more complex services are therefore implemented by internal servers that the microkernel activates or loads only when necessary. You can consider internal servers as extensions of the microkernel. Note that internal servers are only accessible by the microkernel component.

Microkernel

Class External Server	Collaborators • Microkernel
Responsibility • Provides programming interfaces for its clients.	

An *external server*—also known as a *personality*—is a component that uses the microkernel for implementing its own view of the underlying application domain. As already mentioned, a view denotes a layer of abstraction built on top of the atomic services provided by the microkernel. Different external servers implement different policies for specific application domains.

External servers expose their functionality by exporting interfaces in the same way as the microkernel itself does. Each of these external servers runs in a separate process. It receives service requests from client applications using the communication facilities provided by the microkernel, interprets these requests, executes the appropriate services and returns results to its clients. The implementation of services relies on microkernel mechanisms, so external servers need to access the microkernel's programming interfaces.

Microkernel

Class Client	**Collaborators** • Adapter
Responsibility • Represents an application.	

A *client* is an application that is associated with exactly one external server. It only accesses the programming interfaces provided by the external server.

A problem arises if a client needs to access the interfaces of its external server directly. Each client has to use the available communication facilities to interoperate with the external servers.

Every communication with an external server must therefore be hard-coded into the client code. Such a tight coupling between clients and servers, however, leads to various disadvantages:

- Such a system does not support changeability very well.
- If external servers emulate existing application platforms, client applications developed for these platforms will not run without modification.

Microkernel

Class Adapter	***Collaborators*** • External Server • Microkernel
Responsibility • Hides system dependencies such as communication facilities from the client. • Invokes methods of external servers on behalf of clients.	

We therefore introduce interfaces between clients and their external servers to protect clients from direct dependencies. *Adapters*—also known as *emulators*—represent these interfaces between clients and their external servers, and allow clients to access the services of their external server in a portable way. They are part of the client's address space. If the external server implements an existing application platform, the corresponding adapter mimics the programming interfaces of that platform. Clients written for the emulated platform can therefore be compiled and run without modification. Adapters also protect clients from the specific implementation details of the microkernel.

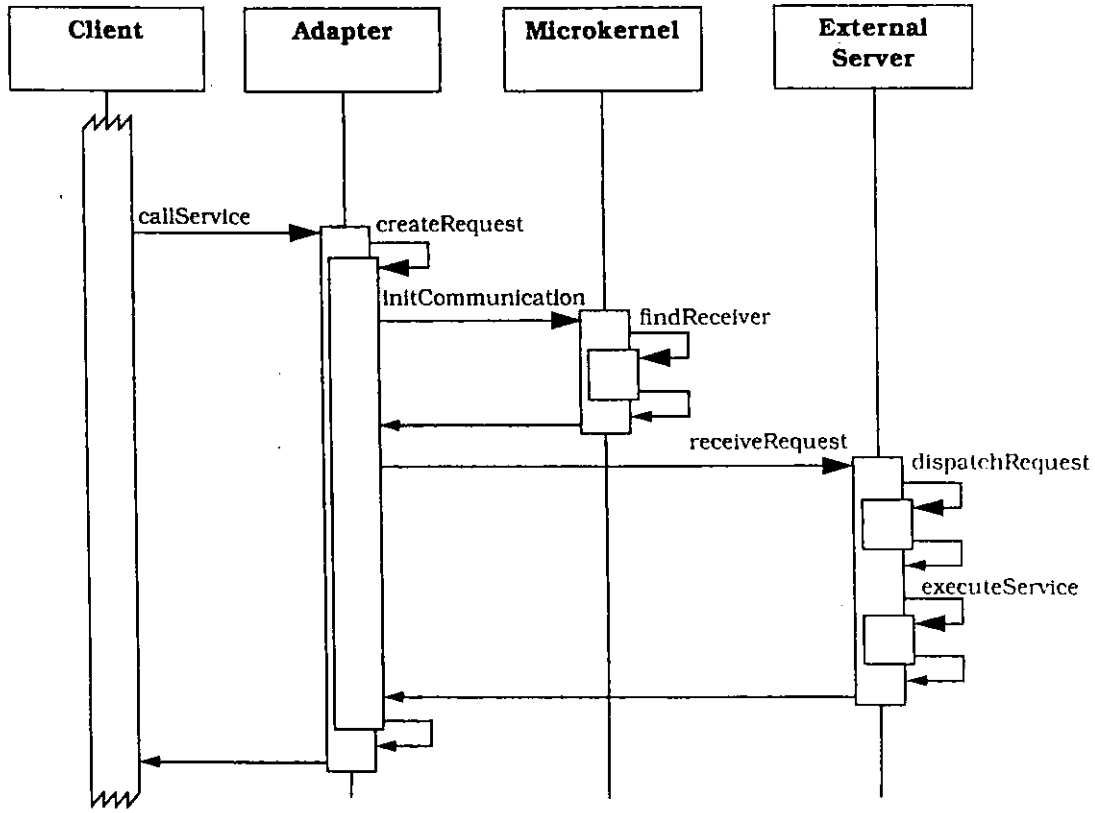
Whenever a client requests a service from an external server, it is the task of the adapter to forward the call to the appropriate server. For this purpose the adapter uses the communication services provided by the microkernel.

Microkernel

Dynamics The dynamic behavior of a Microkernel system depends on the functionality it provides for inter-process communication. In the following scenarios we assume the availability of remote procedure calls. The first scenario also assumes that the external server does not access the microkernel interfaces—this latter case is illustrated in the second scenario.

Scenario I demonstrates the behavior when a client calls a service of its external server:

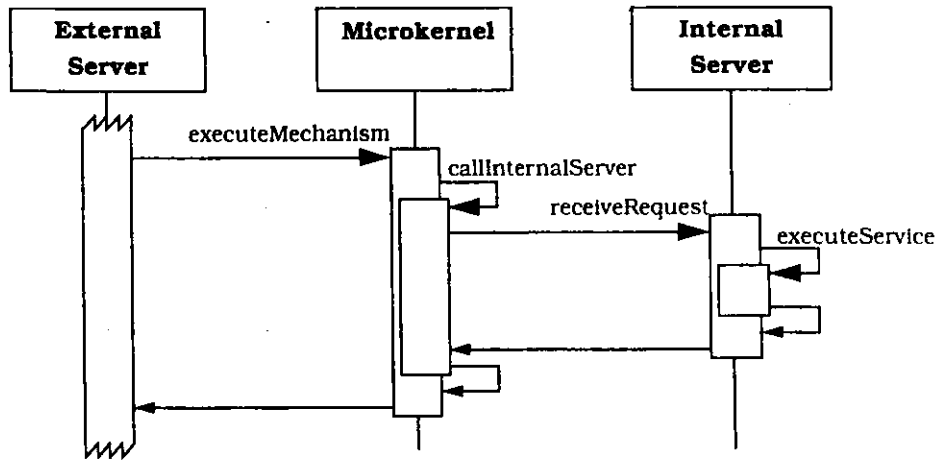
- At a certain point in its control flow the client requests a service from an external server by calling the adapter.
- The adapter constructs a request and asks the microkernel for a communication link with the external server.
- The microkernel determines the physical address of the external server and returns it to the adapter.
- After retrieving this information, the adapter establishes a direct communication link to the external server.
- The adapter sends the request to the external server using a remote procedure call.
- The external server receives the request, unpacks the message and delegates the task to one of its own methods. After completing the requested service, the external server sends all results and status information back to the adapter.
- The adapter returns to the client, which in turn continues with its control flow.



Microkernel

Scenario II illustrates the behavior of a Microkernel architecture when an external server requests a service that is provided by an internal server. In this scenario we assume that the internal server is implemented as a separate process. It could alternatively be implemented as a shared library that is dynamically linked to the microkernel.

- The external server sends a service request to the microkernel.
- A procedure of the programming interface of the microkernel is called to handle the service request. During method execution the microkernel sends a request to an internal server.
- After receiving the request, the internal server executes the requested service and sends all results back to the microkernel.
- The microkernel returns the results back to the external server.
- Finally, the external server retrieves the results and continues with its control flow.



Microkernel

Variants

Microkernel System with indirect Client-Server connections. In this variant, a client that wants to send a request or message to an external server asks the microkernel for a communication channel. After the requested communication path has been established, client and server communicate with each other indirectly using the microkernel as a message backbone. Using this variant leads to an architecture in which all requests pass through the microkernel. You can apply it, for example, when security requirements force the system to control all communication between participants.

Distributed Microkernel System. In this variant a microkernel can also act as a message backbone responsible for sending messages to remote machines or receiving messages from them. Every machine in a distributed system uses its own microkernel implementation. From the user's viewpoint the whole system appears as a single Microkernel system—the distribution remains transparent to the user. A distributed Microkernel system allows you to distribute servers and clients across a network of machines or microprocessors. To achieve this the microkernels in a distributed implementation must include additional services for communicating with each other.

Microkernel

Known Uses

The **Mach** operating system [Tan92] was developed at Carnegie-Mellon-University, and its first version was released in 1986. The Mach microkernel is intended to form a base on which other operating systems can be emulated. One of the commercially-available operating systems that use Mach as its system kernel is NeXTSTEP.

The operating system **Amoeba** [Tan92] consists of two basic elements: the microkernel itself and a collection of servers (subsystems) that are used to implement the majority of Amoeba's functionality. The kernel provides four basic services: the management of processes and threads, the low-level-management of system memory, communication services, both for point-to-point-communication as well as group-communication, and low-level I/O-services. Services not provided by the kernel must be implemented by server processes. This leads to a reduction in kernel size and increases flexibility.

Chorus [Cho90] is a commercially-available Microkernel system that was originally developed by the French research institute INRIA specifically for real-time applications. UNIX System V is available as an external server.

Windows NT [Cus93] was developed by Microsoft as an operating system for high-performance servers. From an architectural point of view Windows NT is definitely a Microkernel system. It offers three external servers, an OS/2 1.x server, a POSIX server and a Win32 server.

The **MKDE** (Microkernel Datenbank Engine) system [Woo96] introduces an architecture for database engines that follows the Microkernel pattern. In this system the microkernel is responsible for providing fundamental services such as physical data access, caching of data and transaction management. Various external servers run on top of the microkernel and provide different conceptual views of the underlying microkernel. A conceptual view denotes a data abstraction according to a given data model, for example the data model of a relational SQL database. Applications such as accounting systems can use the external servers to access databases. MKDE implements the Distributed Microkernel variant to support distributed environments.