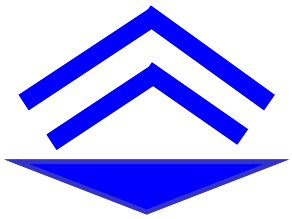
UNIVERSITÄT LEIPZIG

Scriptum zur Lehrveranstaltung

Verteilte Systeme

Komponenten und Aspekte verteilter Systeme, Middleware-Technologien, Verteilungsplattformen, Componentware, Enterprise Computing,
Web Services, moderne Internet-Technologien



Kernfach Praktische Informatik (UL)
Studiengang Informationstechnik (BA)
Studiengang Mobilkommunikation (BA)
Studiengang Praktische Informatik (BA, Bachelor)
Umfang: 2 SWS (15 Wochen)

Prof. Dr.-Ing. habil. Klaus Irmscher
Universität Leipzig
Institut für Informatik
Lehrstuhl Rechnernetze und Verteilte Systeme (em.)

Dresden, den 24. Januar 2013

Gliederung

1 Einführung

1.1 Entwicklung verteilter Systeme (Historie und Trends)

Historische Entwicklung

Bis zur Mitte der 80er Jahre dominierten **zentralisierte Rechner**, die über lokale oder entfernte Terminals genutzt wurden (Datenfernverarbeitung, remote login). Zwei grundlegende Entwicklungen führten zur Abkehr von dieser Richtung: leistungsfähige Arbeitsplatzrechner und eine allseitige Vernetzung derselben über Kabel und Funk.

Es entstanden **Rechnerverbundsysteme**, bestehend aus autonomen Rechnern und Kommunikationssystem (sog. Rechnernetz) zum Austausch von Nachrichten. Wichtige Merkmale dieser Systeme sind die Aufteilung der Ressourcen (Hardware und Software), Verbundfunktionen, wie Nachrichten, Last, Verfügbarkeit und ihr heterogener Aufbau, der zusammen mit einer Ressourcenverwaltung die Basis bildet. Diese Systeme ermöglichen neue Kommunikationsdienste und verteilte Verarbeitung. Klassische Systemunterstützungen für Anwendungslösungen: Netzwerkprogrammierung (z.B. Socket für TCP/IP), Verteilte Betriebssysteme (z.B

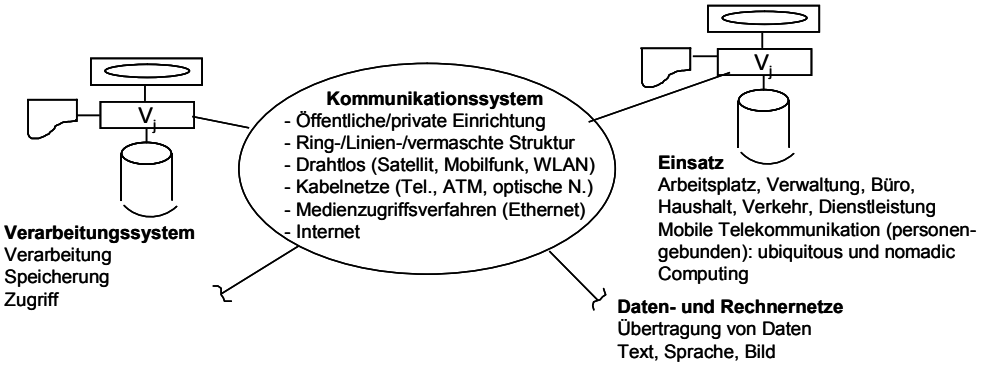


Abbildung 1.1: Verteiltes Rechnersystem / Rechnernetz

Distributed Enterprise Computing

Entwicklung von komplexen Verteilten Systemen

- Verteilte Verarbeitungen ("Verteilte Anwendungssysteme"): markante Beispiele
  - CIM / Fertigungssteuerung (Produktionsplanung, Leitstand, Entwicklung), Grid Computing (z.B. LHC/Cern), Office Computing, PIM, e-Commerce (Banking, Versicherung),
  - World Wide Web (Informationssystem), Web 2.0, Weblogs, Wikipedia, RSS-Feeds, Social Networks (Facebook, Twitter, …),
  - Mobile distributed Computing (Mobile & Nomadic Computing), u.a. PDA, Smartphone, Navigation, ..., LBS, WAP, mobile IP, …, Smartphone-Apps (Online-Messaging…) ...
  - Supercomputing (Wetter, Klima, DNA, ..., Schach),
  - Cloud Computing (Rechen-/Speicherkapazität im Internet), IBM Cebit 2009, Windows 8
  - Verteilte Datenbanken.
- Technologien (Modelle, Kommunikationsformen, Plattformen, Technologien, Dienste)
  - Kooperationsmodelle (Client/Server, P2P, Trader/Broker ...), n-Tier-Architekturen.
  - Programmiermodelle: RPC (prozedural), RMI (objektorientiert, Java), Messaging.
  - Middleware: Verteilungsplattform (DCE, CORBA), Application Server.
    Componentware (J2EE/EJB, .NET/DCOM).
    Dienste-orientiert: Windows Communication Foundation (WCF), Enterprise Application Integration (EAI), Service Oriented Architecture (SOA).
  - Web-Technologien: Web-Services bzw. Lightweight Programming Models (REST, RSS-Feeds), Erweiterung Web zur globalen VPF: Interaktionen, JavaScript, XML, PHP, AJAX (asynchron).

  - Internet-Technologien: Cloud Computing (Azure, W8), Weblications (Web 2.0), Webtop.
  - Datenbanktechnologien (VDBS)
- Neue Dienstleistungen (distributed), neue Formen des Software-Engineering.
  Entwicklung: Prozeduren ~> Objekte ~> Plattformen ~> Komponenten ~> Dienste
  Beispiele: RPC RMI CORBA EJB SOA

Verteilte Systeme integrieren verschiedene Disziplinen, u.a.
- Betriebssysteme (Prozesse, Dateien, Synchronisation, IPC, Sicherheit, ...),
- Rechnerkommunikation (Rechnernetze, Telekommunikation, Kabel / kabellos),
- Parallelverarbeitung, Algorithmentechnik, Datenbanktechnologien,
- Programmierungstechnik,
- Web-Technologien (JavaScript, AJAX, PHP, HTML5, Web-Services),
- Softwaretechnologie (Ressourcenverteilung, Arbeitsorganisation, Management),
- Internet-Technologien (z.B. Cloud Computing).

Entwicklung in verschiedenen Etappen:
1. Klassische verteilte Systeme (2. Hälfte 20. Jahrh.)
   - Erweiterung des Betriebssystem, Ressourcenverwaltung, Synchronisation, Koordinierung, Namensverwaltung udgl.
   - Synchrone/asynchrone Kommunikationsformen (Programmierung RPC, RMI, JMS)
   - Literatur: Couloris/Dollimore/Kindberg, Tanenbaum/Steen
2. Verteilungsplattformen (ab 1989/1995)
   - Middleware (DCE, CORBA), Application Server,
   - Componentware (EJB, .NET), Dienstearchitektur (SOA)
3. Entwicklung von Web-Technologien als globale Verteilungsplattform (ab 2000/2004)
   - Interaktionen (PHP), Datenstrukturen (XML), HTML5, JavaScript, AJAX (asynchrone DÜ zwischen Client und Server, XMLHttpRequest-Objekt).
   - Einsatz Web-Services (SOAP, WSDL). Web-Applikationen, u.a. Web 2.0.
   - Speicher- und Verarbeitungskapazität im Web (Amazon S3, EC2; MS Azure) Zugang über Web (Browser) -> Cloud Computing.

1.2 Middleware: Konzepte und Plattformen

Verteiltes System

George Coulouris: „Ein verteiltes System (VS) ist ein System, in dem sich HW- und SW-Komponenten auf vernetzten Computern befinden und miteinander über den Austausch von Nachrichten kommunizieren“.

Kommunikation auf Basis eines gemeinsamen Protokollstacks ~> regelt Ablauf und Form des Informationsaustausches („Rechnernetz“). Bekannte Protokolle: TCP/IP, X.25, ISDN.

Computer-intern: IPC (Interprocess Communication). Bekanntes einfaches VS (Rechnernetz): Internet.

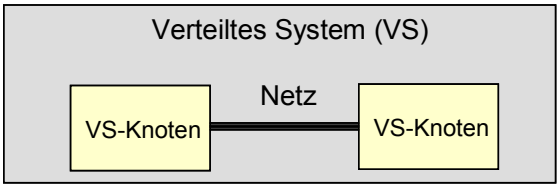


Abbildung 1.2: Einfaches verteiltes System (VS)

Begriffe Verteiltes System (VS) und Rechnernetz (RN) oft synonym verwendet. RN ist ein VS im engeren Sinn: RN ist Transporteinrichtung, im erweiterten VS können bereits anwendungsorientierte Dienste integriert sein, z.B. Namensverwaltung, Sicherheit, usw.

Verteilte Anwendungen (VA)
Verteilung der Anwendungslogik auf mehrere, voneinander weitgehend unabhängige Anwendungskomponenten, die auf separaten Knoten (Rechner) eines VS/RN installiert sind.

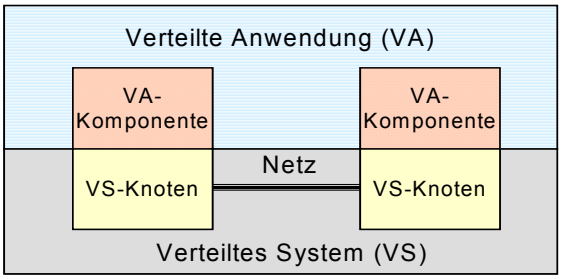


Abbildung 1.3: Verteiltes Anwendungssystem (VA)

VS („Rechnernetz“) dient als Kommunikationsinfrastruktur. Verteilte Anwendungen, u.a.:
- Einfache verteilte Anwendungen: Internet-Anwendungen (WWW, FTP, Email) bis komplexe Anwendungen (Grid Computing, Bürokommunikation, CAD, Wettersimulation ...),
- Verteilte Informationssysteme: Banking, Industrie, Versicherung, Flugwesen (Check-in),
- Eingebettete verteilte Systeme: Automobilindustrie, Fertigungssteuerung, Haushalt,
- Mobile verteilte Systeme: portable Geräte, Mobilfunk, ubiquitous / nomadic computing.

Middleware
Einfache verteile Systeme (RN) bieten i.w. nur Kommunikationsdienste (Verbindungsauf/abbau, Datenübertragung im Byteformat) und einfache Sicherungs- und Fehlerbehandlungsmaßnahmen. Setzt verteilte Anwendung direkt auf dem Protokollstack des VS auf ~> **Netzwerkprogrammierung** (z.B. **Sockets**): hohe Performance, gute Steuerbarkeit, aber sehr aufwandsintensiv. Effektiv nur für kleinere Anwendungen geeignet.
Deshalb: Einführung einer zusätzlichen Softwareschicht, der sog. **Middleware**. Beschreibt das (logische) Interaktionsmuster zwischen den Partnern im Verteilten System. Unterstützt asynchrone und synchrone Kommunikationsformen (im Detail siehe Kap. 2).

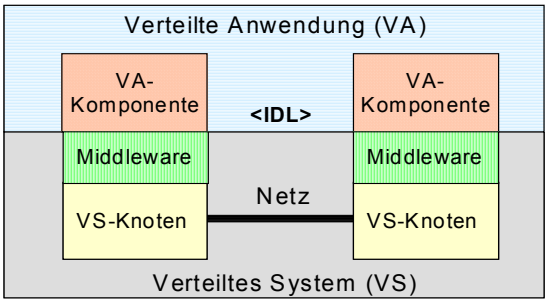


Abbildung 1.4: Middleware-basiertes verteiltes Anwendungssystem

Middleware bietet für die Anwendung eine intelligente Aufrufschnittstelle (IDL: Interface Definition Language), Funktionalitäten, die die Anwendungsentwicklung entlasten und verborgen bleiben, wie
- Namensverwaltung & Verzeichnisdienst, Zeitdienst,
- Sicherheit, Management, Nebenläufigkeit, Datenverwaltung,

sowie Lösungen für heterogene HW/SW-Plattformen.
Middleware ist eine Erweiterung des Betriebssystems, aber im Gegensatz zu verteilten Betriebssystemen als eigenständige Schicht auf das BS aufgesetzt und damit flexibler.
***Kategorien**:*
- **Kommunikationsorientierte Middleware (network awared):**
  Erweiterung der Netzverbindung durch Dienste und Laufzeitaspekte zur Unterstützung von VA, u.a. Threads, Datenanpassungen (Marshalling, Formate), Fehlerbehandlung.

Bekannte Programmiermodelle:

RPC (prozedural): entfernter Prozeduraufruf (Remote Procedure Call), synchrone Kommunikation.

RMI (objektorientiert, Java): entfernter Methodenaufruf (Remote Method Invocation), synchrone Kommunikation.

Nachrichtenorientiertes Modell (Message Passing, Message Queueing), asynchrone Kommunikation, Warteschlangen.

- **Anwendungsorientierte Middleware (content awared):**

Erweiterte Laufzeitumgebung und zusätzliche Dienste, wie Persistenz, Sicherheit, Transaktion, Ressourcenverwaltung. Bereitstellung als

Verteilungsplattform:

DCE (prozedural): Distributed Computing Environment, OSF (Teil der Open Group).

CORBA (objektorientiert): Common Object Request Broker Architecture, OMG.

Componentware (mit Komponentenmodell):

Anwendungsobjekte werden als Komponenten definiert (Schnittstellen) und für eine Komponenten-Laufzeitumgebung bereitgestellt, u.a. ***JavaEE (EJB), .NET (DCOM).***

Dienste-orientierte Architektur: ***SOA*** (Service oriented Architecture), ***.NET/WCF***:

Funktionalitäten in Form von Diensten zur Realisierung verteilter Anwendungen. SW-Komponenten zur Entwicklung und Gestaltung von Diensten und Schnittstellen.

Zusätzliche Entwicklungen (Einsatz als Middleware oder eigenständige Technologie)

- **Web-Services:**

Auf Web-Servern verfügbare SW-Komponenten zur Gestaltung verteilter Anwendungen (Aufrufprotokoll SOAP, Schnittstellenbeschreibung WSDL, Verzeichnisdienst UDDI).

Technologisch eher den entfernten Aufrufen zuzuorden: über Protokoll SOAP (Simple Object Access Protocol, Basis XML) werden Web-basierte entfernte Aufrufe definiert.

- **Internet-Technologien:**

Neue Web-Technologien: Entwicklung Web zu einer globalen Verteilungsplattform.
Einsatz von Web-Services, XML, HTML5, PHP, JavaScript, AJAX, ...
Unterstützung von Web-Anwendungen, u.a. bei Google, Amazon und in Web 2.0.

Cloud Computing: Verlagerung Verarbeitung und Speicherung ins Internet („Wolke“), Zugang über Web-Browser. Unterstützung durch neue Betriebssysteme (Azure, Windows 8)

Plattformen (Technologien für anwendungsorientierte Middleware)

Object Request Broker (ORB): Aufsetzend auf dem Programmiermodell für entfernte Methodenaufrufe (RMI-IIOP). Bietet vielfältige Dienste an, i.d.R. ohne eigene Laufzeitumgebung. Bekannter Standard: CORBA (Common Object Request Broker Architecture).

Application Server: Bieten Kommunikationsinfrastruktur, Dienste, Laufzeitumgebung und Unterstützungen für die Anwendungslogik (Middle-Tier).

Componentware (Komponentenbasierte Middleware-Plattformen):

Erweiterung der Application Server zu einer vollständigen Verteilungsplattform mit zugehörigen Komponentenmodellen. Bekannteste Entwicklungsumgebungen:

- Java 2 Platform Enterprise Edition (J2EE), Komp.-Modell EJB (Enterprise JavaBean).
- .Net-Plattform mit Komponentenmodell COM bzw. DCOM (bzw. Dienstemodell WCF).
- CORBA 3.0 mit Komponentenmodell CCM.

Integrierte Architekturen für Geschäftsprozess

Zielstellungen: Unterstützung für heterogene verteilte Systeme von Geschäftsprozessen (business objects), Anpassbarkeit (time-to market), Wiederverwendbarkeit.

Probleme bei Integration heterogener Systeme: Interoperabilität ~> Schlüssel: EAI, SOA.

• Enterprise Application Integration (EAI):

Vorstufe zu SOA. Mischung aus Konzepten, Technologien und Werkzeugen. Im Gegensatz zu SOA noch stärker auf die technische Struktur orientiert. EAI fokussiert die Integration eigenständiger Anwendungen, Middleware ist mehr auf Kommunikation zwischen Anwendungskomponenten ausgerichtet.

- Service Oriented Architecture (SOA):
  Dienste-orientierte Architektur (seit 2005) nach Konzept der SW-Komponenten zur Entwicklung verteilter, heterogener Geschäftsprozesse ~> Gestaltung von Diensten und Schnittstellen: Funktionalitäten in Form von Diensten.

Entwicklung von Web-Technologien als globale Verteilungsplattform

- Nutzung der verteilten Struktur im Web (Internet, Adressierung, Server).
- Interaktionen (PHP), Datenstrukturen (XML), HTML5, JavaScript, AJAX (asynchrone Datenübertragung zw. Client und Server, Nutzung JavaScript und XMLHttpRequest-Objekt).
- Alternativen zu AJAX: Mono (.NET), Gecko-Engine (Mozilla).
- Einsatz von Web-Services (SOAP, WSDL).
- Entwicklung von Web-Applikationen (Weblications), u.a. Web 2.0, Semantic Web;
  Einsatz als social Web, u.a. YouTube, Flickr, Wiki, Blog (Read-Write-Web).

Ergänzung: *Neue Internet-Technologien*

- Verlagerung Speicherung/Verarbeitung ins Internet, PC nur als Terminal („Webtop“).
  Speicherplatz und CPU-Leistung (Google, Amazon S3 und EC2, Windows Azure).
- Cloud Computing (IBM, Cebit 2009).

1.3 Spezifikation offener verteilter Systeme

VS als Verbund autonomer Systeme

- Koordinierte Kooperation verteilter SW-Komponenten für gemeinsame Anwendung (Kooperation: Zusammenspiel mehrerer Komponenten, Koordination: geordneter Ablauf des Zusammenspiels).
- Programmiermodelle (RPC, RMI ...), standardisierte Verteilungsplattformen (CORBA ...), Komponentenmodelle (J2EE, .NET), Dienste-orientierte Modelle (WCF, SOA).
- Neue Web- und Internet-Technologien: Web-Services, JavaScript, AJAX, Cloud Comp.

Unterstützungen zur Spezifikation (~> Beschreibung, Modellierung & Bewertung „MMB“)

- Komponenten verteilter Systeme (Aufteilung der Funktionen)
- Kooperationsmodelle (Zusammenarbeit der Komponenten bzw. Dienste)
- n-Tier-Architekturen (Verteilung der Komponenten).

Komponenten verteilter Systeme

In verteilten Systemen treten Probleme und Effekte auf, die ein autonomes System nicht aufweist. Nachrichten müssen zum Beispiel über unsichere Netzwerke transportiert werden, wobei sie verloren gehen können und ihre Laufzeit schwer oder gar nicht vorhersagbar ist.

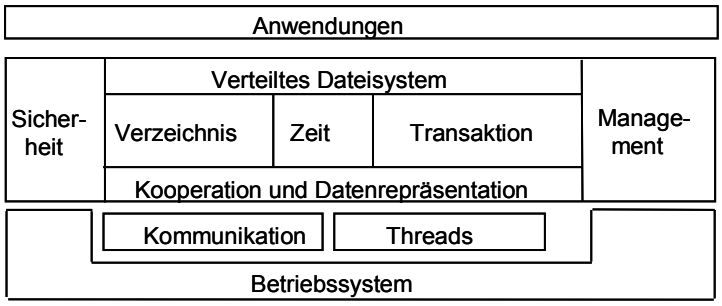


Abbildung 1.5: Betriebssystemerweiterung

Funktionen in verteilter Struktur (als verteiltes BS oder Middleware (Verteilungsplattform)):

- Kommunikation und Nebenläufigkeit: Kommunikationsprotokolle, TCP/IP, Übergänge (Gateways): Threads
- Kooperation: Entfernter Prozeduraufruf (RPC), Methodenaufruf (RMI), Message Passing
- Namensverwaltung und Verzeichnis: Objektidentifikation, Namensraum, Verzeichnisdienst: X.500, LDAP
- Sicherheit: Autorisierung, Authentisierung, Integrität, Zugriffsrecht
- Verteiltes Dateisystem: Transparente Erweiterung lokaler Dateisysteme; Beispiele: NFS, AFS, DFS
- Zeit (Synchronisation und Koordination): Uhrensynchronisation, NTP, Election, Deadlock-Behandlung, ...
- Transaktionen: Sicherung Konkurrenz und Nebenläufigkeit, Datenkonsistenz, Persistenz
- Management: Fehler, Konfiguration, Abrechnung, Leistung, Sicherheit

Kooperationsmodelle in verteilten Systemen

Modelle zur Kooperation zwischen SW-Komponenten in heterogenen verteilten Systemen
- Beschreibung der Kooperation (Zusammenarbeit) von heterogenen SW-Komponenten,
- Bindeprozess, Choreografie der Komponenten/Dienste (Ergänzung durch n-Tier-Achitekturen).

Bekannte Kooperationsmodelle: Client/Server-Modell, Erweitertes Client/Server-Modell (mit Trader), Peer-to-Peer (P2P), Producer/Consumer-Modell, Gruppen-Modell, Aktoren-Modell.

Das **Client/Server-Modell** (seit ca. 1984) basiert auf der Trennung von Diensterbringer (**Server**) und Dienstnutzer (**Client**). Dienste werden aktiv vom Client über die Service-Schnittstelle des Servers angefordert (Rollenwechsel möglich).
Bekannte Anwendung: WWW. Eine Erweiterung dieses Modells vermittelt Dienste mit Hilfe so genannter **Trader**, wobei ganze Traderkonfigurationen und –föderationen aufgebaut werden, u.a. Online-Dienste wie Homebanking und e-Payment oder allgemein bei Service-on-Demand.

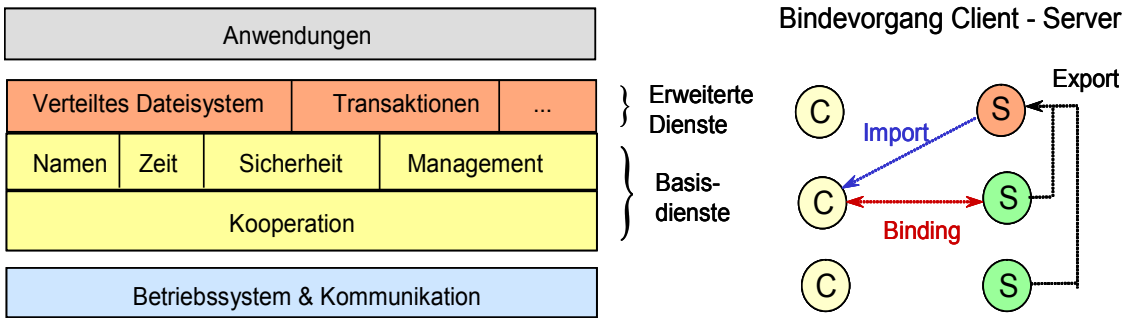


Abbildung 1.6: Client-Server-System und Bindevorgang

Bekannt seit ARPAnet (1969) ist auch das **Peer-to-Peer-Modell**, bei dem beide Partner gleichberechtigt interagieren und eine dezentrale Diensterbringung ermöglichen. Charakteristisch sind dezentrale Diensterbringung, Funktionsverteilung, Shared Filesystems.

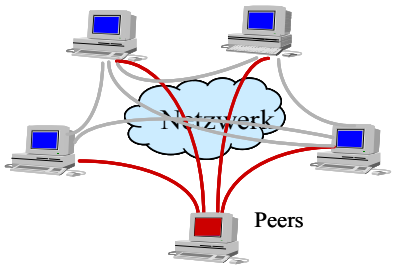


Abbildung 1.7: Peer-to-Peer-System

P2P-Modelle arbeiten mit gleichberechtigten Prozessen. *Bindevorgang:* dezentrales Suchen (ad-hoc-Routing, z.B Flooding), direktes Anbinden. Vorteil: direkter Datenaustausch, Entlastung der hohen Serverbelastung. Nachteil: keine zentralen Dienstleistungen (Routing, Verzeichnisse udgl.). Voraussetzungen: erhöhte Leistungen im Zugangsnetz-Bereich (z.B. xDSL) und kostengünstige Abrechnungsmodelle (z.B. flat rate).Bekannte Lösungen: Musiktauschbörse Gnutella, [Napster]; File-Sharing KaZa. Günstige Unterstützung für Ad-hoc-Netzwerke. Alternativen bietet das **Produzent/Verbraucher-Modell**, bei dem ein Produzent aktiv ist und die von ihm produzierten Informationen ablegt, worauf sie früher oder später von einem Verbraucher beansprucht werden. Auch hier ist manchmal eine Abstimmung notwendig.
Beim **Gruppen-Modell** arbeiteten mehrere Parteien an einem Gegenstand, um ihn in Richtung eines gemeinsamen Zieles zu verändern. Dies geschieht interaktiv.
Im **Aktoren-Modell** ist es auch möglich, anstehende Berechnungen jeweils der nächsten freien Ressource zuzuteilen, die diese durchführen kann.

n-Tier-Architekturen

Problemstellung

Verteilung einer verteilten Anwendung auf die einzelnen Knoten, **ohne** Verluste in der Leistung und erhöhte Sicherheitsrisiken gegenüber einer zentralisierten Lösung. Insbesondere

- Anzahl der Anwendungskomponenten innerhalb der Anwendung,
- Verteilung der Aufgaben auf die Anwendungskomponenten,
- Verteilung der Anwendungskomponenten auf die einzelnen Knoten.

n-Tier-Architekturen

Ergänzung zum Client/Server-Modell und beschreiben die Verteilung einer Anwendung auf die einzelnen Knoten. Grundlage einer n-Tier-Architektur: Tier (engl.: Schicht, Stufe). Im C/S-Modell bezeichnet ein Tier einen eigenständigen Prozessraum innerhalb einer Anwendung. Tier-Grenzen: an Rechnergrenzen bzw. lokal an Grenze des Prozessraums.
In der Regel 2-, 3,- 4-Tier-Architekturen. Verwendung der n-Tier-Architekturen vor allem bei Informationssystemen, DB-Anwendungen, Grid-Computing und Web-Applikationen.
Zentrale Problemstellung (insbes. in verteilten Informationssystemen):

- Präsentation: Bereitstellung einer interaktiven Benutzerschnittstelle zur Anwendung.
- Anwendungslogik: enthält die eigentliche Funktionalität einer Anwendung.
- Datenhaltung: permanente Speicherung der Daten und Sicherung der Datenkonsistenz.

n-Tier-Architektur muss die geeignete Zuordnung der Aufgaben zu den Anwendungskomponenten sowie die Verteilung der Anwendungskomponenten auf die Tiers festlegen.

2-Tier-Architektur

Besteht aus einer Client-Tier (Präsentationskomponente der Anwendung) und einer Server-Tier (Datenhaltungskomponente).
Keine eigene Tier für Anwendungslogik (individuell auf Client- und Server-Tier verteilt).

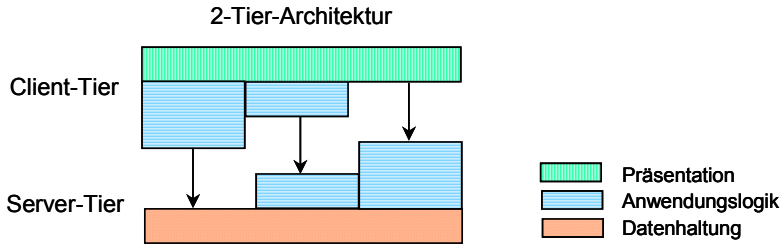


Abbildung 1.8: 2-Tier-Architektur

Beschreibt einfaches Client/Server-Verteilungsmodell. Dabei 2 Technologien entwickelt:
4GL (Fourth Generation Language):

- Programmiersprachen, für Entwicklung simpler datenzentrierter 2-Tier-Anwendungen.
- Automatische Entwicklung der Benutzeroberfläche und Sprachmittel für DB-Zugriff.

Stored Procedures
- Erweiterung relationaler Datenbanken, z.B. SQL-Anfragen in vordefinierten Proceduren.

2-Tier-Architekturen leicht programmierbar, aber schnell unübersichtlich, unstrukturiert und schwierig skalierbar ~> heute weniger angewandt.

3-Tier-Architektur

Erweiterung durch weitere Tier, sog. Middle-Tier. Aufgaben einer verteilten Anwendung auf die einzelnen Tiers verteilt: *Client-Tier*: Präsentation; *Middle-Tier*: Anwendungslogik; *Server-Tier*: Datenhaltung. Vorteile der 3-Tier-Architektur: zentrale Administration der Anwendungslogik, gute Skalierbarkeit.

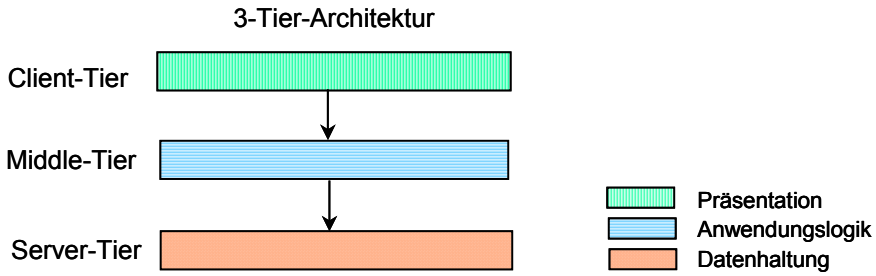


Abbildung 1.9: 3-Tier-Architektur

Anwendung: einfache Internet-Anwendungen, verteilt auf Browser, Web-Server und DB.

4- und mehr-Tier-Anwendungen:

Analog wie 3-Tier. Dabei Anwendungslogik auf zusätzliche Tiers verteilt. Weiterhin: Thin-Client- bzw. Fat-Client-Architekturen. Thin ("dünn"): Client-Tier ausschließlich Präsentationsaufgaben. Fat (dick"): Client auch Aufgaben der Anwendungslogik.

1.4 Verteilte Anwendungssysteme (Beispiele)

CIM: Computer Integrated Manufacturing

Beispiel: Komplexes System der Fertigungssteuerung (Leitstand, PPS, Entwicklung (CAD)); Verteilte Datenbanken, Büroautomatisierung.

LHC Computing Grid (LCG)

Teilchenbeschleuniger LHC (Large Hadron Collidier): Cern/CH. Unterirdischer Ring, 26 km. Globaler „Grid". Kollision von Protonen und Bleikernen, Analyse in 4 unterirdischen Teilchendetektoren, Frage nach Ursprung der Materie und Urknall (sog. Higgs-Boson, „Gottes-Teilchen"). Voluminöse Datenmodelle, 12 Petabyte p.a., 6000 Wissenschaftler, 50 Länder. Weltumspannendes RN, mehrere Ebenen (Tiers), unterschiedlicher RZ.

EDI: Electronic Data Interchange

Austausch elektronischer Dokumente (Geschäftsbereich, Büro) über das Netz. Standardisierung (Nachrichtenformat, Dokumentenaustausch): EDIFACT ~> EBxml (XML)

WWW: World Wide Web

Hypertext-basiertes Informationssystem auf der Basis einer Client/Server-Architektur. Nutzung des Internet, Protokollstack HTTP, TCP, IP (~> Script IntW3).

2 Middleware-Technologien

2.1 Betriebssystem und Middleware

Betriebssystem
Erweiterung des lokalen Betriebssystems um Funktionalitäten zur Realisierung verteilter Anwendungen und zur Verwaltung der Ressourcen (Verteiltes BS vs. Middleware & SWT).
Komponenten s. Kap.1, Abbildung „Betriebssystemerweiterung".
Merkmale, u.a. territorialen Verteilung, Verbundfunktionen, Nachrichten zw. den Knoten.

Middleware
Zwischen Anwendung und Basissystem (lokales Betriebssystem, Kommunikationssystem). Schnittstellen: API und System-Interface.
Bereitstellung von Verteilungsfunktionen und Diensten zur Anwendungsunterstützung. Entlastung Anwendungs-SW und Entwicklungsprozess (time-to-market).

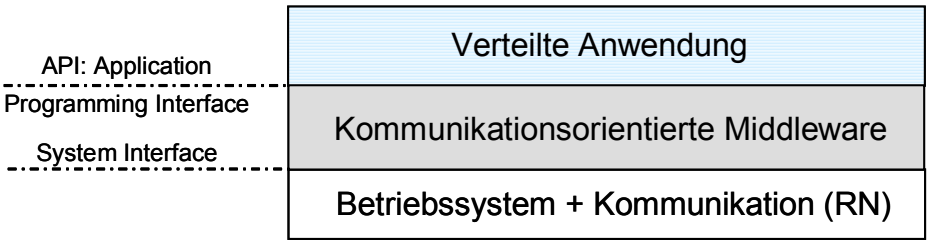


Abbildung 2.1: Einordnung Middleware

Untergliederung:
- Kommunikationsorientierte Middleware (network awared): Bereitstellung einer logischen Kommunikationsinfrastruktur: Zugriff, Datenformatierung, Fehlerbehandlung.
- Anwendungsorientierte Middleware (context awared): Erweiterung der kommunikationsorientierten Middleware um Dienste und Laufzeitaspekte zur Unterstützung der Anwendung.

2.2 Kommunikationsorientierte Middleware (network awared)

Aufgaben
Unterstützung der Kooperation und Übertragung zwischen entfernten Komponenten
- Entfernte Aufrufe (Prozeduren, Methoden), Messaging, Übertragungssteuerung.
- Zusammenwirkung mit der Prozessverwaltung (Threads).
- Steuerung des Kommunikationsflusses, Verwaltung der parallelen Aufrufe.

~> Realisierung durch MW-Protokoll, aufsetzend auf Transportprotokollen (IP, TCP/UDP).
Datentransformation (Marshalling, Heterogenität, Datenformate).
Fehlerbehandlung:
- Übertragungsfehler: Zerstörung/Verfälschung bzw. Verlust von Bits bzw. Paketen.
  ~> Prüfsummenchecks (Bitübertragungs-/DLL-Schicht, z.B. CRC), Wiederholung / FEC.
- Ausfall von Komponenten: Fehlerbehandlung (Erkennung, Behebung) verlagert auf höhere Protokollebenen ~> Verschiedene Fehlerparadigmen, u.a. Ignorieren des Fehlers, Sendewiederholung, Replikate (Verfügbarkeitsverbund), Terminierung.

2.2.1 Datentransformation (Marshalling, Datenformate)

Marshalling und Heterogenität
Marshalling / Unmarshalling
Transformation der Daten (Aufrufparameter) in ein übertragungsfähiges Format (Umkehrung: Unmarshalling). Sicherung eines einheitlichen und definierten Datentyps für Sender und Empfänger in unterschiedlichen Adressräumen, Planarisierung komplexer Datenstrukturen.

Heterogenität Hardware und Betriebssystem
Ausgleichen der Heterogenität u. Zulassen der Individualität der einzelnen Knoten. Beispiele:
1. Big-Endian- und Little-Endian-Darstellung von Zeichen

Reihenfolge der Abspeicherung von Bytes der Mehr-Byte-Datentypen im Speicher:
- *Big-Endian-Systeme* (z.B. Motorola, IBM /390, SUN) speichern höherwertige Bytes auf niedrigeren Speicheradressen,
- *Little-Endian-Systeme* (z.B. Intel i386, Pentium, Vax) speichern höherwertige Bytes auf höheren Speicheradressen.

Beispiel der Zahl 1347: 1347 = 00000000 00000000 00000101 01000011

Speicheradresse	Big-Endian-Darstellung	Little-Endian-Darstellung
00	00000000	01000011
01	00000000	00000101
02	00000101	00000000
03	01000011	00000000

Übertragung in vereinbarter Network Byte Order (TCP/IP: Big-Endian-Format ~> Sockets).

2. Unterschiedliche Formate der Zeichendarstellung in Betriebssystemen
- IBM-Mainframes: *EBCDIC*-Format (Extended Binary Coded Decimal Interchange Code),
- UNIX, Windows: *ASCII*-Zeichencode (American Standard Code for Information Interchange, sog. ANSI X 3.4),
- Java-Plattform: *Unicode* Standard, (Hardware-) plattform-unabhängiger Zeichencode.

Korrekte Interpretation der Daten sowie Aufbereitung für die Anwendungskomponenten. Unterschiedliche Daten werden in ein plattformspezifisches Format gewandelt (transparent für Anwendung).

Heterogenität der Programmiersprachen
Programmiersprachen mit individuellen Datentypen und Hauptspeicherdarstellung. Zielstellung: unterschiedliche Anwendungen (z.B. Java, C++, C#) sollen korrekt interpretiert werden können. Lösung für Marshalling / Unmarshalling: durch übergeordnete Formate, die den Kommunikationspartnern und der Middleware bekannt sein müssen. Unterscheidung zwischen plattformspezifischen und externen Datenformaten.

Plattformspezifisches Datenformat
Format dient als Grundlage der Datentransformation und Datenwiederherstellung. Daten werden in ein Übertragungsformat (i.allg. Byteformat) transformiert. Beispiele:
1. CDR (Common Data Representation) der CORBA-Plattform: CDR ist unabhängig von bestimmter Programmiersprache, kennt jedoch alle in CORBA verwendeten Formate.
2. Byteformat der Java-Plattform: Annahme, dass alle Partner die gleiche Sprache (Java) nutzen. Somit kein zusätzliches Format erforderlich. Java-Objekte werden direkt durch Objektserialisierung in ein Byteformat umgewandelt.

Unterstützt durch viele Middleware-Plattformen. Realisierung durch Schnittstellenbeschreibungssprachen und deren Compiler, z.B. XDR (eXternal Data Representation), IDL (Interface Definition Language) ~> Gestaltung der API (Application Programming Interface).

Externes Datenformat
Plattformunabhängiges Format zur Datenübertragung. Weitgehend durchgesetzt: XML (eXtended Markup Language): Deklarative Sprache zur Beschreibung von beliebigen Strukturen (Dokumente, Daten usw.). Basis: SGML. Semantische Grundlage für XML-Daten ist ein *XML-Schema*: es definiert die Bedeutung der XML-Strukturen.
Nachteil XML gegenüber plattformspezifischer Datenformate: kompakte Datenrepräsentation -> Vergrößerung des Übertragungsvolumens -> Verringerung der Effizienz.
Vorteil XML: eindeutige Repräsentation der Daten -> Transparenz, Wiederverwendbarkeit.
Anwendung XML: Internet / WWW, Web-Services, Web-Technologien (AJAX), Web 2.0 Applications, Geschäftsmodelle für B2B (EAI, SOA).

2.2.2 Programmiermodelle (RPC, RMI, Messaging)

Programmierparadigmen

Programmiermodell bestimmt das Programmierparadigma (prozedural / objektorientiert) und die Kommunikationsform (synchron / asynchron).

Prozedurales Paradigma:

Arbeit mit *Prozeduren*, auf die direkt zugegriffen werden kann. Prozeduren haben keine Identität. Sie werden in Bibliotheken zusammengefasst und können von anderen Prozeduren genutzt werden. Unterstützung durch *imperative Sprachen*, Fortran, Basic, Cobol, C, ...

Objektorientiertes Paradigma:

Basiert auf den Konzepten *Objekt*, *Objektidentität*, *Attribut* und *Methode*. Objekte haben eine Identität, mit der sie eindeutig identifiziert werden können. Sie kapseln Attribute und Methoden und erhalten durch die Attributwerte einen eindeutigen Zustand. Der Zugriff auf die Attributwerte erfolgt über die Objektmethoden. Unterstützung durch *oo-Sprachen*, wie Smalltalk, Eiffel, Python, C++, Java.

Kommunikationsformen

Synchrone (blockierende) / asynchrone (nicht-blockierende) Kommunikationsform.

Synchrone Kommunikation: Sender einer Nachricht bleibt blockiert, bis vom Empfänger der Nachricht eine Antwort vorliegt. Performance-Verlust, aber gleichzeitiger Synchronisationseffekt.

Asynchrone Kommunikation: Sender nicht blockiert. Sender und Empfänger können parallel weitergeführt werden. Weitgehende Entkopplung der Partner, aber ggf. zusätzliche Synchronisation erforderlich.

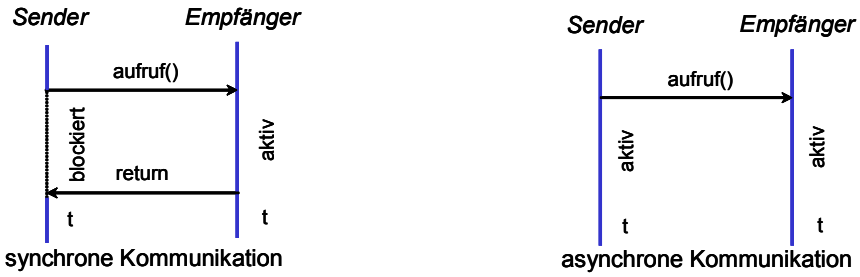


Abbildung 2.2: Kommunikationsformen synchron/asynchron

In verteilten Systemen werden folgende Programmiermodelle unterstützt:

- *Entfernte Prozeduraufrufe* (RPC: Remote Procedure Call):
  Kombination synchrone Kommunikation mit prozeduralem Programmierparadigma.
- *Entfernte Methodenaufrufe* (RMI: Remote Method Invocation):
  Kombination synchrone Kommunikation mit objektorientiertem Programmierparadigma.
- *Nachrichtenorientiertes Modell* (Messaging: Message Queueing, Message Passing):
  Asynchrone Kommunikation, Programmierparadigma beliebig.

Entfernter Prozeduraufruf (RPC)

RPC (Remote Procedure Call): ältestes VS-Programmiermodell: Birrell / Nelson (1984) für Netzwerkbetriebssystem Unix und Client/Server-Modell. Zugriff zur Prozedur lokal oder entfernt soll für Nutzer transparent sein. Lokaler bzw. entfernter Zugriff nach einheitlichem Prinzip. Beim entfernten Aufruf erhält der Client statt der tatsächlichen Prozedur einen *Stellvertreter:* eine **Client-Stub-Prozedur** (engl. Stub: Stummel).

Client-Stub-Prozedur simuliert das Verhalten der Serverprozedur, leitet jedoch die Aufrufe über das Netz zum Server weiter. Erforderlich wegen getrennter Adressräume, wodurch ein normaler UP-Aufruf (Stack) nicht möglich ist. Eine Server-Stub-Prozedur nimmt Aufruf entgegen und übergibt ihn an die Serverprozedur. In gleicher Weise werden die Ergebnisse an

Client zurückgeschickt. Client bleibt bis zum Empfang der Ergebnisse blockiert (~> synchrone Kommunikation).

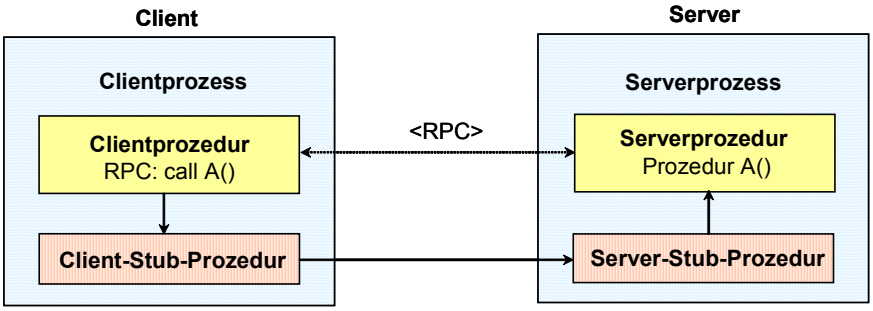


Abbildung 2.3: Ablauf eines entfernten Prozeduraufrufs

Client- und Server-Stub-Prozeduren werden aus der Schnittstellenbeschreibung der Serverprozedur generiert und stehen in lokalen Bibliotheken zur Verfügung. Schnittstellenbeschreibungssprachen, z.B. **XDR (eXternal Data Representation)** für Sun-RPC. Zusätzlich werden Hilfsprozeduren für Marshalling / Unmarshalling generiert und in Bibliotheken hinterlegt. Sie werden von den Stub-Prozeduren genutzt.

Entfernter Methodenaufruf (RMI)
RMI (Remote Method Invocation): Entfernter Methodenaufruf in objektorientierten Sprachen, analog den entfernten Prozeduraufrufen ~> Erweiterung des lokalen Zugriff von Objekten des Clients auch auf verteilte Objekte des Servers. Realisierung des entfernten Methodenaufrufs durch Einführung eines Stellvertreterobjekts (***Proxy-Objekt***) des eigentlichen Serverobjekts.

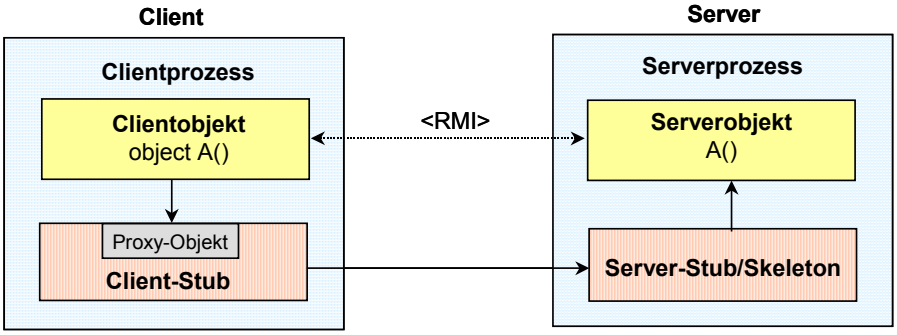


Abbildung 2.4: Ablauf eines entfernten Methodenaufrufs

Proxy-Objekt verhält sich gegenüber Client wie ein Serverobjekt. Intern leitet es aber die Aufrufe an das eigentliche Serverobjekt weiter. Proxy-Objekt ist Teil des Client-Stubs. Stub bietet weiterhin Funktionalität zu Marshalling, Unmarshalling und Verbindungsaufbau.
Auf Serverseite nimmt ein Server-Stub (Skeleton) die Aufrufe entgegen und übergibt sie an Serverobjekt. Die von der Methode ermittelten Ergebnisse werden in gleicher Weise an Client zurückgeschickt. Client bleibt bis zum Empfang der Ergebnisse blockiert (~> synchrone Kommunikation). Client- und Server-Stub werden aus der Schnittstellenbeschreibung des Serverobjekts generiert. Schnittstellenbeschreibung, Bereitstellung in Klassenbibliothek.
Beispiel: ***IDL (Interface Definition Language).***

Nachrichtenorientiertes Modell (Messaging)
Messaging: klassisches asynchrones Programmiermodell: Nachrichten direkt an einen oder mehrere Empfänger verschickt. Sender prüft nicht, ob Nachricht angekommen ist, und wartet nicht auf Ergebnis (~> asynchrone Kommunikation).
Formen: Message Queueing, Message Passing (~> MOM: Message Oriented Middleware).

Basis: Warteschlange (Entnahme aus WS zu beliebigem Zeitpunkt). Keine Methoden- oder Prozeduraufrufe. Daten und Aufrufe werden in Form von Nachrichten verpackt und übertragen, das Format ist von der jeweiligen Middleware-Technologie vorgegeben.

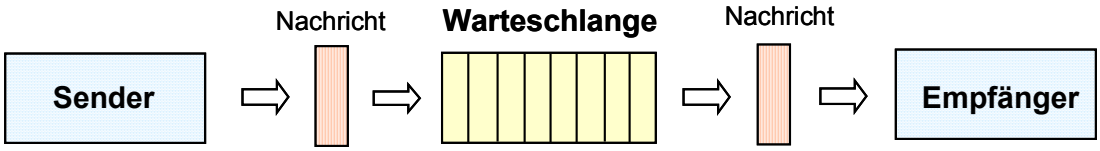


Abbildung 2.5: Asynchrone Nachrichtenübermittlung (Messaging)

Vorteile: Sender und Empfänger weitgehend voneinander unabhängig. Empfänger kann Zeitpunkt der Nachrichtenentgegennahme selbst bestimmen.
Nachteile: Sender weiß nicht, ob Empfänger empfangsbereit ist. Keine Synchronisation.

2.2.3 Technologien (Sun-RPC, Java RMI, MQSeries)

Plattformen

Es sind konkrete Implementationen der Middleware. Sie basieren i.d.R. auf offenen Standards oder zumindest standardisierten Schnittstellen. Verschiedene Klassen:

1. Middleware für entfernte Aufrufe:

Entfernte Prozedur- bzw. Methodenaufrufe (synchron, blockierend), u.a.

- verteilte Dateisysteme, u.a. NSF (Network File System), AFS (Andrew File System), ...
- RPC-Implementationen für entfernten Prozeduraufruf, u.a. Sun-RPC, DCE-RPC (Bestandteil der Verteilungsplattform DCE (Distributed Computing Environment) der OSF).
- Entfernte Objektaufrufe in Plattform CORBA (Common Object Request Broker Architecture), Gremiums OMG: statische und dynamische Aufrufe, IDL, Repositorium.
- Java RMI (Bestandteil der J2EE) für entfernten Methodenaufruf in Java,
- Web-Services mit Zugriffsprotokoll SOAP für web-basierten entfernten Prozeduraufruf.

2. Message Oriented Middleware (MOM):

Nachrichtenorientiert, Warteschlangentechnik (asynchron, nicht-blockierend). Technologie z.T. veraltet, geringe Standardisierung. Bekannteste Standards:

- Java Message Service (JMS),
- Quasi-Standard WebSphereMQ von IBM, auch bekannt unter MQSeries.

3. Asynchrone Web-Technologie:

- AJAX (Asynchronous JavaScript and XML): asynchrone Kommunikation zwischen Web-Client und Web-Server (Ergänzung zur statischen, synchronen Client/Server-Relation); Basis moderner Web-Applikationen, z.B. im Web 2.0.

Standardisierungsgremien

- OSF (Open Software Foundation, Teil der Open Group): DCE, DCE-RPC
- OMG (Object Management Group): CORBA, IDL, IIOP, IIOP-RMI
- Sun Microsystems: Sun-RPC, Java, J2EE (RMI, EJB)
- W3C (WWW Consortium): XML, Web-Services (SOAP, WSDL)
- OASIS (Organization for the Advancement of Structured Information Standards): UDDI
- IETF: TCP/IP

2.3 Anwendungsorientierte Middleware (context awared)

Aufgaben und Struktur

Erweiterung der kommunikationsorientierten Middleware um unterstützende Dienste und eine Laufzeitumgebung, insbes. Erweiterte Laufzeitumgebung, Dienstkomponenten, Komponentenmodell. Kommunikationsinfrastruktur entspricht in Aufgaben und Umfang der kommunikationsorientierten Middleware.

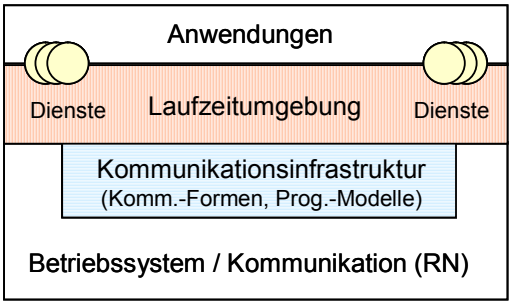


Abbildung 2.6: Struktur Anwendungsorientierter Middleware

2.3.1 Laufzeitumgebung

Erweiterte Laufzeitumgebung

Laufzeitumgebung des Betriebssystems nicht ausreichend für verteilte Anwendungen. Laufzeitumgebung der Middleware (sog. „Container“) setzt darauf auf, erweitert diese und stellt Funktionalität unabhängig von Hardware und Betriebssystem bereit. Aufgaben:

- Ressourcenverwaltung: Hauptspeicher, Prozessorzeit, Prozesse und Verbindungen.
- Nebenläufigkeit: Prozesse, Threads.
- Verbindungsverwaltung: Pooling.
- Verfügbarkeit: Fail-over-Cluster, Load-balancing-Cluster.
- Sicherheit: Authentisierung, Autorisierung, Integrität ~> Verschlüsselung, Zertifikate, Signaturen.

Zusammenwachsen Betriebssysteme und Middleware?

- Microsoft: Forcierung Zusammenwachsen, Plattform .NET ~> Herstellerabhängigkeit
- IBM, Sun, IONA u.a.: strikte Trennung ~> Interoperabilität

Ressourcenverwaltung

Verwaltung Hauptspeicher, Prozesse, Threads usw. (Erweiterung der BS-Funktionalität), inkl. Sicherheitskonzepte in den verteilten Knoten.

Nebenläufigkeiten

Parallele Ausführung von Anwendungskomponenten ~> Prozesse und Threads. Threads: leichtgewichtige Prozesse, im Prozessraum, Nutzung dessen Ressourcen. Betriebssystem bietet Basisfunktionalität für Prozesse und Threads, Middleware nutzt dies oft für eigenständige Prozess- und Threadverwaltung. Threadwechsel einfacher gegenüber Prozesswechsel.

Verbindungsverwaltung

Erweiterung der Betriebssystemressourcen in verteilten Systemen um *Verbindungen*. Verbindungen (Connections) sind Endpunkte von Kommunikationskanälen innerhalb einer Anwendung. Sie existieren an Tier-Grenzen und sind im aktiven Zustand immer mit einem Prozess oder Thread assoziiert. Verbindungen benötigen Speicher und Prozessorzeit ~> dynamische Zuordnung.

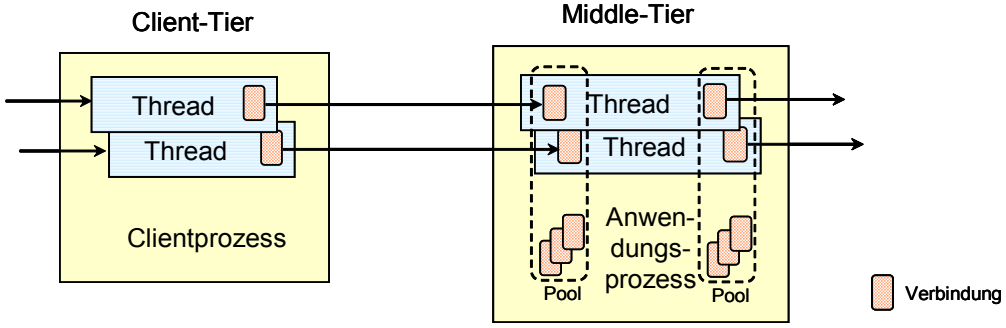


Abbildung 2.7: Verbindungsverwaltung an der Schnittstelle Client-Server

Pooling: aktuelle Technik zur Verbindungsverwaltung. Ein Pool übernimmt die Verwaltung freier Verbindungen und vergibt auf Anfrage freie Verbindungen an Threads. Initialisierung einer Verbindung erfolgt durch die Laufzeitumgebung oder Anwendung. Neu initialisierte Verbindungen werden in Pool gestellt. Bei Aufruf wird Verbindung aus Pool geholt und dem Prozess / Thread zugeordnet. Nach Ablauf wird Verbindung in Pool zurückgestellt.

Verfügbarkeit

Verfügbarkeit: eine Anwendung erfüllt ihre Aufgabe fehlerfrei und korrekt. Es können aber Störfaktoren bei HW/SW-Komponenten auftreten: Ausfall, Überlastung, Wartung von Komponenten. Lösung: Ignorieren der Störung oder Behebung ~> Verwendung eines Clusters. Einrichtung Cluster kostspielig. Für unkritische Anwendungen kein Cluster erforderlich.
Cluster: gebildet aus mehreren Rechnern mit Replikaten der HW und SW, nach außen als eine Einheit agierend. 2 Typen von Clustern:

- Fail-over-Cluster: Einsatz zur Vermeidung von Ausfällen. Für jede HW/SW- Komponente steht ein Replikat im Cluster bereit. Bei Fehler im Hauptserver wird Replikatserver aktiviert. Falls Wechsel für Anwendung transparent: sog. *Hot-Stand-by* ~> Replikate immer aktiv; falls nicht-transparent: sog. *Cold-Stand-by* ~> Replikatserver ist dann zu aktivieren.
- Load-balancing-Cluster: Zur Lastverteilung und Erhöhung der Parallelität, weniger zur Ausfallsicherung. Replikatserver immer aktiv, bearbeiten Anfragen parallel.

Sicherheit

Durch Netzverbindung erhöhte Gefahr vor unberechtigtem Zugriff ~> Sicherheitsmaßnahmen, insbes. für Zugriffskontrolle und Zugriffsrechte:

- Authentifizierung (Zugriffskontrolle): Sicherung der Identität des Benutzers (Kennung und Passwort);
- Autorisierung (Zugriffsrechte): Vergabe von Zugriffsrechten an Benutzer für bestimmte Dienste.

Zum Schutz der Daten gegen unberechtigtes Abhören oder Manipulation ist weiterhin der Kommunikationskanal für kritische Daten zu sichern:

- Vertraulichkeit: Verhinderung der Interpretation ~> *Verschlüsselungsalgorithmen*.
- Integrität: Sicherung der Datenmanipulation bei Übertragung ~> *Zertifikate*.

Verschlüsselung

Verschlüsselung der Daten mittels kryptographischer Methoden. 2 Verfahren für Umgang mit Schlüsseln: symmetrisch / asymmetrisch. Verschiedene Algorithmen: DES, IKEA usw.
Symmetrisches Verfahren: Sender und Empfänger mit gleichem (geheimen) Schlüssel. Problem: sichere Schlüsselverteilung.
Asymmetrisches Verfahren (Public-Key-Verfahren): Nutzung öffentlicher und privater Schlüssel. Beide gemeinsam von einem Schlüsselbesitzer erstellt. Public Key kann von jedem genutzt und beliebig verteilt werden. Private Key dient i.w. zur Entschlüsselung, verbleibt lokal beim Schlüsselbesitzer. Somit entfällt die kritische Verteilung der geheimen Schlüssel.

Authentisierung

Verfahren mit privaten oder öffentlichen Schlüsseln. Protokoll: *Needham-Schroeder*.

Signaturen und Zertifikate

Zur Sicherstellung der Authentizität des Senders und der Datenintegrität: Einsatz von elektronischen Unterschriften oder Signaturen. Bekannte Anwendungen: e-cash, elektronische Wahl. Grundlage: asymmetrische Verschlüsselungsverfahren (Public Key).
Eine Signatur wird durch Verschlüsselung der Nachricht mit privatem Schlüssel erzeugt. Empfänger kann Nachricht mit dem zugehörigen öffentlichen Schlüssel der Nachricht verifizieren. Signierte Nachricht wird zur Wahrung der Vertraulichkeit für Übertragung erneut verschlüsselt. Signatur allein gewährleistet noch keine Authentizität ~> es ist zu sichern, dass der

öffentliche Schlüssel zur Verifikation der Signatur tatsächlich zu der Person gehört, die die Signatur erstellt hat ~> erfolgt durch Zertifikate.
Zertifikate werden von Zertifikations-Authoritäten (CA: Certification Authority) vergeben. Diese müssen sichern, dass ein öffentlicher Schlüssel zu einer bestimmten Person und damit zu einem bestimmten privaten Schlüssel gehört. Zertifikat entspricht elektronischem Ausweis, zugeordnet zu einer Person oder Programm. Je nach Zertifizierungsstelle kann elektronische Signatur als rechtlich gültige Unterschrift oder als technische Unterschrift gelten.

2.3.2 Middleware-Dienste

Middleware bietet für eine Anwendung über eine Schnittstelle definierte Dienste an, i.d.R. technische Funktionen. Dienste anwendungunsabhängig, beliebig wiederverwendbar, implizit über Laufzeitumgebung verfügbar. Schnittstellen in Spezifikation festgelegt (firmenintern bzw. offen). Bekannte anwendungsorientierte Dienste: Namensdienst, Sitzungsverwaltung, Transaktionsverwaltung, Persistenzdienst. Hinzu kommen systemorientierte Dienste, ggf. bereits durch Laufzeitumgebung angeboten, u.a. Zeitdienst, Management.

Namensdienst
Namens- oder Verzeichnisdienst dient der Veröffentlichung der Dienste zur Auffindung. Eintragungen enthalten Adressen (sog. Referenz) von Ressourcen: HW/SW-Komponenten, z.B. Datenbank, Web-Service bzw. einfaches Objekt. Referenz im Internet: IP-Adresse, Port-Nr. des Dienstes, Identifikator des Prozesses und eines Objektes.
Dienstanbieter stellen Referenzen in den Namensdienst (Service Export). Um eine Referenz auf eine Resssource zu erhalten, übergibt bei Anmeldung der Client den Namen der Ressource an Namensdienst und erhält die Referenz (Service Import). Danach kann Verbindung zu der Ressource aufgebaut werden (Binding). Standard-Schnittstellen für Namensdienste. Bekannte Beispiele: Java Naming and Directory Interface (JNDI) in EJB, Interface Naming Service (INS) in CORBA. Interne Struktur eines Namensdienstes nicht festgelegt, i.allg. Baumstruktur der Name-Referenz-Paare.

Sitzungsverwaltung
Aufgabe: Verwaltung der Sitzungen vieler paralleler Anwender. Verwendete Techniken:
- HTTP Sessions am Webserver bei Webanwendungen,
- verschiedene Middlewaretechnologien mit Sitzungsverwaltungen.

Sitzungsverwaltung insbes. bei interaktiven Anwendungen. In der Sitzung werden relevante Daten zwischengespeichert und am Ende wieder gelöscht.
Sitzungsdaten: Informationen zum Anwender (u.a. Kennung, URL, verwendete Browser). Zusätzlich auch Zwischenergebnisse (transparent oder verborgen). Internet-Shops arbeiten mit visuellen Sitzungen in Form von „Warenkörben“; Beendigung der Sitzung, wenn Kunde zahlt. Eine Sitzung enthält 2 Arten von Daten:
- transiente (flüchtige) Daten: werden nach Sitzungsende gelöscht,
- persistente Daten: dauerhaft auf Datenträger abgelegt.

Transaktionsverwaltung
Dienst für interaktive, datenzentrierte Anwendungen, insbes. Erhalt der Datenkonsistenz (d.h. alle persistenten Daten der Anwendung repräsentieren in ihrer Gesamtheit einen gültigen Zustand). Bei parallelen Zugriffen können zwei Anwender auf gleiche Daten arbeiten und diese ändern. Zur Sicherung der Datenkonsistenz werden Transaktionen benötigt.
Transaktion:
Logische Klammer um eine Reihe von atomaren Aktionen. Diese sichern für die Durchführung der Aktionen folgende Eigenschaften (sog. **ACID**):
- Atomarität (*Atomicity*): alle oder keine Aktion innerhalb der Transaktion ausgeführt.

- Konsistenz (*Consistency*): Transaktion überführt einen konsistenten Zustand
  Immer in einen neuen konsistenten Zustand.
- Isoliertheit (*Isolation*): Transaktionen laufen isoliert ab, keine gegenseitigen Störungen.
- Dauerhaftigkeit (*Durability*): Zustand am Ende der Transaktion dauerhaft festgeschrieben.

Transaktionen von allen Ressourcen unterstützt, die Daten festschreiben, u.a. Datenbanken, Middleware-Technologien (insbes. Nachrichtenorientierte Middleware), Drucker-WS.
2-Phasen-Commit (2PC):
Festschreiben der Änderungen in einer verteilten Anwendung erfolgt in 2 Phasen:
- in ersten Phase befragt der Transaktionsverwalter alle beteiligten Ressourcenverwalter, ob die Änderung lokal festgeschrieben werden können.
- bei positiver Bestätigung aller Verwalter fordert der Transaktionsverwalter in einer zweiten Phase das Festschreiben der Änderungen.

Falls bei mindestens einem Ressourcenverwalter ein Konsistenzproblem auftritt, fordert Transaktionsverwalter alle Ressourcenverwalter auf, ihre lokale Transaktion zurückzusetzen. 2-Phasen-Commit bildet wesentlichen Unterschied zwischen verteilter und lokaler Transaktionsverwaltung. Entscheidend dabei die Schnittstelle zwischen Transaktionsverwalter und Ressourcenverwaltern: sog. XA-Schnittstelle (Standard der Open Group).

Persistenzdienst
Persistenz: Gesamtheit aller Mechanismen zur dauerhaften Speicherung flüchtiger Daten im Hauptspeicher auf persistentes Speichermedium. Persistenzdienst und Transaktionsverwaltung bilden die beiden wichtigen Säulen datenzentrierter verteilter Anwendungen.
Persistenzdienst bietet einer verteilten Anwendung eine intelligente Schnittstelle zu Festspeicher oder Datenbank: Daten an Dienst übergeben, auf geeignetes Speicherformat abgebildet und an Speichermedium oder DB weitergereicht.
In Praxis bewährter Persistenzdienst: OR-Mapper (objektrelationaler Mapper). Realisieren die Abbildung zwischen oo-Programmiersprache und relationaler DB. OR-Mapper arbeiten transaktional, unterstützen verteilte Transaktionen und verschiedene Sperrmechanismen. Persistenz als eigenständige Komponente oder in Middleware integriert. Kommerzielle Persistenzdienste und Open-Source-Dienste (z.B. Quasar Persistence, Hibernate).

2.3.3 Komponentenmodelle (DCOM, EJB, CCM)

Komponentenorientiertes Paradigma
Komponentenmodell: Konzeptionelle und technische Basis zur komponentenbasierten Entwicklung verteilter Anwendungen. Erweiterung des objektorientierten Paradigmas, verstärkt durchgesetzt.

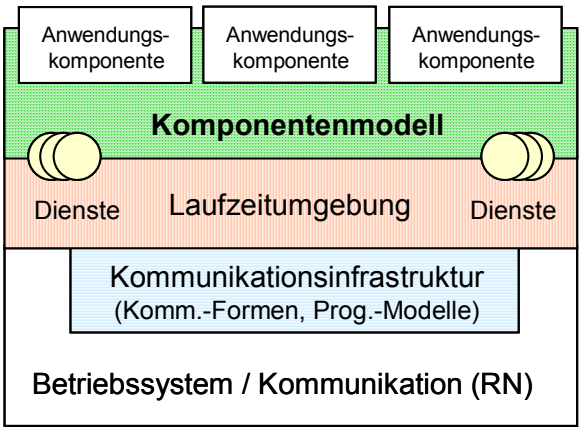


Abbildung 2.8: Struktur Anwendungsorientierter Middleware mit Komponentenmodell

Komponenten als „große Objekte“ dienen zur Strukturierung der Anwendung. Sie kapseln Teile der Anwendung und bieten Funktionalitäten über Schnittstellen an.

2 Aspekte der komponentenbasierten Architektur: *Entwicklung*: Grundlage ist der Komponentenbegriff, durch den die grundlegende Struktur der Komponenten festgelegt wird. *Laufzeitumgebung*: geeignete Ablaufumgebung für die fertigen Komponenten.

Elemente des Komponentenmodells
Komponentenbegriff: Grundlegende Philosophie des Komponentenmodells, u.a. Art und Anzahl der Schnittstellen und Komponenteneigenschaften. Aus Komponentenbegriff wird Entwicklungsstruktur der Komponenten sowie ihre Einbettung in die Bibliotheken der Komponentenlaufzeitumgebung festgelegt.
Schnittstellenverträge: Legen die Abläufe zwischen den Komponenten sowie zwischen Komponenten und Laufzeitumgebung fest. Damit Nutzungsform der Dienste definiert.
Komponentenlaufzeitumgebung: Setzt auf Laufzeitumgebung der Middleware auf und erweitert deren Funktionalitäten ~> auch als „Container" bezeichnet. 2 Aufgaben:

1. Verwaltung der Komponentenlebenszyklen: Lebenszyklus umfasst die Initialisierung der Komponenten, die aktive Phase im Hauptspeicher und die Löschung. Aufgaben:
   - Initiierung der Phasenwechsel und Überwachung der Phasen,
   - Verwaltung der Phasen (nach Entwicklung werden die Komponenten in der Laufzeitumgebung angemeldet).
2. Implizite Bereitstellung von Diensten: Komponenten benötigen Dienste der Middleware und ihre Abläufe. Anwendungen ohne Middleware mit Komponentenmodell müssen direkt Zugriffe auf Dienstkomponenten durchführen. Komponentenlaufzeitumgebung bietet den Komponenten diese Dienste implizit an. Die Komponente teilt bei der Anmeldung der Laufzeitumgebung die Anforderungen in Textformat mit, welche Dienste später benötigt werden.

Vorteile der Komponentenmodelle: Vereinfachung der Entwicklung und des Betriebes der verteilten Anwendungen, strikte Trennung von inhaltlichen und systemtechnischen Aspekten.
Bekannte Komponentenmodelle:
- EJB (Enterprise JavaBEans) in Plattform J2EE,
- COM bzw. DCOM (Distributed Component Object Model) in Plattform .NET,
- CCM (CORBA Component Model) in Plattform CORBA 3.0.

2.3.4 Technologien (DCE, ORB, Application Server, .NET, J2EE)

Typen Anwendungsorientierter Middleware-Technologien
1. Verteilungsplattformen (Distribution Platforms):
Erweiterte Laufzeitumgebung und zusätzliche Dienste, wie Persistenz, Sicherheit, Transaktion, Ressourcenverwaltung.
1a. Distributed Computing Environment (DCE):
Basis: Programmiermodell der entfernten Prozeduraufrufe und Kommunikationsinfrastruktur für verteilte Programme. Prozedurales Ablaufschema (DCE-RPC). DCE-Zelle.
Basis-Dienste:
- Datenrepräsentation und Remote Procedure Call (RPC)
- Threads, Namensverwaltung, Directory-Service, Sicherheit u. Uhrensynchronisation

Erweiterte Dienste (zur verteilten Datenhaltung):
- Verteiltes Dateisystem, Diskless Support, Integration von Personal Computern

Entwicklungsgremium: OSF (Open Software Foundation, Teil der Open Group). Produkte insbes. durch Digital Equipment Corporation. 1995 Ablösung durch objektorientiertes Paradigma (CORBA).

1b. Object Request Broker (ORB):
Basis: Programmiermodell der entfernten Methodenaufrufe und Kommunikationsinfrastruktur für verteilte Objekte.

ORB bietet Vielzahl von Diensten. ORB zwar ohne Laufzeitumgebung im eigentlichen Sinn, aber viele ORB bieten Laufzeitfunktionalitäten, wie Prozess- und Threadverwaltung
Bekanntester Standard: CORBA (Common Object Request Broker Architecture).
OMA/CORBA: Spezifikation der OMG. Produkte: Visibroker (Inprise), Orbix (IONA), ...
Diensteangebot: CORBA-Services (u.a. Name-, Event-, Persistence-, Security-, Transaction-Services), CORBA-Facilities und Adapter (BOA: Basic Object Adapter, POA: Portable OA). DII (Dynamic Invocation Interface), IIOP (Internet Inter-ORB Protocol), Java-Integration (RMI-IIOP). Schnittstellenspezifikation: IDL (Interface Definition Language).

2. Application Server:
Middleware-Technologie, die sich ausschließlich auf die Unterstützung der Anwendungslogik der Middle-Tier konzentriert. Unterstützen Konzepte wie bei ORB und Kommunikationsinfrastruktur, Dienste, Laufzeitumgebung mit Komponentenmodell und Unterstützung für verteilte Anwendungen. Application Server selten in Reinform, i.allg. Teil einer vollst. Middleware-Plattform.
Bekannte Application-Server: BEA – Weblogic, IBM – Websphere, Borland – AppServer, Evidian – JonAS.

3. Componentware (Komponentenbasierte Middleware-Plattformen):
Erweiterung der Application Server zu einer vollständigen Verteilungsplattform für verteilte Anwendungen mit zugehörigen Komponentenmodellen. Anwendungsobjekte werden als Komponenten definiert (Schnittstellen) und für eine Komponenten-Laufzeitumgebung bereitgestellt.
Bekannteste Standards und Entwicklungsumgebungen:
- Java 2 Platform Enterprise Edition (J2EE), mit Komponentenmodell EJB (Enterprise JavaBean).
- .Net-Plattform mit Komponentenmodell COM bzw. DCOM (Distributed COM).
- CORBA 3.0 mit Komponentenmodell CCM (CORBA Component Model).

2.4 Ursprung und Erweiterungen

2.4.1 Transaktionsmonitore

OLTEP und Transaktionsmonitore
Bilden Wurzeln datenzentrierter Middleware-Plattformen in 60 / 70er: Mainframes mit entsprechenden Betriebssystemen nicht ausreichend für verteilte Anwendung. Dominierend für entfernten Zugriff: Terminals (Datenfernverarbeitung).
Anwendungen: OLTEP-Anwendungen (OnLine Transactional Processing), d.h. Transaktionale Informationssysteme. Eigenschaften (wie bei heutigen Internet-Anwendungen): viele parallele Zugriffe, große Benutzeranzahl, hohe Anforderungen an Verfügbarkeit.
Entwicklung von Transaktionsmonitoren ~> Transaction Processing Monitor (TP-Monitor):
Aufgabe der TPM: Mainframe-Betriebssysteme um 2 Dienste zu erweitern:
- Verwaltung von Transaktionen,
- Verwaltung von Ressourcen, wie Hauptspeicher, Prozesse, Verbindungen.

Erste Transaktionsmonitore für Mainframes Ende der 60er, z.T. noch erhalten:
- CICS (Customer Information Control System) und IMS/TPM, beide IBM,
- BS2000 von Siemens.

Technologie weiterlebend infolge Leistungsstärke der Mainframes.

Encina und Tuxedo
Heute Weiterentwicklung der Transaktions-Technologie durch IBM und Siemens, insbes. durch Integration neuer Technologien wie Java, CORBA, J2EE. Mit dem Internet wurden

mehr und mehr die OLTEP-Anwendungen von Mainframes auf erteilte Systeme migriert. *Migration*: Die Anwendung wird auf der neuen Technologie entwickelt, mit zusätzlicher Funktionalität. Das Kernstück, die Daten, werden in die Datenbank der neuen Anwendung überführt. Die Aufgaben der Transaktionsmonitore übernahmen nun spezielle Server-Transaktionsmonitore wie Encina (IBM) oder Tuxedo (BEA) ~> verschmolzen mit der kommunikationsorientierten Middleware in den heutigen Application Server.

2.4.2 Web-Services

Web-Services
Es sind Anwendungskomponenten, die auf den Knoten eines web-basierten System liegen. Technologisch eher den entfernten Aufrufen zuzuorden: über das Protokoll SOAP (Simple Object Access Protocol, Basis XML) werden web-basierte entfernte Aufrufe definiert. Unterstützung von Web-Anwendungen, u.a. bei Google, Amazonas. Standardisierung durch W3C und OASIS. Realisierungen von Web-Services:
- mit der .NET-Plattform.
- mit der Java-Plattform: JAX-RPC (Java APIs for XML-based RPC) ist ein Standard zur Entwicklung Java-basierter Web-Services. Entwickelt unter Verantwortung der JCP (Java Community Process), ein Standardisierungsgremium unter Sun Microsystems.
- mit der PHP-Plattform (PHP: Personal Hypertext Pre-Processor), Scriptsprache ~> serverseitige Web-Technologie.

4 Standards (für je einen Aufgabenbereich, unabhängige Standardisierungsgremien):
SOAP (ursprünglich Simple Object Access Protocol, ab Version 1.2 ohne Namen):
Zugriffsprotokoll für Web-Services, aufsetzend auf Transportprotokollen (z.B. HTTP, SMTP). Verwaltung durch W3C. Aktuelle Version: SOAP 1.2. (Vorgänger: *XML-RPC*, Dave Winer, 1989)
WSDL (Web-Services Description Language): Schnittstellenbeschreibungssprache für Web-Services. Definiert in XML. Verwaltung durch W3C.
XML (eXtended Markup Language): Deskriptive Sprache zur Beschreibung und Austausch komplexer Datenstrukturen. SOAP und WSDL verwenden XML zur Beschreibung von Schnittstellen, Datenstrukturen und Übertragungsformaten.
Verwaltung durch W3C. Aktuelle Version (Febr. 2004): Version 1.1.
UDDI (Universal Description, Discovery, and Integration): Verzeichnisdienst zur Veröffentlichung der Web-Services (Namensdienst für Web-Services). Entwicklung seit 2000. Verwaltung durch OASIS (Organization for the Advancement of Structured Information Standards). Aktuell: UDDI-Version 3, i.allg. noch V2. (Vorgänger: *DISCO*, Microsoft).

Lightweight Programming Models
Für lose gekoppelte Systeme gewinnen einfache Innovationen gegenüber den komplexen Web-Services großer Firmen an Bedeutung, u.a. RSS (Rich Site Summary) oder REST (Representational State Transfer). RSS-Feeds (XML) vereinfachen die Speicherung und Weiterverarbeitung von Web-Seiten. Somit Anwendungen einfach mit vorhandenen Komponenten verknüpfbar, z.B. Google Maps und Foto-Plattform flickr (per Geotags damit Fotos dem Ort der Erstellung zuordenbar -> auf Landkarte sichtbar).

2.4.3 Enterprise Computing (EAI, SOA)

Integrierte Konzepte für Geschäftsprozesse
Historische Entwicklungen für verteilte Anwendungen und ihre Plattformen:
- 60er, 70er: Mainframes, COBOL-Anwendungen
- 80er: Client/Server-Anwendungen, Unix, C, C++, Sockets
- 1. Hälfte 90er: RPC, DCE, OLE

- 2. Hälfte 90er: CORBA, COM/DCOM, Java, J2EE (1998)
- Anfang 2000: EJB, ActiveX
- Danach: .NET, Web-Services, AJAX.

Dadurch neue Geschäftsmodelle (business objects) und Dienstleistungen ermöglicht. Integration bestehender Lösungen (legacy) in neue, übergeordnete Anwendungen. Probleme bei Integration heterogener Anwendungssysteme: Interoperabilität. Schlüssel dazu: **EAI, SOA.**
Zielstellungen: Unterstützung für heterogene verteilte Systeme von Geschäftsprozessen (business objects), Anpassbarkeit (time-to market), Wiederverwendbarkeit (reusable services).

Enterprise Application Integration (EAI):
EAI (Vorstufe zu SOA): Mischung aus Konzepten, Technologien und Werkzeugen. Im Gegensatz zu SOA noch stärker auf die technische Struktur orientiert. EAI fokussiert die Integration eigenständiger Anwendungen, Middleware ist mehr auf Kommunikation zwischen Anwendungskomponenten ausgerichtet.

Service Oriented Architecture (SOA):
Im Gegensatz zum technisch orientierten EAI ist SOA ein fachlich orientierter, dezentraler, service-getriebener Ansatz. Dienste-orientierte Architektur (seit 2005) nach dem Konzept der SW-Komponenten zur Entwicklung verteilter, heterogener Geschäftsprozesse. Es bietet ein
- Managementkonzept für schnelle Reaktionen auf Änderungen im Geschäftsumfeld und
- Systemarchitekturkonzept zur Bereitstellung von Diensten und Funktionalitäten in Form von Diensten.

Service als Funktionalität definiert, die über standardisierte Schnittstelle genutzt wird. Somit Konzept der Softwarekomponenten. Anwendungssysteme zur Unterstützung von Geschäftsprozessen realisiert durch Aneinanderreihung von Serviceaufrufen („Komposition von Services“). Programmlogik dabei nicht in einem Programm, sondern über mehrere unabhängige Dienste verteilt.
Enterprise SOA-Konzept
- SOA sieht Menge lose gekoppelter, voneinander unabhängiger Dienste vor.
- Dienste von einem *service provider* angeboten.
- Ein *service consumer* stellt Anfrage (*service request*) an einen Dienst, die vom Anbieter mit einem *service response* beantwortet wird.
- Services: für Funktionen und Daten eines Dienstes zuständig und kapseln den Zugriff.

Komponenten im SOA-Konzept:
- Services (über Service-Repositiories lokalisierbar, Sicherung der Authentizität).
- Service-Konsumer und Service-Produzent.
- Enterprise Service Bus (ESB) als zentraler Kommunikationsweg für gesamten Datenaustausch zwischen Service-Konsumer und -Produzent.

Aufbau von SOA‘s auf jeder dienstbasierten Technologie möglich, u.a.
- Web Services (häufiger Einsatz),
- CORBA, DCOM, EJB, WCF (Windows Communication Foundation in .NET).

SOA häufig zur Integration der verschiedenen Programmiersprachen und VPF genutzt.
Weiteres wichtiges Ziel von SOA: Kapselung persistenter Daten durch Dienste, die exklusives Lese- und Schreibrecht auf ihre Daten besitzen.

2.4.4 Web-Technologien

Entwicklung von Web-Technologien als globale Verteilungsplattform (ab 2000/2004)
- Nutzung der verteilten Struktur im Web (Internet, Adressierung, Server).
- Gestaltung von Web-Applikationen als verteilte Anwendungen:
  - Web-Architektur: HTTP 1.1, URI, Datenstrukturen (XML, CSS, SMIL, ...).
  - Clientseitige Interaktion zwischen Browser und Web-Server: JavaScript (Interpreter).

- Serverseitige Interaktion: PHP (bzw. CGI), Nutzung Formulartechnik für Parameterübergaben.
- AJAX (Asynchronous JavaScript and XML): asynchrone Datenübertragung zwischen Client und Server ~> ermöglicht nicht-blockierende Arbeitsweise im Client.
  Voraussetzungen: JavaScript und XMLHttpRequest-Objekt (in allen gängigen Web-Browsern vorhanden).
  Alternative zu AJAX: Mono (.NET), Gecko-Engine (Mozilla).
- Einsatz von Web-Services (SOAP, WSDL). Interaktion zwischen Web-Servern.

• Entwicklung von Web-Applikationen (Weblications).
  Beispiele:
  - Web 2.0: social Web, u.a. YouTube, Flickr, Wiki, Blog (Read-Write-Web).
  - RSS-Feeds (Rich Site Summary): Speicherung von Artikeln einer Webseite bzw. von Nachrichten in maschinenlesbarer Form zur Weiterverarbeitung; reine XML-Datei, ohne Layout. Einsatz bei Weblogs.
  - Online-Bezahlsysteme (wie Paypal).
  - Datenbanken (wie Google web crawl).
• Ergänzung:
  Verlagerung Speicherung/Verarbeitung ins Internet, PC nur als Terminal („Webtop“).
  Bereitstellung von Speicherplatz und CPU-Leistung im Internet (Google, Amazon S3 und EC2, Windows Azure).
  Cloud Computing (IBM, Cebit 2009).

3 Kommunikationsmodelle

3.1 Interprozesskommunikation (IPC)

Der Informationsaustausch zwischen Programmen bzw. Prozessen soll so organisiert werden, dass das Verfahren für lokale und entfernte Zugriffe für den Nutzer gleiche ist. Lokal: IPC (Interprozesskommunikation), entfernt: Telekommunikation (Networks).
Je nach Kooperationsmodell bezeichnet man die Prozesse als Sender/Empfänger, Erzeuger/ Verbraucher, Client/Server (Slave/Master), Peers (gleichberechtigte Teilnehmer)
Unabhängig davon, wie die Nachrichten transportiert werden, lassen sich verschiedene Formen der Kommunikation charakterisieren (Kommunikationsmodelle).

Kriterium	Optionen	
Adressierung	direkt	indirekt
Blockierung	synchron	asynchron
Pufferung	ungepuffert	gepuffert, Mailbox
Kommunikationsform	meldungsorientiert (Nachricht, Messaging)	auftragsorientiert (Aufruf, Prozedur/Methode)

3.1.1 Adressierung

Für Zielangabe der Nachrichten zwei Prinzipien: die **direkte** und die **indirekte** Adressierung.
Direkte Adressierung: Sender gibt den Empfänger explizit an. Außerdem, ob beide Parteien direkt adressieren (*symmetrisch*) oder nur eine, wobei die andere dann aus der Nachricht den Absender (id) entnehmen kann und diesem wiederum direkt antwortet (*asymmetrisch*).
Indirekte Adressierung: Sender spricht Empfänger über eine Empfangsstelle an, z.B. Postfach oder Port. Bei **Mailbox** werden global eindeutig benannte FIFO-Puffer verwendet, in die der Sender seine Nachricht legt und aus der sie der Empfänger später abholt. Bei **Ports** wird die Kommunikation über spezielle, vom Betriebssystem zur Verfügung gestellte Sende- und Empfangspunkte realisiert wird (analog Sockets). Die Kommunikation über einen Sende- und einen Empfangsport erfordert eine Verbindungsherstellung mittels eines expliziten **Bindevorgangs**, wofür ein externer Vermittlungsdienst benötigt wird.
Für beide Verfahren müssen die Adressen für den Prozess zur Verfügung gestellt werden. I.w. werden drei Möglichkeiten genutzt: statisches Binden (feste Adressen), Broadcast-Anfrage und Nutzung eines Namensdienstes oder Traders.

Methode	Vorteile	Nachteile
Statisches Binden Die Adressen sind fest im Programmcode integriert.	- einfach zu programmieren - schnell	unflexibel (jede Adressänderung erfordert Neuübersetzung)
Broadcast-Anfrage Client schickt Adressanfrage Broadcast ins Netz und der zugehörige Server antwortet mit sog. „here-i-am“-Nachricht, aus der seine Adresse entnommen werden kann.	- Lokationstransparenz - unabhängig von festen Adressen	je nach Häufigkeit der Anfragen hohe Netzlast und lange Antwortzeiten (Caching möglich)
Namensdienst/Trader Client schickt Anfrage an ihm bekannten Trader (statisch gebundene Adresse oder Broadcast nötig). Dieser kennt alle Adressen und schickt die angeforderte zurück.	- sobald Adresse des Namensdienstes oder Traders bekannt: kein Broadcast mehr nötig. - keine festen Adressen	Namensdienst/Trader muss verfügbar sein

3.1.2 Blockierung

Nachrichtenaustausch zwischen zwei Partnern: blockierend (**synchron**) oder nicht blockierend (**asynchron**).

Bei **synchronem Nachrichtenaustausch** bleibt der Sender blockiert, d.h. er wartet, bis die Nachricht losgeschickt und evtl. eine Empfangsbestätigung angekommen ist. Der Empfänger blockiert entsprechend, bis eine Nachricht angekommen ist. Vorteilhaft ist die implizite Synchronisation zwischen Sender und Empfänger, die keine zusätzlichen Mechanismen erfordert, und keine Pufferung von Nachrichten nötig ist, was Verwaltungsaufwand und Speicherplatz spart. Die Parallelität von Sender und Empfänger wird jedoch stark eingeschränkt und der Empfänger kann zu jedem Zeitpunkt nur auf jeweils eine Nachricht warten. Da Sender oder Empfänger ausfallen können, muss zur Verhinderung einer ständigen Blockierung ein timeout verwendet werden, was eine zusätzliche Fehlerbehandlung erfordert.

Bei **asynchronem Nachrichtenaustausch** wartet der Sender nicht, bis die Nachricht im Netz ist, sondern kehrt von der Senderoutine zurück, sobald die Daten in den Ausgangspuffer geschrieben wurden (je nach Implementierung kann dies aber auch der Empfangspuffer des Empfängers sein). Der Empfänger versucht, Nachrichten aus seinem Empfangspuffer zu lesen und verarbeitet sie, sobald sie ihn erreichen, ohne auf sie zu warten. Vorteilhaft ist hier die zeitliche Entkopplung von Sende- und Empfangsvorgang, die parallele Verarbeitung und Anwendung in Echtzeit- oder ereignisgesteuerten Systemen ermöglicht. Jedoch ist es zur Gewährleistung von Zuverlässigkeit notwendig, auf beiden Seiten Nachrichten zu puffern. Sind die Puffer voll, muss der Sender blockieren, was dem asynchronen Konzept widerspricht.

3.1.3 Nachrichtenpufferung

Ungepufferte Kommunikation: falls jede Nachricht direkt an den Empfänger zugestellt werden soll, muss dieser zum Zeitpunkt des Eintreffens der Nachricht bereits auf diese warten und Speicher zur Übernahme reserviert haben. Ansonsten muss das Kommunikationssystem die Nachricht verwerfen, was im Falle eines Servers, der Anfragen mehrere Clienten erhält, oft der Fall wäre. Jedoch spart diese Verfahrensweise Speicher und Verwaltungsaufwand. Der Sender ist dabei gezwungen, darauf zu warten, bis das Kommunikationssystem die Nachricht erfolgreich abgeliefert hat, wenn er sicherstellen will, dass sie ankommt (ggf. neu senden).

Gepufferte Kommunikation: stellt das Betriebssystem Puffer bereit, um ein- und ausgehende Nachrichten zwischenzuspeichern, so müssen Nachrichten nicht verworfen werden, so lange Platz im jeweiligen Puffer frei ist, falls der Empfänger oder das Netz nicht bereit sind, sie abzunehmen. Dazu ist eine Pufferverwaltung nötig, um Speicherplatz bereitzustallen und Pufferüberläufe zu verhindern

3.2 Kommunikationsformen

Basisformen

Meldungsorientiert: Messaging, nachrichtenorientiertes Paradigma (asynchron)

Auftragsorientiert: Entfernter Aufruf (synchron). 2 Formen

entfernter Prozeduraufruf (RPC: Remote Procedure Call), prozedurales Paradigma,

entfernter Methodenaufruf (RMI: Remote Method Invocation), objektorientieres Paradigma.

Die beiden Basisformen der Kommunikation, meldungsorientiert und auftragsorientiert können in Abhängigkeit von den Blockierungseigenschaften in vier Klassen dargestellt werden.

Kommunikationsformen	**asynchron**	**synchron**
meldungsorientiert	Datagramm	Rendezvous
auftragsorientiert	asynchroner entfernter Dienstaufruf	synchroner entfernter Dienstaufruf

3.2.1 Meldungsorientierte Kommunikation

Bei meldungsorientierter Kommunikation werden grundsätzlich unidirektional Nachrichten versandt. In der standardmäßigen **Datagramm-Form** ist die Kommunikation asynchron, abgesehen von optional eingesetzten Quittungsnachrichten. Dabei werden Datagramme verschickt, ohne dass der Sender blockiert. Die Sicherstellung eines korrekten Transports ist dem Transportsystem überlassen.
Bei der ergänzenden **Rendezvous-Form** (synchron) erfolgt eine Blockierung. Die Quittungsnachrichten werden zur Synchronisierung benötigt, wobei der Sender blockiert, bis der Erhalt der versendeten Nachricht bestätigt wurde.

3.2.2 Auftragsorientierte Kommunikation

Auftragsorientierte Kommunikation beinhaltet stets die Übermittlung eines Ergebnisses vom Empfänger der Auftragsnachricht zurück zum Sender (bidirektionla). Bei der synchronen Variante blockiert der Sender, bis das Ergebnis ihn erreicht, wohingegen er bei asynchroner Kommunikation weiter arbeitet und später das Ergebnis empfängt.

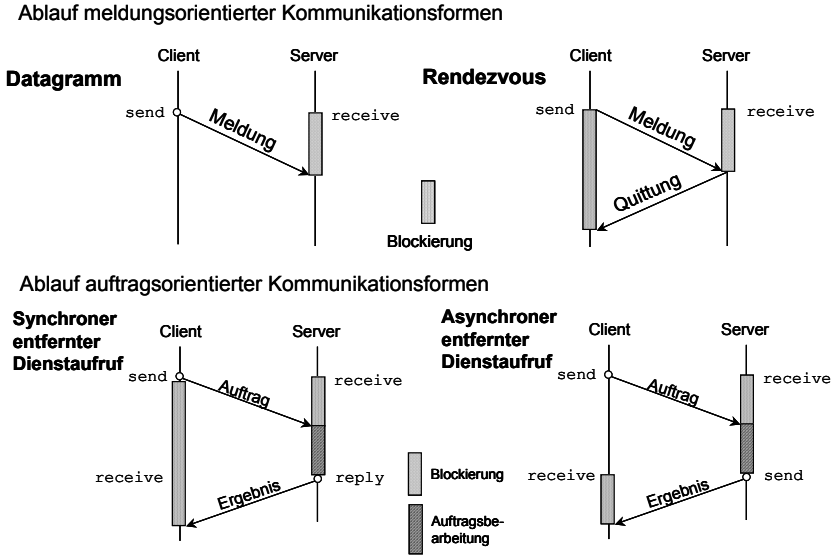


Abbildung 3.1: Meldungs- und Auftragsorientierte Kommunikationsformen

Bei auftragsorientierter Kommunikation ist der korrekte Ablauf wichtig (Sender, Empfänger und Kommunikationssystem korrekt funktionieren). Ist der Zielrechner z.B. nicht erreichbar, muss bei einem Adressierungsproblem das Programm geändert werden, wohingegen bei Netzwerkfehlern u.U. das Transportsystem (z.B. erneutes Senden) den Fehler beheben kann. Wird asynchrone Kommunikation verwendet, bleibt beim Absturz eines beteiligten Prozesses auch der andere hängen. Hier werden spezielle Prozeduren zur Fehlerbehandlung nötig.

3.2.3 Programmiermodelle

Programmiermodelle bilden Basis der sog. kommunikationsorientierten Middleware, gekennzeichnet durch

- *Programmierparadigma* (bei MW: prozedural / objektorientiert) und
- *Kommunikationsform* (Blockierung: synchron / asynchron).

In verteilten Systemen werden folgende Programmiermodelle unterstützt:

- Entfernte Prozeduraufrufe (RPC: Remote Procedure Call):
  Kombination synchrone Kommunikation mit prozeduralem Programmierparadigma.
  Anwendung: Sun-RPC, DCE-RPC, DCOM, NFS, ... , XML-RPC (SOAP, Twitter)
- Entfernte Methodenaufrufe (RMI: Remote Method Invocation):
  Kombination synchrone Kommunikation mit objektorientiertem Programmierparadigma.

Anwendung: Java-RMI, Corba (RMI-IIOP).

- Nachrichtenorientiertes Modell (Messaging: Message Queueing, Message Passing): Asynchrone Kommunikation, Programmierparadigma beliebig.
  Anwendung: JMS, MQSeries, Pull-Dienste (RSS-Feeds).

3.2.4 Fehlersemantiken und Fehlerbehandlung

Unterscheidung der Fehlersemantiken in vier Klassen:

Maybe

Es finden keine Fehlerbehandlungsmaßnahmen statt. Dienst wird **nicht oder höchstens 1 mal** durchgeführt. Kein Hinweis im Fehlerfall, ob Dienst ausgeführt wurde. Anwendung: Auskunftsdienste (erneutes Probieren nach gewisser Zeit, falls Auskunft nicht erteilt).

At-Least-Once

Fehlersemantik sichert, dass Auftrag **mindestens 1 mal** ausgeführt wird. Bei Nachrichtenverlust wird Auftrag bis zum Erfolg wiederholt. Da keine Filterung von Duplikaten, kann ein Auftrag mehrfach ausgeführt werden. Bei idempotenten Operationen (d.h. mehrfache Ausführung verändert Ergebnis nicht) ist diese Semantik ausreichend, z.B. wiederholtes Lesen einer Datei.

At-Most-Once

Fehlerbehandlung: Wiederholung (wie At-Least-Once), zusätzlich alle Duplikate ausgefiltert. Bei Nachrichtenverlust Auftrag **höchstens 1 mal** ausgeführt. Falls Server abgestürzt, so wird kein Auftrag mehr erwartet --> somit Atomarität der entfernten Operation garantiert. Auftrag komplett ausgeführt oder ein Fehler an Client gemeldet (dieser muss dann Folgehandlung organisieren). Typische Anwendung: RPC (Remote Procedure Call).

Exactly-Once

Fehlersemantik berücksichtigt Absturz und Wiederanlauf der Komponenten. Konsistente Rücksetzungsmaßnahmen sichern, Operationen werden **genau einmal** durchgeführt. Implementierung diese Fehlersemantik durch Aufsetzen persistenter Datenhaltung und verteilte Transaktionsmechanismen auf ein At-Most-Once-Protokoll. Großer Aufwand - nur bei hohen Sicherheitsanforderungen sinnvoll.

Exemplarisch sollen vier häufige Fehlersituationen und ihre Behandlung dargestellt werden:

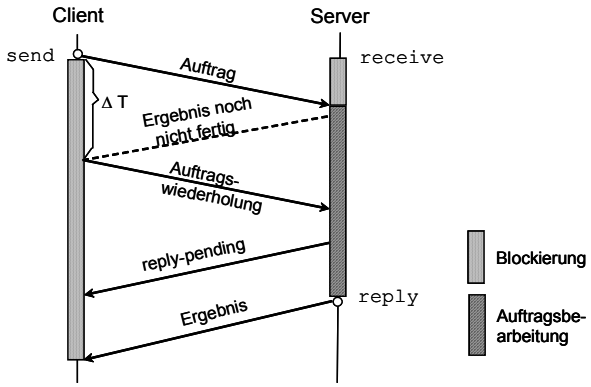


Abbildung 3.2: Auftragsbearbeitung (Beispiel)

Beispielszenarien für Fehlersituationen

- **Auftragsnachricht ging verloren:** Auftragswiederholung.
- **Antwort des Servers ist nicht angekommen:** Auftragswiederholung. Server kann Antworten eine Zeit lang zwischenspeichern und bei Wiederholungsaufträgen ohne Neuberechnung versenden.
- **Server ist abgestürzt:** Auftragswiederholung nach timeout. Nach Serverneustart tritt i.allg. **Server-Amnesie** auf, d.h. Server hat keine Kenntnis über bereits bearbeitete Aufträge, eine Rückfrage beim Client ist nötig. Falls Kommunikationsbeziehungen ungültig Neuaufbau

durch Client, ansonsten wartet der Server n * T, bis er wieder Aufträge annimmt, wobei n die max. Anzahl von Wiederholungen einer Nachricht und T die maximale Lebensdauer einer Nachricht (Client-timeout) ist; Clienten verwerfen in dieser Zeit alte Aufträge.
- **Server braucht lange für Bearbeitung des Auftrages:** Bei Eintreffen eines Wiederholungsauftrages wird **reply-pending Nachricht** an Client verschickt als Zeichen, dass sein Auftrag noch bearbeitet wird.

3.3 Entfernter Prozeduraufruf (RPC)

3.3.1 Remote Procedure Call (RPC)

Charakteristika: synchroner Aufruf (blockierend), Fehlersemantik at-most-once (höchstens 1 mal), prozedurales Programmierparadigma, kommunikationsorientierte Middleware.
Die RPC zugrunde liegende Idee ist, Anwendungen **Prozeduraufrufe** auf entfernten Rechnern zu ermöglichen, die sich von lokalen nicht unterscheiden. Sockets bieten keine Unterstützung für Kommunikationssteuerung und Datendarstellung. RPC entwickelt vor allem für Client/Server-Anwendungen.
Entwicklungswerkzeuge:

SunRPC	Teil der Sun ONC-Umgebung (Open Network Computing, Sun). Zusammen mit XDR (eXternal Data Representation) und NFS (Network File System). Von nahezu allen Herstellern unterstützt.
ApolloRPC	Unterstützung NCA-Plattform: Network Computing Architecture (nun HP)
RPC im OSF/1	DEC: Digital Equipment Corporation.
ECMA-RPC	
DCE-RPC	OSF / DCE (Weiterentwicklung des NCA RPC). Industriestandard, unterstützt u.a. von IBM, Microsoft. Einsatz im DCOM, .NET
Microsoft-RPC	Microsoft
XML-RPC	SOAP (Web-Service), Twitter (Web 2.0)

Mechanismus des entfernten Prozeduraufrufs (engl. Remote Procedure Call: RPC). Erstmalig veröffentlicht: Birell/Nelson (1984) [Lit.: Birell, A.D.; Nelson, B.J.: Implementing Remote Procedure Calls. ACM Transactions on Computer Systems. Vol. 2, 39-59, Febr. 1984].

Merkmale
- Komponente zur SW-Unterstützung der Kooperation in verteilten Systemen.
- Realisierung der synchronen Kontrollfluss- und Datenübergabe in Form von Prozeduraufrufen mit Parameterübergabe.
- Parameterübergabe an entfernte Prozedur, Ergebnisrückgabe.
- Günstige Unterstützung des Client/Server-Modells.
- RPC: synchrones Kommunikationsschema (Client ist während Ausführung der Server-Prozedur blockiert; At-Most-Once-Fehlersemantik („höchstens einmal")).

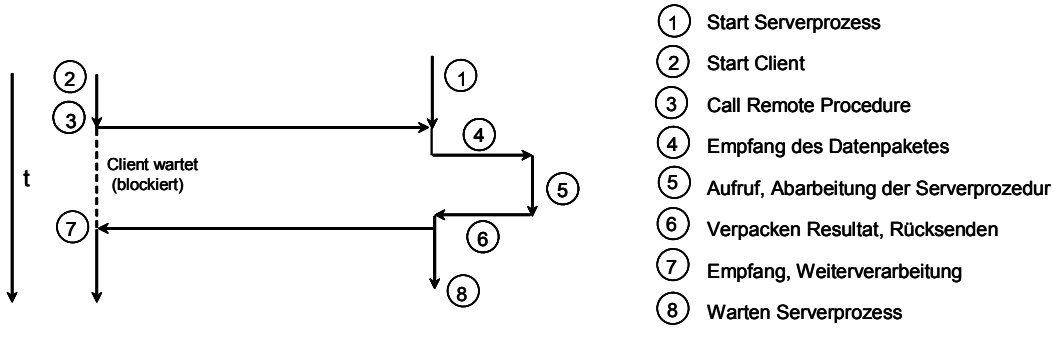


Abbildung 3.3: Ablaufschema RPC

3.3.2 Funktionsweise RPC

Prinzip

- RPC-System erlaubt *Aufruf von Prozeduren* auf anderen (entfernten) Rechnern.
- Übergabe von *Parametern* möglich
- Mechanismus eines *synchronen entfernten Dienstaufrufs*, d.h. wie bei lokalen Prozeduraufruf blockiert der Sender, bis Aufruf komplett abgeschlossen ist.

Programmierer kann somit nach gleichem Paradigma programmieren wie beim lokalen Programm (UP-Technik). Dennoch Besonderheiten und Einschränkungen beim RPC:

- Einbezogenes Kommunikationssystem ist zusätzliche Fehlerquelle.
- Da auf 2 verschiedenen Rechnern gearbeitet wird, gibt es *keinen gemeinsamen Adressraum* (--> Einfluss auf Parameterübergabe)
- Beteiligte Prozesse haben separate Lebenszyklen: somit nicht sicher, ob entfernte Prozedur ansprechbar ist.
- Durch Overhead der entfernten Kommunikation ist Aufrufdauer wesentlich größer als bei lokaler Kommunikation.

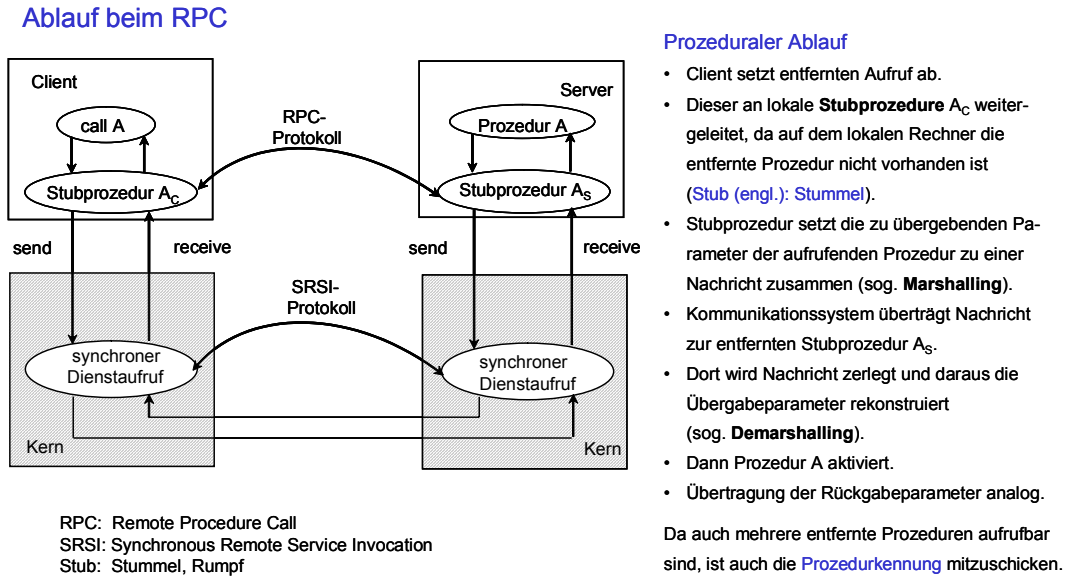


Abbildung 3.4: RPC-Ablauf und Stubprozeduren

Stubprozeduren (engl. Stub: Stummel)

Wenn ein Rechner einen RPC startet, ist die entfernte Prozedur auf dem gleichen Rechner nicht vorhanden (anderer Adressraum). Dazu Stubprozeduren eingerichtet (durch Stubcompiler), sog. Stellvertreterprozeduren. Sie übernehmen den Aufruf transparent für das aufrufende Programm, inkl. Parameter. Aufgaben der Stubs

- RPC-Aufrufe abfangen und an entfernte Instanz weiterleiten (so bleibt der Anwendung die entfernte Ausführung verborgen).
- Binden des Client-Anwenderprogramms an Server-Prozedur.
- Einsammeln der Parameter des RPC und in eine Nachricht einpacken (Marshalling) sowie umgekehrt Auspacken auf Gegenseite (Demarshalling).
- Fehlerbehandlungen (Kommunikationsfehler, Klientenfehler, Serverfehler).

Parameterübergabe

Parameter über das Netzwerk zu übertragen erfordert die Behandlung von Problemen der unterschiedlichen Datenformate (Architekturen little Endian, big Endian, Verwendung ASN.1) und Adressierungen. Unterstützung durch Verpacken und Entpacken der Parameter (sog. **Marshalling** bzw. **Demarshalling**).

Formen der Paramaterübergabe:
Call-by-Value: Im lokalen Fall wird eine Hilfsvariable angelegt, die mit übergebenen Wert geladen wird. Bei RPC genügt es, Wert des betreffenden Parameters zu übergeben (Beachtung: bei Array nicht nur Startzeiger übergeben, sondern gesamte Datenstruktur).
Call-by-Reference: Falls Parameter als Referenz vorliegen, ist Übergabe schwierig, da durch die getrennten Adressräume keine Dereferenzierung möglich ist. In diesem Fall muss der Inhalt des betreffenden Speicherbereiches typgerecht ausgelesen und später eventuell wieder geschrieben werden. Bei der Übergabe von Feldern bedeutet das, dass das gesamte Feld übertragen werden muss. Im Adressraum des Empfängers wird entsprechend eine Kopie angelegt und eine Referenz auf diese Kopie an die Prozedur übergeben.

Schnittstellenbeschreibung
Für korrekte Interpretation ist eine exakte Aufrufbeschreibung erforderlich. Diese Beschreibung (signature) enthält Name der Prozedur, Typen der Ein- und Ausgangsparameter, ggf. Vorschriften zur Ausnahmebehandlung (bei Fehlern).
Dazu spezielle Schnittstellen-Beschreibungs-Sprachen entwickelt. Dienen zur Spezifikation der Signatur und zur automatischen Generierung der Stubprozeduren (über zusätzliche Deklarationen). Zugehörige Generatoren: RPC-Compiler.
Bekannte Schnittstellenbeschreibungssprachen:
- ASN.1 (Abstract Syntax Notation): durch ISO
- XDR (eXternal Data Representation): bei Sun ONC-RPC (Open Network Computing)
- IDL (Interface Definition Language): bei OSF-RPC (Open Software Foundation)
- IDL (Interface Definition Language): bei CORBA / OMG
- Matchmaker: bei Mach

Beispiel (Sun-RPC)
Aufgabe: Addition 2er Integerzahlen (entfernt)
Aufruf: `result = add (request);` mit `request = a,b`
Schnittstellenbeschreibung in XDR (Datei add.x)

```
struct result     {int x;};
struct request    {int a; int b;
                   };
program ADD_PROG {
            version ADD_VERS {
                  result ADD (request)=1;
                  }=1;
             }=20 000 000;
```

Strukturen `result` und `request` deklarieren die Parameter
Struktur `program` vergibt
- Identifikatoren für die Prozedur `ADD` (=1)
- für die Version von `program` (=1) und
- eine Nummer des `program` selbst (=20 000 000)

RPC-Compiler **rpcgen** generiert aus der XDR-Beschreibung folgende Header-Datei (add.h):

```
struct result {int x;
            };
typedef struct result result;
bool_t xdr_result ( );
struct request {
         int a;
         int b;
};
typedef struct request request;
bool_t xdr_request ( );
#define ADD_PROG ((u_long) 20 000 000)
```

```
#define ADD_VERS ((u_long) 1)
#define ADD      ((u_long) 1)
extern result *add_1 ( );
```

Diese Header-Datei muss dann beim Client und beim Server eingebunden werden. `xdr_request` und `xdr_result` verweisen auf Konvertierungsroutinen, die zum Packen/Entpacken der Parameter verwendet werden.

Bindevorgang

Die einfachste Möglichkeit, festzulegen, an welchen Server ein Prozeduraufruf gesendet werden soll, ist, die Serveradresse **statisch** zu binden, also im Code des Clienten. Dabei muss bei jeder Änderung der Code des Clienten geändert werden (bzw. die von ihm verwendeten Ressourcen).

Eine andere Möglichkeit ist, jeden Server seine exportierten Prozeduren bei einem zentralen (oder verteilten) Register bekannt zu machen, auf das dann **dynamisch** vom Clienten entweder bei Initialisierung oder bei jedem RPC-Aufruf zugegriffen wird, um mittels eines **lookup** einen passenden Server auszuwählen. Diese Funktionalität wird aus Transparenzgründen vom Client-Stub gekapselt.

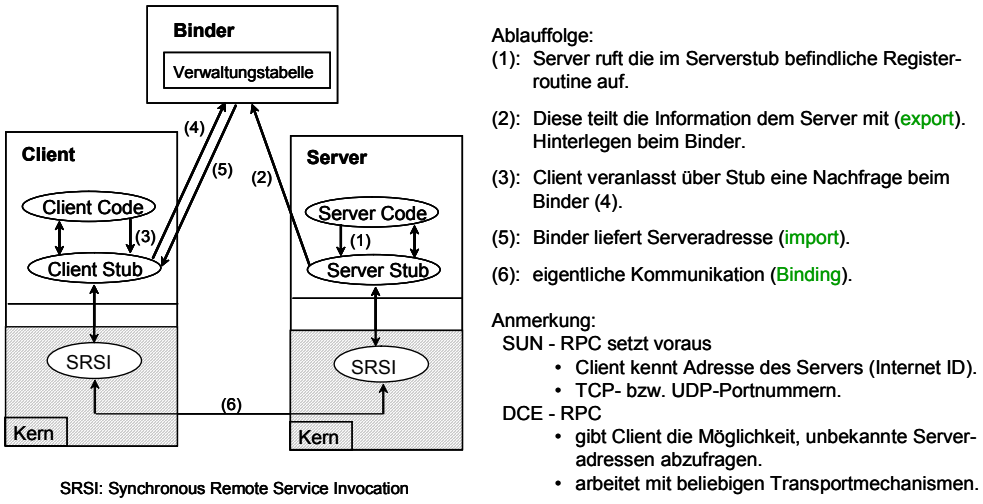


Abbildung 3.5: RPC Bindevorgang (Ablauf)

Dynamisches Binden (Verwendung eines Binders):

- Externe Instanz ("Binder"), bei der ein Server seine anzubietenden Prozeduren explizit bekannt macht (export oder register).
- Im Binderverzeichnis werden alle exportierten Prozeduren mit ihren Servern verwaltet (Server kann einen Dienst auch wieder deregistern).
- Gespeicherte Information: Tupel aus <servername, program, version, handle>
- Client läßt sich vom Binder einen Server vermitteln (lookup). Dazu erhält Client vom Binder ein sog. Handle (import) zum Verbindungsaufbau Client - Server.
- Lokalisierung des Servers kann erfolgen bei
  Initialisierung einer Anwendung (dynamisches Binden zur Initialisierungszeit) oder
  direkt vor Aufruf (dynamisches Binden zur Laufzeit). Letzteres sinnvoll bei langer Client-Lebensdauer, da sich Server ändern können).
- Binder und Verzeichnisse können auch verteilt implementiert sein.

RPC verwendet typischerweise zur Fehlerbehandlung eine **At-Most-Once** Semantik.

3.3.3 Sun-RPC

SUN-RPC ist das Ergebnis der Bemühung um eine von Betriebssystem und Rechner unabhängige, einheitliche Darstellung von Daten und der Möglichkeit des Unterprogrammaufrufes auf entfernten Rechnern ohne Programmkontextwechsel. Zusammengefasst als Open Network Computing (ONC) liegen die Ergebnisse in Form zweier Standards vor: **XDR** und **RPC**. Die Quellcodes der SUN-eigenen Implementierung stehen als Public Domain Software zur Verfügung.
Verwendung findet XDR/RPC hauptsächlich bei NFS (Network File System) und NIS (Network Information System). Auch für das Internet wurden die Standards übernommen und vom IETF als RFC 1050 (RPC), 1014 (XDR) und 1094 (NFS) veröffentlicht.
Bei SUN-RPC wird jedes RPC-Dienstprogramm über ein **Tupel** (program number, version number, procedure number) mit Einträgen von jeweils 4 Byte identifiziert. Der Client muss dazu noch die Adressierung des Server-Rechners im Netzwerk übernehmen, wobei die **Port-Nummer** des Servers dem Client nicht bekannt ist.
Stattdessen läuft auf jedem Server ein **Portmapper** auf dem Port 111, der zu einer Anfrage mit Programm- und Versions-Nummer die Portadresse zurückliefert. Somit geht dem jeweils ersten RPC-Aufruf einer Prozedur stets ein weiterer zur Erkennung des Zielports voraus.
Zur Schnittstellenbeschreibung dient **XDR (eXternal Data Representation)**. XDR ursprünglich nur zur Darstellung von Datentypen in einem unabhängigen Format gedacht, inzwischen zur vollständigen Schnittstellensprache entwickelt. Eine Schnittstellenbeschreibung in XDR enthält die Informationen: Programmnummer, Versionsnummer, Prozedurdefinitionen (Signatur und Nummer der Prozedur), Datentypdefinitionen (Datentypen an der Schnittstelle).
Mit C-Generator (rpcgen) werden aus der Schnittstellenbeschreibung alle Hilfsprozeduren generiert: Client- und Server-Stub, Dispatcher, Marshalling- und Unmarshalling-Prozeduren ~> als C-Prozeduren in Bibliothek abgelegt.

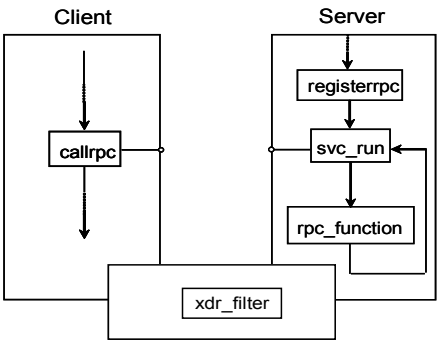


Abbildung 3.6: RPC-Aufruffolge

3.4 Entfernter Methodenaufruf (RMI)

3.4.1 Programmiersprache Java

Java: Eigenschaften
Objektorientierte, plattformunabhängige Programmiersprache. An C++ angelehnt, jedoch einfacher und klarer strukturiert. Laufzeitsystem mit Bytecode-Interpreter für Java (JVM - Java Virtual Machine) ~> Plattformunabhängigkeit.
Entwicklungsumgebungen: JDK (Java Development Kit) bzw. J2SE. Dynamisch ladbare Applets, mit WWW integriert (wichtiger Vorteil von Java-Nutzung), Einbindung in Web-Services.
Entfernte Kommunikation zwischen Java-Objekten via **RMI (Remote Method Invocation)** ~> Bestandteil der Java SE bzw. Java EE (Java Platform, Standard/Enterprise Edition; frühere Bezeichnungen: J2SE bzw. J2EE).

Schnittstellen zu CORBA (u.a. RMI-IIOP). Datenbankschnittstelle JDBC (Java Database Connectivity) sowie JDBC-ODBC-Bridge (Open Database Connectivity). Sun Microsystems.

Java Applets und Servlets

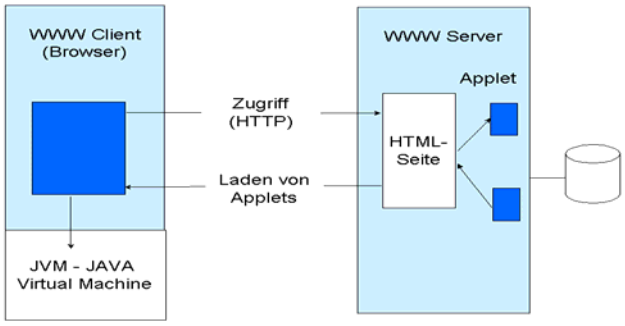


Abbildung 3.7: Java und WWW

Applet: vorgefertigte, vorkompilierte Java-Programmteile (Java-Klasse): ladbare Programmteile, Abspeicherung auf Server, im Internet / WWW integriert. Aufruf und Abarbeitung:

Browser: HTML mit Java-Tag <APPLET CODE = ...
Server: sendet Bytecode an Browser
Browser: Abarbeitung interpretativ (**Java Virtual Machine JVM**, z.B. ab Netscape 3.0)

Servlet: Vorgefertigte Java-Programmteile auf Browser. Abarbeitung auf Server.

3.4.2 Remote Method Invocation (Java RMI)

Entfernter Methodenaufruf in Java

RMI (Remote Method Invocation): entfernter Methodenaufruf zwischen verteilten Objekten. Unterstützung eines *synchronen* Programmiermodells (blockierend). Fehlersemantik: At-most-once (höchstens 1-mal) ~> kommunikationsorientierte Middleware.

Von Sun als Teil der Java-Plattform spezifiziert und voll in die Plattform integriert. De facto Spracherweiterung von Java zur Programmierung verteilter Anwendungen. In Java geschriebene Objekte, die in verschiedenen Adressräumen existieren, kommunizieren über RMI in einer Client/Server-Beziehung.

Ursprüngliches Protokoll **JRMP** (Java Remote Method Protocol) wegen Interoperabilität durch CORBA-Protokoll **IIOP** (Internet-Inter-ORB-Protocol) ersetzt. Konzept somit an CORBA angelehnt, aber einfacher und primär für verteilte Java-Anwendungen vorgesehen.

Java RMI Architektur

Java RMI setzt standardmäßig auf Internet-Protokollstack auf, aber auch andere Übertragungsprotokolle verwendbar. Mehrschichtige Architektur Java RMI:

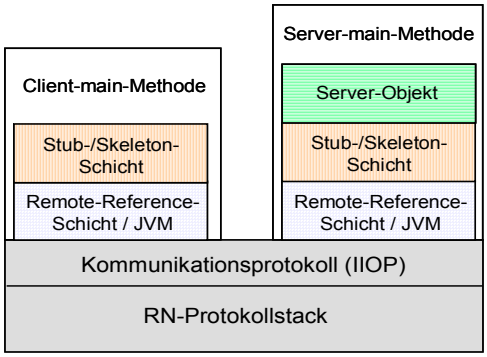


Abbildung 3.8: Java RMI Architektur

Stub/Skeleton-Schicht: realisiert Schnittstelle des Serverobjekts auf beiden Seiten. Client übergibt Aufruf an Client-Stub. Stub übernimmt Marshalling der Daten und gibt Aufruf an Remote-Reference-Schicht zur Übertragung. Server-Skeleton nimmt Aufruf an, führt Demarshalling der Daten durch und übergibt Aufruf an Server-Objekt.
Remote-Reference-Schicht: in Java Virtual Machine (JVM) integriert. Es ist eine Bibliothek, die anhand von Objektreferenzen Serverobjekte lokalisiert und die Aufrufe weiterleitet. Außerdem zusätzliche Funktionalität zur Aktivierung von Serverobjekten.
Kommunikationsprotokoll: früheres JRMP (Java Remote Method Protocol) wegen Interoperabilität durch CORBA-Protokoll IIOP ersetzt. IIOP setzt auf Transportprotokoll des RN auf und koordiniert die Kommunikation zw. den JVMs auf Client- und Serverseite.

Schnittstelle der Objekte
Methoden eines RMI-Serverobjekts werden mit ihren Parametern in einer Schnittstellenbeschreibung definiert. Schnittstellenbeschreibung in Java, da Client und Server auch Java unterstützten. Jedes Serverobjekt erhält eine Java-Schnittstelle, in der die Methodensignatur und alle benötigten Datentypen definiert werden. Vorteile der homogenen Java-Philosophie:
- *geringe Komplexität von Java-RMI-Anwendungen*: keine extra Schnittstellensprache, Java-Datentypen, Java-Methoden, einfachere Programmierung.
- *Unterstützung von Call-by-Value-Semantik* für entfernten Methodenaufruf: beliebige komplexe Objektstrukturen können serialisiert (object serialization), über Netz übertragen und auf Empfängerseite als Objektstruktur wiederhergestellt werden.

Entwicklung RMI-Anwendung
javac: Compiler der Java-Plattform. Übersetzt Klassen und Schnittstellen in Bytecode.
rmic: RMI-Compiler. Generiert aus der Schnittstellendefinition den Client-Stub und Server-Skeleton sowie alle explizit definierten Datentypen (werden an Client- bzw. Servercode angebunden). Alle Klassen der Anwendung werden auf Client und Server verteilt. Werkzeuge: JDK, J2SE, J2EE

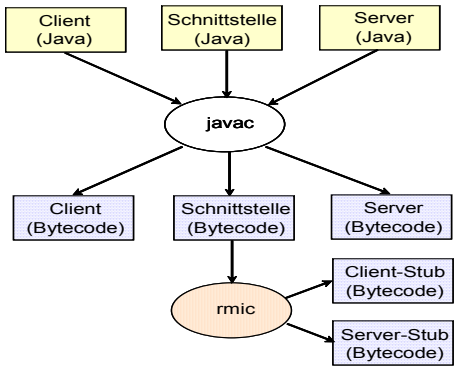


Abbildung 3.9: Entwicklung einer RMI-Anwendung

Kommunikationsablauf
Zur Durchführung eines entfernten Aufrufs auf einem Serverobjekt benötigt ein Client dessen Objektreferenz. Objektreferenz enthält alle Informationen zur Lokalisierung des Serverobjekts im Netz. Java RMI verwendet als Objektreferenzen einfache URLs (Uniform Resource Locator). Referenz enthält den Namen des Rechners, auf dem das Objekt liegt, die Portnummer des Servers und einen Objektidentifikator des Serverobjektes.
Zur Veröffentlichung der Objektreferenzen stellt die Java-Plattform einen einfachen Namensdienst zur Verfügung: RMI Registry. Andere Namensdienste (z.B. JNDI - Java Naming and Directory Interface) ebenfalls verwendbar.

Bei der Initialisierung und Veröffentlichung eines Serverobjekts wird die Objektreferenz erzeugt und unter einem festen Namen in der Registry angemeldet (~> Service Export). Clients können über den Namen die Objektreferenz von der Registry anfordern (~> Service Import).

Java RMI und WWW
Interaktion mit WWW-Server, dynamische Anfragen (z.B. für Investment-Informationen auf Kontenservern), Rückaufrufe des Servers bei Client-Objekten (z.B. für Parametereingaben).

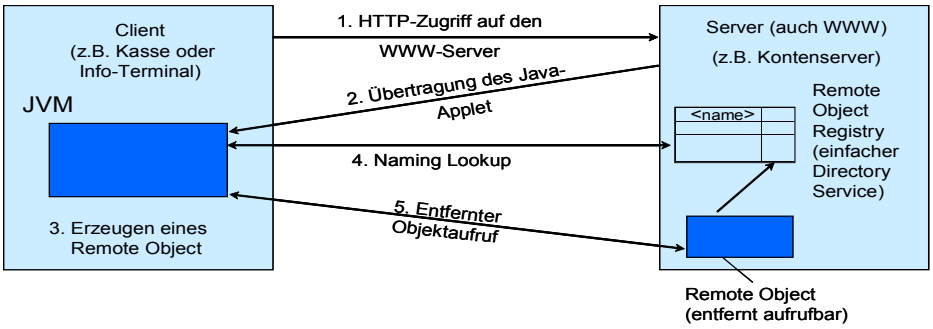


Abbildung 3.10: Java, RMI und WWW

RMI und Sicherheit
Sicherheitsprobleme bei Java RMI:
- verteilte Objekte sind über das Netz generell angreifbar,
- zusätzlich sicherheitskritisch ist der Mechanismus des dynamischen Nachladens von Klassen zur Laufzeit (es muss beim Nachladen der Klassen sichergestellt werden, dass ihre Methodenimplementierung keine Sicherheitslücken bewirkt).

Java RMI nutzt Sicherheitsmodell des Java-Standards („Sandbox"-Modell). Sicherheitskritische Anwendungen erhalten einen Security-Manager: hält Regeln (Policies) zur Zugriffsverwaltung. Die Regeln legen Bereich („Sandbox") fest, auf dem die Anwendung arbeiten darf. Zugriffsrechte betreffen beliebige Ressourcen, z.B. Sockets (Verbindungsauf-/abbau) oder Dateien (Lese-/Schreibrechte). Policies werden in Policy-Dateien abgelegt und bei Initialisierung dem Security-Manager übergeben. Security-Manager ist optional, wenn Client und Server auf gleichem Rechner, dagegen obligatorisch, wenn auf verschiedenen Rechnern.

3.4.3 Programmentwicklung mit Java RMI (Beispiel)

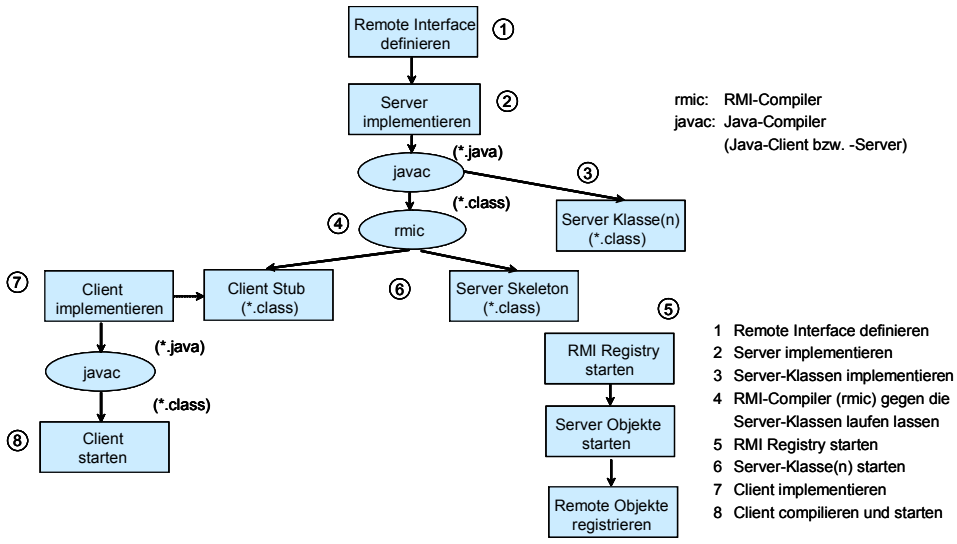


Abbildung 3.11: Programmentwicklung Java RMI

Entwicklungsprozess

1. Remote Interface definieren

- Jedes Server-Objekt, das seine Methoden entfernten Client-Objekten verfügbar machen soll, muss diese Methoden in einem Interface deklarieren, welches wiederum direkt oder indirekt von java.rmi.Remote abgeleitet wird.
- Die Methoden-Deklarationen müssen die Exception java.rmi.RemoteException oder einer ihrer Elternklassen (z.B. java.io.IOException, java.lang.Exception) spezifizieren.

Beispiel:

```
public interface Addition extends java.rmi.Remote
{
        public int berechneSumme (int a, int b)
                throws java.rmi.RemoteException;
}
```

oder:

```
import java.rmi.*

public interface Addition extends Remote
{
        public int berechneSumme (int a, int b)
                throws RemoteException;
}
```

2. Server implementieren

- Die Server-Klasse muss spezifizieren, welche Remote Interfaces implementiert werden; die Klasse sollte in jedem Fall von java.rmi.server.UnicastRemoteObject abgeleitet sein:

```
import java.rmi.server.UnicastRemoteObject;
public class AdditionServer extends UnicastRemoteObject
                implements Addition
{
```

- Die Imlementierung muss einen Konstruktor für das Server Objekt anbieten:

```
private String name;
public AdditionServer (String s) throws RemoteException
}
        super  ();
        name  = s:
}
```

- Die in dem Interface oder den Interfaces angegebenen Methoden, die von einem Client aufgerufen werden, müssen implementiert werden:

```
public int berechneSumme (int a, int b) throws RemoteException
{
        return (a + b);
}
```

- Ein Security Manager muss erstellt werden:

```
public static void main (String args [])
{
   System.setSecurityManager (new RMI.securityManager ());
```

- Mindestens ein Server-Objekt muss instanziiert werden:

```
try  {
   AdditionServer obj = new AdditionServer ("Additions-Server");
   System.out.println ("Additions-Server ist bereit.");
```

- In der Remote Object Registry müssen sich die angelegten Objekte anmelden:

```
   Naming.rebind ("//10.0.0.1/AdditionsServer", obj);
   System.out.println ("Additions-Server in der Registry angemeldet.");
} catch (Exception e)   {
   System.out.println ("AdditionServer error: " + e.message ());
   e.printStackTrace  ();
  }
}
```

3. Server-Klassen compilieren

```
>javac *.java          Addition.class
                      AdditionServer.class
```

4. RMI-Compiler gegen die Server-Klassen laufen lassen

```
>rmic AdditionServer  AdditionServer_Stub.class
                      AdditionServer_Skel.class
                      (Erzeugen Stub- und Skeleton-Prozeduren)
```

5. RMI Registry starten

```
>rmiregistry
```

6. Server-Klasse(n) starten

```
>java AdditionServer
```

Programmcode (Server)

```
//Addition.java
import java.rmi.*
public interface Addition extends Remote
{
public int berechneSumme (int a, int b)
        throws RemoteException;
}
// AdditionServer.java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.*;
public class AdditionImpl extends UnicastRemoteObject implements Addition  {
private String name;
public AdditionImpl (String s) throws RemoteException  {
        super ();
        name = s;
}
public int berechneSumme (int a, int b) throws RemoteException  {
        return (a + b);
}
public static void main (String args []) {
    System.setSecurityManager (new RMISecurityManager ( ));
    try  {
        AdditionImpl obj = new AdditionImpl ("AdditionServer");
        System.out.println("Additions-Server ist bereit.");
        Naming.rebind("//10.0.0.1/AdditionServer", obj);
         System.out.println
                ("Additions-Server in der Registry angemeldet.");
    }
    catch (Exception e)  {
    System.out.println
        ("AdditionServer error: " + e.getMessage ());
    e.printStackTrace ();
    }
  }
}
```

7. Client implementieren

```
import java.rmi.*;
public class AdditionClient  {
      public static void main (String args [ ])  {
          if (System.getSecurityManager () == null)
                System.setSecurityManager
                        (new RMI.securityManager ());
      try { . . .
```

- Referenz auf das Server-Objekt lesen:

```
Addition obj = (Addition)Naming.lookup
                        ("//10.0.0.1/AdditionServer");
```

- Aufrufen der Methode(n) des Server-Objektes:

```
int   a = 5;
```

```
int    b = 7;
int result = obj.berechneSumme (a, b);
System.out.println ("Ergebnis: " + result);
catch (Exception e) {
        System.out.println (e.getMessage ());
        e.printStackTrace ();
}
```

Programmcode (Client)

```
//AdditionClient.java
import java.rmi.*;
public class AdditionClient {
    public static void main (String args []) {
        if (System.getSecurityManager () == null)
            System.setSecurityManager(new RMISecurityManager ());
        try {
    Addition obj = (Addition) Naming.lookup
            ("//10.0.0.1/AddditionServer");
        int    a = 5;
        int    b = 7;
        int result = obj.berechneSumme (a, b);
        System.out.println ("Ergebnis: " + result);
    }
    catch (Exception e) {
        System.out.println (e.getMessage ());
        e.printStackTrace ();
    }
  }
}
```

8. Client compilieren und starten

```
> javac AdditionClient.java
> java AdditionClient
```

3.5 Nachrichtenorientierte Middleware (Messaging)

3.5.1 Architektur MOM (Message Oriented Middleware)

Nachrichtenorientierte Middleware

Basis: nachrichtenorientiertes Modell, asynchroner Austausch von Nachrichten zwischen Prozessen. Bezeichnungen: Messaging, Message Queueing, Message Oriented Middleware (MOM). I.d.R. Einsatz Warteschlangentechnik (WS):

- Sender stellt Nachricht in WS des Empfängers.
- Sender und Empfänger wirken unabhängig voneinander.
- Sender kann weiterarbeiten, ohne Kenntnis, ob Nachricht vom Empfänger abgeholt wurde.
- Empfänger kann Nachricht zu beliebigem Zeitpunkt aus WS abholen.

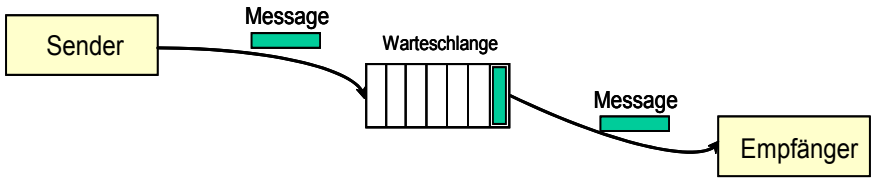


Abbildung 3.12: Warteschlangentechnologie

Nachrichtenorientierte Kommunikationsform ist asynchron, nicht-blockierend.

Architektur

Komponenten: Nachrichten (Messages), Warteschlangen (Queues), WS-Verwalter (Manager).

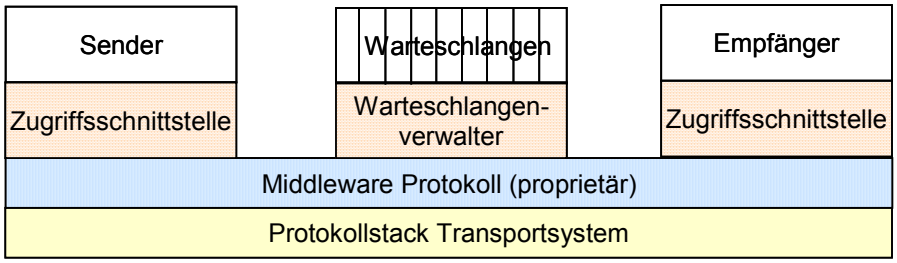


Abbildung 3.13: Architektur nachrichtenorientierter Middleware

WS-Verwalter verwaltet mehrere WS für unterschiedliche Empfänger ~> zuständig für
- Zuordnung von Nachrichten zu Warteschlangen sowie für die Benachrichtigung der Empfänger über Eintreffen der Nachricht.
- Initialisierung und Überwachung der WS,
- Zugriffsschnittstelle und Übertragungsprotokoll.

Schreiben / Lesen in WS erfolgt immer *lokal*, d.h. Sender bzw. Empfänger und WS auf gleichem Rechner. Falls *remote*, dann über WS-Verwalter (als Router): führt Vermittlung und Wandlung der Datenformate durch.

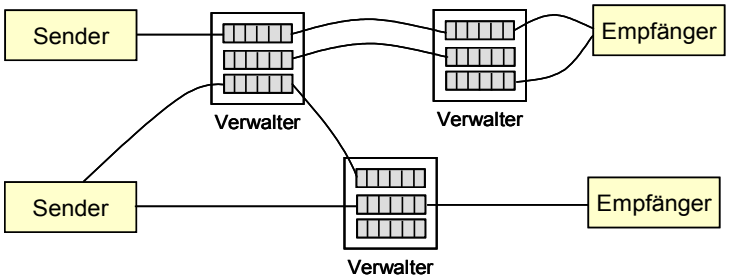


Abbildung 3.14: Warteschlangeninfrastruktur (Beispiel)

I.allg. keine garantierte Auslieferung, insbes. bei Ausfall einer Komponente.
Falls garantierte Auslieferung gefordert: Anwendung des Konzepts persistenter WS. Nachrichten hierbei bis zur Auslieferung persistent abgespeichert (Dateisystem, DB). Auslieferung erfolgt, wenn Empfänger erreichbar ist (ohne zeitliche Zusage).

3.5.2 Programmiermodelle

Modelle für asynchrone Nachrichtenübermittlung
Point-to-Point-Modell (2 Partner): Standard.
Kommunikationsform: asynchron (nicht-blockierend), Programmierparadigma: beliebig.
Zwei Erweiterungen:
Request-Reply-Modell:
Simulation einer synchronen Kommunikation über asynchrone Infrastruktur, d.h. Sender bleibt bis zum Erhalt einer Antwortnachricht blockiert.

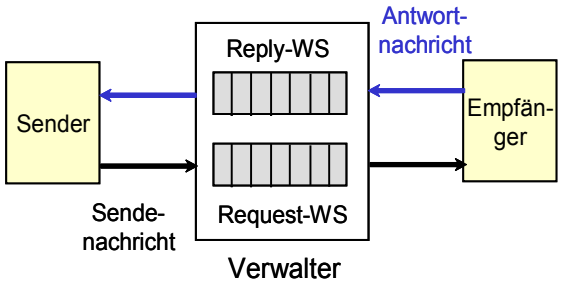


Abbildung 3.15: Request-Replay-Modell

Realisierungsvarianten:

1. WS-Verwalter legt temporär eine *WS für Antwortnachricht* an, aus der Sender die Antwort abrufen kann ~> engere Kopplung als normale asynchrone Kommunikation.
   Nach Entnahme der Nachricht wird temporäre WS gelöscht.
2. Verwendung von *Identifikatoren* zur Kennzeichnung zusammengehöriger Nachrichten. Anfragenachricht (Message-ID) und Antwortnachricht (Corellation-ID) über gleiche WS. Auch mehrere parallel Verbindungen über eine Queue möglich. Sender blockiert, bis Antwortnachricht eingetroffen.

Publish-Subscribe-Modell:

Simuliert ein Abonnentensystem, in dem Sender Nachrichten veröffentlichen, die interessierte Empfänger abbonieren können.

Komponenten: Publisher (Veröffentlichung von Nachrichten), Subscriber (Abonnierung von Nachrichten), Vermittler (Broker, Koordination Nachrichtenvermittlung).

Subscriber abonnieren beim Broker Nachrichten zu einem Thema (Topic). Publisher übergibt Broker eine Nachricht zum Thema. Broker übernimmt Verteilung der Nachricht, z.B. bestimmter Empfänger, gleichmäßige Verteilung udgl.

Publisher und Subscriber sind vollständig unabhängig. Verbindung nur über Vermittler (Subscriber und Publisher kennen sich nicht).

Anwendungsbeispiele: Börsendienst, RSS-Fedds (Pull-Dienste, Web 2.0)

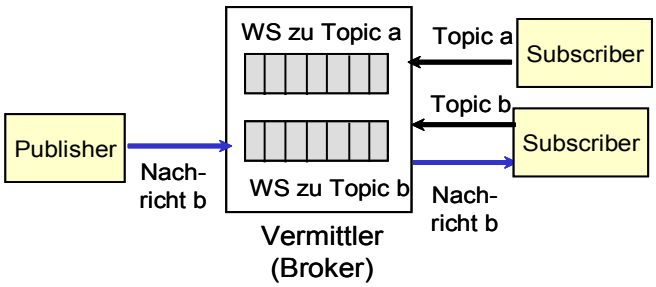


Abbildung 3.16: Publish-Subscribe-Modell

3.5.3 Java Message Service (JMS)

JMS-Standard

JMS (Java Message Service): Standard für eine einheitliche Java-Zugriffsschnittstelle auf nachrichtenorientierte Middleware. Entwicklung durch Sun Microsystems als integraler Bestandteil der Java-Plattform.

JMS-Standard definiert die Schnittstellen, über die Java-Anwendungen die Dienste eines beliebigen MOM-Servers nutzen können.

JMS definiert *2 Basiskonzepte*: Nachrichten und administrative Objekte.

Nachricht: festgelegtes Format, zum Versenden über den JMS-Provider bestimmt.

Administrative Objekte: sind von den JMS-Providern bereitzustellen, um JMS-Clients Informationen für den Verbindungsaufbau zu liefern.

JMS-Standard definiert Schnittstellen für 2 administrative Objekte:

- *ConnectionFactory*: enthält alle Informationen, die ein JMS-Client zum Aufbau der Verbindungen zu einem JMS-Provider benötigt und bietet über eine Schnittstelle Methoden zur Initialisierung einer Verbindung.
- *Destination*: repräsentiert eine konkrete Warteschlange des JMS-Providers. Das Objekt wird von einem JMS-Client zum Einstellen und Auslesen von Nachrichten verwendet.

Administrative Objekte werden in einem Namensdienst veröffentlicht und können bei Bedarf von JMS-Clients angefordert werden.

Schnittstellendefinition von JMS unterstützt folgende nachrichtenorientierte Modelle:

- Point-to-Point mit Unterstützung synchroner Antwortnachrichten (Request-Reply) und
- Publish-Subscribe-Modell.

Allgemeines JMS-Modell
Kommunikationsablauf: Verbindungsaufbau und Kommunikation über JMS. Erster Schritt: Aufbau einer Verbindung zw. JMS-Client und JMS-Provider. Standard definiert Schnittstellen zu 4 Objekten, über die ein JMS-Client die Dienste seines JMS-Providers nutzen kann:
- *Connection*: repräsentiert eine konkrete Verbindung zu einem JMS-Provider. Über diese Schnittstelle kann Verbindung gestartet oder terminiert werden.
- *Session*: repräsentiert den Kontext, in dem Nachrichten erzeugt und verarbeitet werden. Optional Transaktionsunterstützung.
- *MessageProducer*: um Nachrichten an eine Warteschlange zu schicken.
- *MessageConsumer*: um Nachrichten von Warteschlangen zu empfangen.

JMS-Provider muss für eine geeignete Implementierung der Schnittstellen sorgen. Ablauf kann je Modell (Point-to-Point, Publish-Subscribe) modifiziert sein.
Einsatz: nachrichtenorientierte Middleware, Nachrichtenverteilung (1:n-Beziehungen).

JMS-Client muss für Verbindungsaufbau und Kommunikation folgende Schritte durchführen:
- Verbindungsaufbau: JMS-Client fordert vom Namensdienst die administrativen Objekte eines JMS-Providers an: *ConnectionFactory* und *Destination*.
  An der *ConnectionFactory* fordert JMS-Client eine aktive Verbindung (*Connection*) zum JMS-Provider an. Das Objekt *Destination* stellt eine WS des JMS-Providers dar.
- Kommunikationsumgebung: Zur Steuerung der Kommunikation wird an der Verbindung eine Sitzung (*Session*) eröffnet. Zusätzlich wird ein *MessageProducer* initialisiert und an das WS-Objekt *Destination* gebunden.
- Senden einer Nachricht: JMS-Client erstellt Nachricht (*Message*) zum Versenden und übergibt sie dem *MessageProducer* zum Einstellen in die WS.
- Empfangen einer Nachricht: JMS unterstützt synchrones und asynchrones Empfangen.
  - **synchron**: JMS-Client solange blockiert, bis M*essageConsumer* eine Nachricht aus seiner WS empfangen hat.
  - **asynchron**: JMS-Client installiert einen *MessageListener*. Falls Nachricht in WS, wird JMS-Client über den *MessageListener* vom JMS-Provider informiert und kann Nachricht entnehmen.

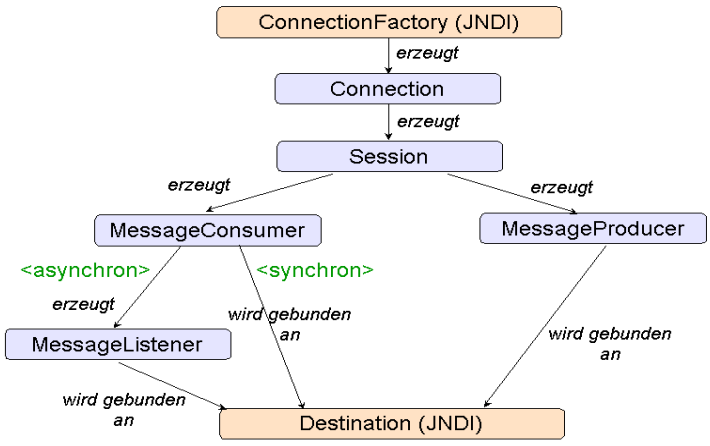


Abbildung 3.17: Allgemeines JMS-Modell

4 Synchronisation und Koordination

4.1 Zeit in verteilten Systemen

In verteilten Systemen müssen Entscheidungen oft auf Basis unvollständiger Informationen gefällt werden, unter anderem fehlt eine exakte einheitliche Zeitbasis, wie sie durch eine gemeinsame physikalische für Prozesse auf einem einzelnen Netzwerk-Knoten gegeben ist. Sie lässt sich in verteilten Systemen nur mit einer Genauigkeit erreichen, die durch die Varianz der Netzlaufzeiten bestimmt ist.
Oft ist das Auftreten von Ereignissen eng mit dem Zeitpunkt korreliert. Außerdem wird die Zeit benötigt, um kausal abhängige Ereignisse in der Reihenfolge ihres Auftretens zu ordnen. Aufgabe: Synchronisation der Uhren, einheitlicher Zeitmaßstab, Abstimmung mit Referenzzeit. Zwei Ansätze: Verteilung der Zeitinformation im verteilten System ist hierarchisch unter Verwendung dedizierter Zeitserver und **physikalischer Uhren** oder über verschiedene Protokolle mit **logischen Uhren** ohne Verwendung dedizierter Server möglich.

4.1.1 Synchronisation physikalischer Uhren

Uhren in einem verteilten System können intern gegeneinander oder gegen eine externe Zeitquelle abgeglichen werden. Interne Synchronisation: Abgleichen der internen Uhren; Externe Synchronisation: mit externer Refernzzeit. Zu beachten ist, dass die internen Uhren der meisten Netzwerkknoten eine Drift aufweisen, die +/-1ms/Tag betragen kann, was auf einem Rechner bereits viele Tausend Takte und damit tausende Instruktionszyklen bedeutet ~> Anforderung einer *Universalzeit (UTC) und* Abgleichen der internen Uhren mit der Universalzeit (UTC), insbes. in verteilten Systemen, die einen einheitlichen Zeitmaßstab voraussetzen.

Die **astronomische Zeit** ist bedingt durch die sich verlangsamende Erdrotation zu ungenau. Externe Quellen, wie die **Internationale Atomzeit** (TAI, 1958) oder die **Koordinierte Universalzeit** (UTC, 1967) liefern eine driftfreie Zeit (Schaltzeit), allerdings oft nur mit einer Genauigkeit im Millisekundenbereich. UTC wird per Kurzwelle oder Satellit (GEOS, GPS) ausgestrahlt. Weltweit garantieren 13 sog. primäre Cäsium-Atomuhren die 1972 eingeführte koordinierte Universalzeit UTC. Übergangsperioden in Cäsium-Atomen –> durch Mikrowellenbestrahlung der Atome werden die Schwingungen der Atome meßbar und zählbar
(9 162 631 770 Schwingungen ergeben eine Sekunde).
Die Kompensation der Zeitdrift ist ein komplexer Angleichungsvorgang.

Bei der Synchronisation zwischen interner Uhr und UTC-Zeitdienst wird allgemein auf ein *Client/Server-Modell* gesetzt, in dem verschiedene Hierarchieebenen existieren können.

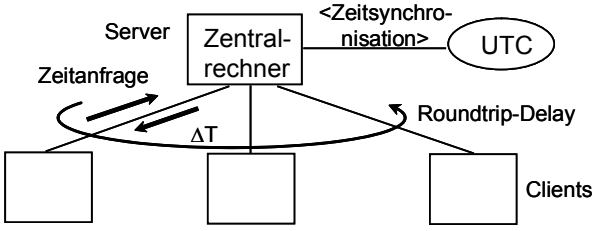


Abbildung 4.1: Uhrensynchronisation (Cristians Method)

Cristians Methode (1984) verwendet einen extern synchronisierten, **passiven Server**, der Anfragen aller Clients beantwortet. Der Client misst hierbei den so genannten Roundtrip-Delay (Zeit zwischen Absenden der Anfrage und Annahme der Server-Antwort) und gleicht unter Verwendung dieser Information seine lokale Zeit mit der des Servers ab. Die Genauigkeit hängt von der Netzlaufzeit ab**. Nachteilig:** Server bildet einen Single-Point-of-Failure.

Der **Berkeley-Algorithmus** (1989) arbeitet mit einem **aktiven Server**, der regelmäßig seine Clients abfragt, den Roundtrip-Delays ermittelt und mit statistischen Methoden deren relative Zeitdifferenz berechnet und diese den Clients mitteilt, welche sie auf ihre lokale Zeit aufaddieren. Eine Genauigkeit von +/-10 ms lässt sich im LAN unabhängig von den Laufzeiten erreichen. Auch hier ist die Schwachstelle die zentralisierte Architektur.
Das **Network Time Protocol** (NTP, 1991) realisiert einen verteilten, dezentral organisierten Zeitdienst im Internet (i.allg. auf Basis UDP). NTP arbeitet mit einem extern synchronisierten, virtuellen Server (UTC-Zeitquelle) auf der 0. Ebene, der eine Reihe von Primärservern im Stratum 1 synchronisiert, die wiederum Sekundärserver im Stratum 2 synchronisieren. Auf jene schließlich haben die Clients Zugriff.
Die Genauigkeit nimmt mit jeder Ebene ab. Server einer Ebene gleichen ihre Zeit untereinander immer wieder ab (symmetrischer Modus), werden per Multicast von einer höheren Ebene synchronisiert (Multicast-Modus) oder arbeiten im Procedure-Call-Modus, in dem sie analog zu Cristians Methode direkt übergeordnete Zeitserver kontaktieren.

4.1.2 Logische Zeit und logische Uhren

Für die Feststellung einer zeitlichen Reihenfolge benötigt man keine physikalische Zeit. Lediglich eine **zeitliche Ordnung** ist erforderlich. Diese logische Zeit muss nicht kontinuierlich sein und nicht gleichmäßig oder stetig verstreichen. Logische Uhren erreichen eine sehr hohe Genauigkeit und benötigen wesentlich weniger Kommunikationsaufwand für ihre Arbeit als die Synchronisation der physikalischen Zeit.
Zwei Versionen sind die **Lamport-Zeit** (Leslie Lamport, 1978), die einen **Zeitbildungsalgorithmus** beschreibt und die **Vektorzeit** (1988), einem alternativen Ansatz für logische Uhren, der auf **lokalen Ereigniszählern** basiert. Beide erzeugen Zeitstempel für Ereignisse, anhand derer man eine Kausalitätsbeziehung erkennen und die Ereignisse ordnen kann.

4.2 Kollaborative Algorithmen

Es sind Algorithmen in verteilten Systemen, deren Prozesse gemeinsam eine Aufgabe erfüllen. Die Teilprozesse erledigen jeweils eine Teilaufgabe und können nur mit ihren direkten Nachbarn kommunizieren. Den Prozessen ist die Gesamttopologie nicht bekannt. Jedoch sollen alle Prozesse in der Lage sein, das globale Endergebnis bereitstellen zu können. Bekannte Algorithmen: Heartbeat, Probe/Echo.

4.2.1 Heartbeat-Algorithmen

Algorithmus („Herzschlag“-Algorithmus) teilt sich in zwei Phasen, die einander abwechseln, bis das Ergebnis vorliegt.
1. In einer **Expansionsphase** werden Informationen zum nächsten Nachbarn gesendet,
2. In einer **Kontraktionsphase** werden von diesen die neuen Informationen eingesammelt.

Nach lokaler Verarbeitung wiederholen sich diese Phasen (sog. Runden), bis das Ergebnis berechnet ist (konvergente Algorithmen erforderlich).
Anwendung: paralleles Sortieren, Matrizenberechnung, Bildverarbeitung, Routingverfahren (z.B. Distance-Vector-Routing im Internet).

4.2.2 Probe/Echo-Algorithmen

Algorithmus arbeitet mittels eines **virtuellen Broadcasts** und wird daher vor allem in Netzen verwendet, die keinen Broadcast unterstützen. Grundlage des Verfahrens ist das Weiterreichen von PROBE- und ECHO-Mitteilungen an jeweils alle Nachbarknoten.
Algorithmus:
Ein Knoten ist der Initiator und schickt PROBE an alle seine direkten Nachbarn (Flooding). Diese schicken wiederum an alle Nachbarn, außer dem, von dem sie das PROBE erhielten, eine PROBE und warten auf ein ECHO. Erhält ein Knoten von einem Nachbarn, an den er

PROBE geschickt hat, ebenfalls ein PROBE, so wird die Kante aus dem Graphen eliminiert. Besitzt ein Knoten keine weiteren Nachbarn mehr (Blatt), so schickt er an den Knoten, von dem er das erste PROBE erhielt, ein ECHO zurück. Alle anderen Knoten warten, bis sie von jedem verbliebenen Nachbarn ein ECHO erhalten haben, verpacken die Information in ein neues ECHO und schicken das ebenfalls an den Knoten, von dem sie das erste PROBE erhalten haben. Schließlich erhält der Initiator die Gesamtinformation, wenn er von allen Nachbarn ein ECHO bekommen hat.
Vorteil gegenüber Heartbeat: besseres Konvergenzverhalten, geringere Nachrichtenanzahl.
Anwendung: Erkennen von Verklemmungen (deadlocks), Überprüfung der Terminierung verteilter Anwendungen, Routing (z.B. Link State Algorithmus).

4.3 Election-Algorithmen

Election-Algorithmen (Auswahl-Algorithmen) dienen zur Auswahl eines ausgezeichnete Prozesses z.B. als **Koordinator** aus einer Menge von Prozessen. Anwendung bei Ausfall des vorherigen Koordinators oder bei wechselseitigen Ausschluss.
Election: Bestimmung eines Extremwertes, z.B. auf einer Ordnung beteiligter Prozesse.
Algorithmen (Auswahl):
Bully-Algorithmus (Garcia/Molina, 1982): Alle Prozesse tragen eindeutige Identifikation ID und kennen die der anderen Prozesse. ID‘s lassen sich ordnen, Prozessstatus unbekannt. Bei Ausfall des bisherigen Koordinators ist der Prozess mit der höchsten ID zu finden, der noch aktiv ist --> dieser wird dann zum neuen Koordinator. Einfacher Algorithmus. Nachteilig: hohe Nachrichtenkomplexität.
Ringbasierter Election-Algorithmus (Chang/Roberts, 1979): alle Prozesse werden in einem gerichteten Ring angeordnet und können somit nur sequentiell aktiv werden. „Election“-Nachricht mit ID wird als Token weitergereicht, ID‘s verglichen. Auswahl des Prozesses mit höchster ID. Verfahren kommt mit weniger Nachrichten aus. Aber: Election im Ring ist ein sequentielles Verfahren und damit wenig effizient (Ziel: paralleles Verfahren).
Election auf Bäumen (Mattern, 1989) ist ein dritter Weg, bei dem das Probe/Echo-Prinzip direkt auf Baumtopologien angewendet wird. 3 Phasen: Explosionsphase (Probe), Kontraktionsphase (Echo), Informationsphase. Auswahl des Prozesses mit höchster ID.

4.4 Verteilter wechselseitiger Ausschluss

Ein von den Mechanismen von Multitasking-Betriebssystemen her bekanntes Verfahren dient der Regelung des **Zugriffs auf gemeinsame Ressourcen**: **wechselseitiger Ausschluss** und **kritische Regionen**. Oft ist in einem verteilten System kein gemeinsamer Speicher vorhanden, auf dem atomare Operationen möglich sind. Hier können Semaphore oder Monitore nicht eingesetzt werden..
Grundlegend sind die Anforderungen der Sicherheit (max. ein Prozess befindet sich in der verteilten kritischen Region), der Lebendigkeit (es darf kein anderer Prozess den Eintritt in die kritische Region beantragen; dies impliziert Verklemmungsfreiheit (deadlock free) und kein Verhungern (no starvation) und optional noch eine Ordnung über die Reihenfolge, mit der Prozesse in die kritische Region eintreten (z.B. nach der happened-before-Relation).
Verschieden Algorithmen:
Der **zentralisierter Algorithmus** basiert auf der Einrichtung eines **zentralen Servers** zur Koordination des Zugriffs (Coulouris, Dollimore, Kindberg, 1994). Clientanfrage: falls Ressource frei, dann Zugriff erlaubt, sonst Anforderung in Warteschlange eingereiht. Dies garantiert Sicherheit und Lebendigkeit, schafft aber keine Ordnung. Die Implementierung ist einfach und erzeugt wenig Nachrichten, ist aber fehleranfällig (single-point-of-failure).
Basierend auf der **logischen Zeit von Lamport** ist ein **verteilter Algorithmus** möglich. Alle Prozesse entscheiden gemeinsam über den Eintritt eines einzelnen in die kritische Region. Hier sind alle drei Bedingungen erfüllt, wobei aber jeder ausfallende Prozess den gesamten

Ablauf zum Absturz bringen kann, wenn er nicht mehr reagiert, womit das Verfahren sehr fehleranfällig ist.
Als dritte Möglichkeit kann auch hier ein **logischer Ring** dienen, in dem je ein **Token** mit dem Zugriffsrecht für je einen kritischen Bereich kreist und während eines Zugriffes so lange nicht weitergegeben wird, bis der jeweilige Prozess den kritischen Bereich verlässt.
Alle drei Verfahren sind nur für sehr zuverlässige Systeme zu empfehlen. Die Zugriffsrechte sollten besser an der Datenverwaltungsstelle geregelt werden, evtl. per Sperrmechanismus (z.B. TSL-Instruktion, Sperrvariable `turn`).

4.5 Transaktionen

Soll der wechselseitige Ausschluss auf einer höheren Ebene (Programmierebene) gelöst werden, können **Transaktionen** verwendet werden, die die zugrunde liegende Synchronisationsproblematik verbergen. Zugehörige Operationen werden hier in einer Sequenz zusammengefasst, die nach außen hin als geschlossene Einheit wirkt. (z.B. eine Anfrage eines Clients ~> entspricht einer atomaren Operation).
Transaktionen besitzen sog. **ACID-Eigenschaften**:
Atomarität (atomicity): Es werden alle oder keine Operation ausgeführt. Eine Transaktion ist erfolgreich (`commit`) oder nicht (`abort`). Zwischenzustände nach außen nicht sichtbar.
Konsistenz (consistency): Die Transaktion überführt das System von einem Konsistenzzustand in einen anderen. Zwischenzeitliche inkonsistente Zustände können durchlaufen werden, aber sind nach außen nicht sichtbar.
Serialisierbarkeit (isolation): Eine noch nicht abgeschlossene Transaktion gibt Ergebnisse nicht an andere Transaktionen weiter. Dadurch nebenläufige Ausführung ~> wird zur seriellen Ausführung.
Dauerhaftigkeit (durability): Wenn Transaktionen erfolgreich beendet sind, sind die Ergebnisse dauerhaft, auch bei nachfolgenden Systemfehlern.

Basis-Primitive:
- **Commit**: erfolgreicher Abschluss einer Transaktion, neuer konsistenter Zustand.
- **Abort**: nicht erfolgreicher Abschluss einer Transaktion, Zurücksetzen auf den konsistenten Zustand vor Aufruf der Transaktion (sog Recovery).

Realisiert werden die Eigenschaften durch private Arbeitsbereiche, in denen Schattenkopien aller benötigten Objekte existieren, auf denen die Transaktion arbeitet. Bei einem abort wird der Bereich einfach gelöscht, bei einem commit die Originale von den Kopien überschrieben. Ebenfalls möglich ist die Führung einer Intentionenliste in Form eines write-ahead-logs. Dort werden alte und neue Werte geschrieben. Bei einem abort kann der Ausgangszustand anhand der Datei wiederhergestellt werden, bei einem commit werden die alten Werte gelöscht. Oft Einsatz von Datenbanksystemen.

Zwei-Phasen-Commit-Protokoll

Daten in VST über mehrere Rechner verteilt ~> erfolgreiches Beenden (`Commit`) einer Transaktion muss aber atomar unter allen beteiligten Prozessen ablaufen.
2-Phasen-Commit (Gray, 1978): Algorithmus zum Beenden einer verteilten Transaktion:
- Jeder Prozess entscheidet für seine Teiloperation über `Commit` oder `Abort`.
- Danach einigen sich alle Prozesse über ein Gesamtergebnis: falls ein Prozess mit `Abort`, ist auch Gesamtergebnis `Abort` ~> alle Teiloperationen müssen rückgängig gemacht werden, andernfalls `Commit` für alle ausführbar.
- Prinzip durch Zwei-Phasen-Commit-Protokoll implementiert:
  Abstimmungsphase (voting phase): Koordinator schickt `Commit`-Anfrage an alle TN, die mit `Yes` oder `No` an Koordinator antworten.

Abschlussphase (completion phase): Koordinator entscheidet auf Basis aller Antworten und sendet `DoCommit` oder `DoAbort` an alle TN (`Commit` ist zu quittieren).

- Nachteilig: blockierendes Protokoll. Abhilfe: Gesamtergebnis frühzeitig `Abort` gesetzt.

4.6 Verteilte Terminierung

Selbst wenn alle lokalen Prozesse in einem System terminiert sind, kann daraus nicht auf eine globale Terminierung geschlossen werden. Es können sich noch Nachrichten im Netz befinden, die bei ihrer Ankunft einen erneuten Start eines Prozesses bewirken.
Definition global terminiertes System:

Ein Prozess gilt als idle, falls er terminiert ist oder auf eine Nachricht wartet. Ein verteiltes System ist global terminiert, wenn alle Prozesse idle sind und keine Nachricht mehr unterwegs ist.

Verteilte Verfahren, die eine vollständige Terminierung erkennen oder bewirken, sind Probe/Echo- oder logische Zeit-Verfahren. Auch das Legen eines logischen Ringes durch alle Prozesse ist möglich. Außerdem existieren noch Schnappschuss-Verfahren. Ein Schnappschuss (snapshot) hält den globalen Zustand eines Systems fest. Dieser besteht aus den lokalen Zuständen aller Prozesse und allen unterwegs befindlichen Nachrichten. Er kann auch zur Erkennung von Verklemmungen bzw. zur verteilten Terminierung verwendet werden.

4.7 Verklemmung in verteilten Systemen

Eine Verklemmung entsteht, wenn zwei Prozesse auf eine Nachricht des jeweilig anderen warten.

Drei mögliche Strategien zum Umgang mit Verklemmungen:

- Erkennung (dedection): Erkennen und ggf. Auflösen.
- Vorbeugen (prevention): Programmstruktur verhindert generell die Verklemmungen.
- Verhinderung (avoidance, “Umgehung”, Vermeidung): Systemseitige Verteilung der Ressourcen soll Verklemmungen verhindern.

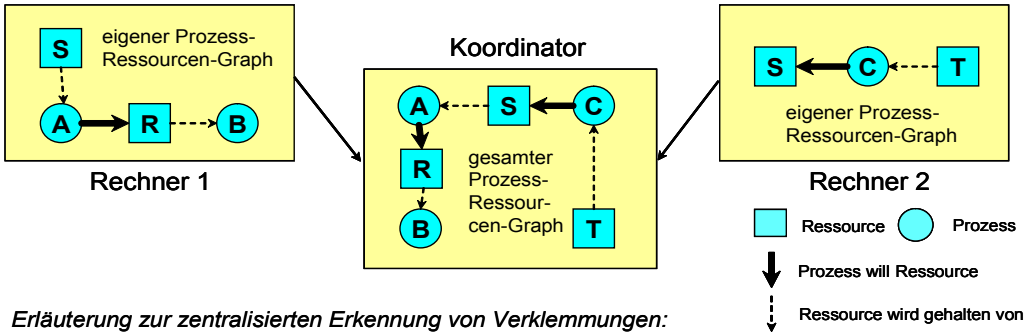
In einer lokalen Umgebung kennt das BS alle Ressourcen und Prozesse und kann handeln. In verteilten Systemen muss zuerst der globale Zustand des Systems ermittelt werden.

Zum **Erkennen von Verklemmungen** werden Prozess-Ressourcen-Graphen oder Wait-for-Graphen (mit Petri-Netzen für Erreichbarkeitsgraphen, Fallen, …) verwendet. Liegt ein Zyklus vor, muss er durch Terminierung eines Prozesses aufgebrochen werden. Zur Festlegung, welcher Prozess terminiert wird, ist ein Abstimmungsverhalten (Voting) erforderlich.
Zentralisierte Erkennung vs. verteilte Erkennung.

Zentralisierte Erkennung von Verklemmungen

* Jeder Computer verwaltet einen eigenen Prozess-Ressourcen - Graph.
* Jede Änderung an einer Kante des Graphen wird einem zentralen Koordinator mitgeteilt.
* Koordinator bildet einen Gesamtgraphen und kann so Zyklen (und damit Verklemmungen) erkennen.

Genereller Nachteil: Schwächen eines zentralisierten Systems.



Erläuterung zur zentralisierten Erkennung von Verklemmungen:

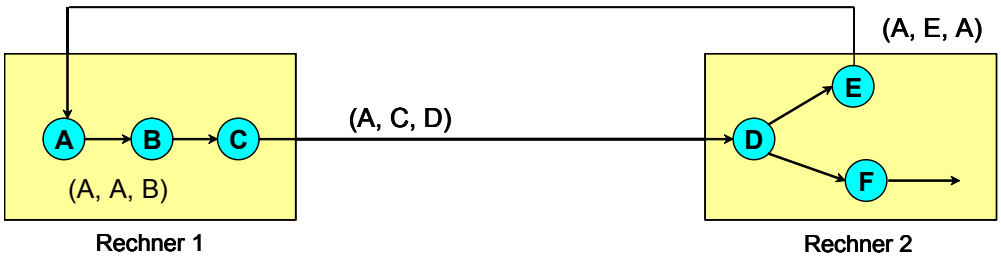
Rechner 1: Ressource S von A gehalten.
A benötigt Ressource R, die noch von B gehalten wird.

Rechner 2: Ressource T von C gehalten, C benötigt Ressource S, die von A gehalten wird.

Koordinator: falls B die Ressource R freigibt, kann A weitermachen und Ressource S freigeben
=> dann keine Verklemmung bzw. Blockierung.

Abbildung 4.2: Zentralisierte Erkennung von Verklemmungen

Verteilte Erkennung von Verklemmungen (Beispiel: Chandy-Misra-Haas-Algorithmus, 1983):



Syntax der Nachricht:
< blockierter Prozess, sendender Prozess, empfangender Prozess >

Legende:
B wartet auf Ressource, weil A
blockiert (hält C).
D bekommt Nachricht (A, C, D).
D schickt Nachricht an E und F die Nachricht (A, D, E) bzw. (A, D, F).
E schickt an A die Nachricht (A, E, A).
A will an B schicken: (A, A, B).
A erkennt sich selbst als blockierter Prozess --> Verklemmungserkennung.

Abbildung 4.3: Verteilte Erkennung von Verklemmungen

Algorithmus wird gestartet, wenn ein Prozess auf eine Ressource warten muss. Diesem wird dann eine Nachricht gesendet mit einem Tupel <*blockierender Prozess, sendender Prozess, empfangender Prozess*>. Blockiert der Empfänger ebenfalls, ändert er Feld 2 und 3 und schickt die Nachricht weiter. Kommt die Nachricht wieder beim ersten blockierenden Prozess an, so erkennt er sich selbst im Feld 1 und die Verklemmung wurde entdeckt. Im allgemeinen Fall terminiert dann der Initiator, um die Verklemmung aufzulösen.

Vorbeugen von Verklemmungen:
In der Regel ist dies eine Aufgabe des Systementwurfs / Systemimplementierung. Dazu sind bei der Programmgestaltung einige Dinge zu berücksichtigen:

- Jeder Prozess kann zu einem Zeitpunkt max. 1 Ressource halten.
- Prozesse müssen alle benötigten Ressourcen bereits zur Initialisierungszeit anfordern.
- Alle Prozesse müssen ihre gehaltenen Ressourcen freigeben, bevor sie neue anfordern.
- Ressourcen erhalten eine Ordnung. Anfragen erfolgen strikt nach dieser Ordnung, d.h. wer „hohe“ Ressourcen haben will, braucht auch alle „niedrigen“.

Gut lösbar bei Verwendung eines Transaktionssystems und Verfügbarkeit einer globalen Zeit.

4.8 Replikationen

In verteilten Systemen werden häufig große Mengen von Daten gemeinsam genutzt. Zur Effizienzsteigerung, Verfügbarkeit und Fehlertoleranz werden Caching und Replikation eingesetzt.

Caching ist die Speicherung häufig zugegriffener Daten in schnellen Zwischenspeichern und gut erreichbaren Netzknoten. Es werden **lokale Kopien** von entfernten Daten angelegt. Welche Daten zu kopieren sind, wird zur Laufzeit entschieden. Dabei werden **Ersetzungsstrategien** zur Sicherung von Effizient und Konsistenz verwendet.

Replikation findet i.allg. unabhängig von der aktuellen Zugriffshäufigkeit statt. Es handelt sich um das **bewusste Kopieren** bestimmter Daten. Das Anlegen von Replikaten kann auf zwei Ebenen erfolgen. In Client/Server und Peer-to-Peer Systemen werden Replikate durch Nachrichtenaustausch angelegt und modifiziert. In verteilten gemeinsamen virtuellen Speichern werden Speicherobjekte repliziert und verwaltet. Der Zugriff erfolgt dann von verschiedenen physikalischen Orten aus.
Folgende Anforderungen werden an Replikationsverfahren gestellt: schneller Zugriff (Effizienz), bei Serverausfällen muss ein anderer Replikationsserver zur Verfügung stehen (Verfügbarkeit, Fehlertoleranz), Clienten dürfen nur genau ein logisches Datenobjekt sehen, egal wie viele Replikate existieren (Transparenz) und es muss für Konsistenz gesorgt werden, wenn einzelne Replikate geändert werden.

Es gibt grob gesehen drei Vorgehensweisen.

Asynchrones Replikationsmanagement:

- Clientanfragen werden am „lokalsten“ Replikat bedient.
- Aktualisierung zwischen den einzelnen Replikaten nur periodisch durchgeführt.

Kausal geordnetes Replikationsmanagement:

- Veränderungen der Replikate erfolgt über eine kausale Ordnung
  --> somit Replikate kohärent.
- Falls auf Replikate nur von 1 Client aus zugegriffen wird, wird periodisch aktualisiert.

Synchrones Replikationsmanagement:

- Alle Replikate sind immer kohärent.
- Alle Veränderungen müssen total geordnet und atomar an allen Replikaten durchgeführt werden.

Synchrones Management ist sehr aufwendig und sollte nicht in Systemen angewendet werden, in denen häufig gelesen wird (viele Replikate, wie z.B. USENET).

5 Prozessmanagement

5.1 Prozesskonzept in autonomen Systemen

Prozess und Programm

Prozess := Abstraktion eines sich in Ausführung befindlichen Programms (Programm := Lines of Code). Gesamte Zustandsinformation der Betriebsmittel für ein Programm wird als Einheit angesehen und als Prozess (task) bezeichnet.

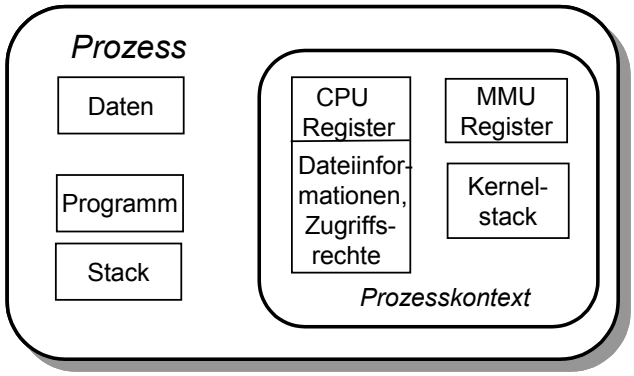


Abbildung 5.1: Zusammensetzung der Prozessdaten

Einprozess- und Mehrprozesssysteme

Einprozess-System: 1 Prozessor, 1 Task (hierbei immer nur 1 Prozess aktiv, die anderen sind blockiert und warten).

Mehrprozess-System: Erlaubt gleichzeitiges Ausführen mehrerer Programme (Prozesse):

falls 1 CPU: Mehrprogrammbetrieb (Multiprogramming),

falls mehrere CPU: mehrere Prozesse (Multiprozessing/Multitasking).

Multiprogramming (Mehrprogrammbetrieb): Mehrere Aufgaben werden gleichzeitig erledigt ~> Parallelverarbeitung erforderlich:

- Echte Parallelität: Mehrprozessorsysteme bzw. CPU und E/A-Prozessoren.
- Quasi- oder Pseudoparallelität (Nebenläufigkeit): Ein Prozessor kann immer nur 1 Programm abarbeiten. Nutzung I/O-Zeiten, um CPU anderen Programmen zuzuordnen.

Verteiltes System: Autonom arbeitende Systeme, die über Telekommunikation miteinander kooperieren ~> Synchronisation der Betriebsmittel erforderlich.

Prozessmodell

- Alle ausführbaren Programme eines Computers (incl. Betriebssystem) bestehen aus einer Anzahl sequentieller Prozesse.
- Prozess := ein in Ausführung befindliches Programm, einschließlich aktueller Wert des Programmzählers, der Register und der Variablen.

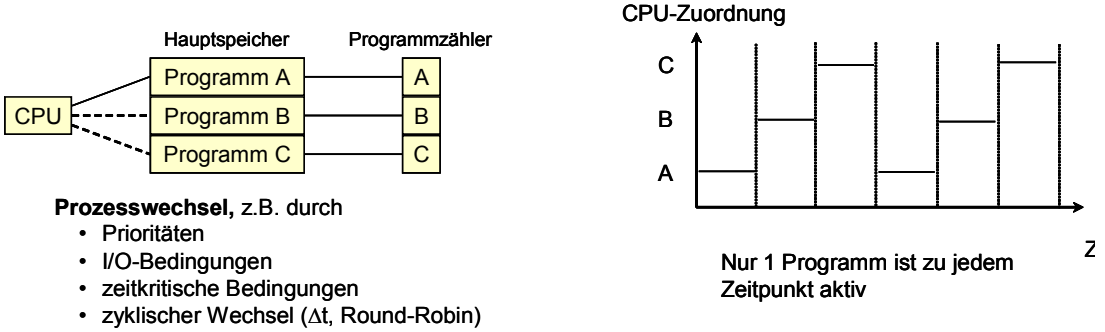


Abbildung 5.2: Multiprogramming (Beispiel)

Somit besitzt jeder Prozess konzeptionell seinen eigenen virtuellen Prozessor. Parallelität dadurch, dass ein *Wechsel des realen Prozessors* zwischen den Prozessen durch Hin- und Herschalten erreicht wird (sog. *Multiprogramming*).

Prozesszustände:
Zusätzlich zum Zustand "aktiv" (running) für einen aktuellen Prozess lassen sich weitere Zustände unterscheiden, worauf die anderen Prozesse warten.
Verschiedene Prozesszustände:
- neu (nicht-existent): Prozess gerade neu erzeugt.
- aktiv (rechnend): Prozessor ist dem Prozess zugeteilt (1).
- ready: Prozess ausführbar (2), aber Prozessor anderem Prozess zugeteilt (rechenbereit)
- blockiert: Prozess kann nicht ausgeführt werden (3), da er auf Eintreten eines externen Ereignisses wartet (sog. *Suspendierung*) oder Betriebssystem hat Prozessor einem anderen Prozess zugeordnet.
- beendet (nicht-ex.): Prozess hat seine Ausführung beendet.

Zustand (1), (2): Ein Prozess kann ausgeführt werden, wenn ein Prozessor verfügbar ist.
Zustand (3): Prozess kann selbst dann nicht ausgeführt werden, wenn Prozessor frei ist.

Zustandsübergänge:
Vier Übergänge sind zwischen den 3 dominanten Prozesszuständen möglich (abgesehen von den Übergängen 5 und 6):

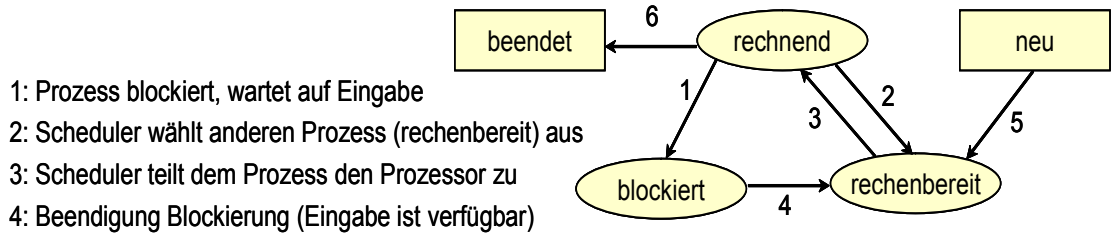


Abbildung 5.3: Zustandsübergänge

Übergang 1: ausgeführt, wenn Prozess nicht weiter ausgeführt werden kann (blockiert). Blockierung durch
- Warten auf externes Ereignis, z.B. Eingabe vom Terminal (sog. "Suspendierung"),
- Betriebssystem hat Prozessor einem anderen Prozess zugeteilt (Systemeigenschaft, zu wenige Prozessoren).

Übergang 2: Prozess-Scheduler entscheidet, dass die CPU einem anderen Prozess zugeordnet wird (z.B. nach Zeit Δt).
Übergang 3: Prozess-Scheduler teilt dem 1. Prozess wieder die CPU zu, nachdem alle anderen Prozesse den Prozessor zugeteilt bekommen haben.
Übergang 4: Das externe Ereignis, auf das der Prozess wartet, tritt ein. Falls Prozessor frei, wird Übergang 3 durchgeführt und Prozess kann Arbeit fortsetzen. Ansonst verbleibt er im Zustand rechenbereit, bis CPU wieder verfügbar ist.
Übergang 5: Ein neu erzeugter Prozess geht in Zustand rechenbereit.
Übergang 6: Fertig ausgeführter Prozess wird terminiert.

Leichtgewichtige Prozesse
Speicherbedarf eines Prozesses i.allg. sehr umfangreich, u.a. Prozess-Nr., CPU-Daten, Statusinformationen, Angaben zu Dateien und E/A-Geräten, Programmcode und seine Daten.

Bei Prozesswechsel sind die Prozessdaten auf Sekundärspeicher auszulagern bzw. wieder einzulagern ~> hohe Systemlast, zeitaufwendig.
Viele Anwendungen erfordern keine völlig neuen Prozesse, sondern nur unabhängige Codestücke (**threads**), die im gleichen Prozesskontext agieren, z.B. Prozeduren eines Programms (sog. Coroutinen).
Verwendung von Threads ermöglicht es, innerhalb eines Prozesses ein weiteres Prozess-System aus sog. Leichtgewichtigen Prozesses (*LWP, lightweight processes*) zu schaffen. LWP sind innerhalb des Prozesses über ihren Namen (*ID*) zu identifizieren.
Für den Wechsel der LWP benutzen die LWP meistens den *Stack des Prozesses*, der für jeden Prozess extra als Platz bei der Erzeugung reserviert wird. Allerdings benötigen LWP im Gegensatz zu schwergewichtigen Prozesse nur wenige *Kontextdaten*, die beim Umschalten geändert werden müssen. Die wichtigsten sind: Prozessstatuswort (PS) und Stackpointer (SP).
Selbst der Programmzähler (PC) kann auf dem Stack abgelegt werden, so dass er nicht explizit übergeben werden muss. Effiziente Implementierungen des Umschaltens in Assemblersprache ermöglichen ein sehr schnelles Prozesssystem.

5.2 Prozessmanagement in verteilten System

Prozessmanagement in verteilten (Client/Server-) Systemen auf 2 Ebenen:
- Prozessverwaltung in einem Server zur Bearbeitung verschiedener Clientanfragen, Einsatz iterativer und nebenläufiger Server,
- Verfeinertes, abgestuftes Prozesskonzept (Threads).

Iterative und nebenläufige ServerEin Server, der auf Anfragen mehrerer Clients reagieren kann, wird iterativ oder nebenläufig realisiert.
Verschiedene Lösungsvarianten und kommerzielle Lösungen, u.a.
- verschiedene autonome iterative Server, Parallelisierung.
- Nebenläufige Server mit Multitasking-Betriebssystemen (Multiplex-Server, Multiprozessorsysteme).

5.3 Threads

Um die Vorteile der Betriebssystemunterstützung für die Prozessverwaltung nutzen zu können, ohne aufgrund der hohen Komplexität schwergewichtiger Prozesse Effizienzeinbußen hinnehmen zu müssen, wurden leichtgewichtige Prozesse, so genannte **Threads** (lightweight processes), eingeführt.
Threads arbeiten innerhalb eines Prozesses und teilen sich einen **gemeinsamen Adressraum**. Jeder besitzt jedoch einen eigenen Kontrollfluss. Die Inter-Thread-Kommunikation kann über gemeinsame Variablen erfolgen, die mittels Semaphoren geschützt werden. Die Erzeugung neuer Threads ist deutlich weniger aufwendig als die neuer Prozesse, da sie in einen bereits existierenden Adressraum eingelagert werden. Ebenfalls weniger Aufwand entsteht bei einem Kontextwechsel, da die Seitentabellen und Informationen über geöffnete Dateien und andere periphere Ressourcen nicht ausgelagert werden müssen.

Einfacher Kontexwechsel bei Threads
Nur wenige Zustandsinformationen sind zu wechseln:
* Programmzähler und Registersatz,
* Thread-lokaler Stack und Heap,
* Liste der Kinder-Threads,
* Ausführungszustand (running, blocked, ready).

Dies entspricht i.w. einem Verwaltungsaufwand wie bei lokalen Prozeduraufrufen.
Im Gegensatz dazu beinhaltet der Kontextwechsel bei Prozessen:
* Kompletter virtueller Adressraum (d.h. Seiten und Seitentabellen),
* Liste geöffneter Dateien und deren Zustände (Deskriptor, Stack, Pointer, ...),

* Liste der Kindprozesse,
* Kontrollstrukturen verwendeter BS-Ressourcen (Timer, Signale oder Semaphore),
* Management-Informationen (Accounting-Informationen, Benutzerdaten).

Die Zusammenarbeit von Threads in verteilten Systemen kann nach verschiedenen Modellen organisiert werden. Das **Dispatcher/Worker-Modell** ist dem Client/Server-Modell ähnlich. Ein Master empfängt die Aufträge und gibt sie an andere Bearbeitungsthreads weiter. **Im Team-Modell** hingegen sind alle Threads gleichberechtigt und entnehmen ihre Aufträge aus einem gemeinsamen Pool, der wie eine gemeinsam benutzte Ressource verwaltet wird. Das **Pipeline-Modell** ordnet die Threads in einer logischen Kette an, wobei jedes Glied nur einen Teilschritt der Berechnung durchführt. Der Auftrag muss eine solche Bearbeitung allerdings erlauben.
Die **Verwaltung von Threads** kann auf verschiedenen Ebenen implementiert werden, im Betriebssystemkern (Kernel-Level-Threads) oder im Benutzer-Arbeitsbereich (User-Level-Threads). User-Level-Threads setzen dabei auf ein Laufzeitsystem auf, das sie verwaltet.
User-Level-Threads lassen das Betriebssystem unverändert und threadbasierte Anwendungen unabhängig vom Betriebssystem. Thread-Umschaltungen sind schneller, da keine Kern-Traps ausgelöst werden. Das Scheduling ist an die Anwendung anpassbar und das System skaliert gut. Nachteilig wirken sich Probleme bei blockierenden Systemaufrufen aus, da das Laufzeitsystem für das BS als ein Prozess gilt. Außerdem sind rechnende Threads nicht unterbrechbar. Zum Preis einer größeren Kontextmenge kann man die Nachteile durch den Einsatz von Kernel-Level-Threads ausgleichen, verliert dann aber auch die Vorteile.

Threads können auf globale Variablen zugreifen, die von anderen Threads verwendet werden. Da sich globale Variablen nicht immer vermeiden lassen (z.B. Unix) können als Abhilfe pro Thread Kopien der globalen Variablen angelegt werden. Dies führt zu zusätzlichen Sichtebenen:

System-globale Variable (sichtbar für alle Threads),
Thread-globale Variable (sichtbar innerhalb eines Threads),
Lokale Variable (sichtbar innerhalb der Prozedur eines Threads).

Hierbei werden neue Systemfunktionen zur Behandlung globaler Variablen nötig.

Zum Programmieren mit Threads stehen zahlreiche **Thread-Bibliotheken** zur Verfügung, z.B. das Thread-Paket von OSF/DCE oder das Leichtgewicht-Prozess-Paket von Sun. Diese Pakete bieten i.allg. folgende Komponenten an:

- Thread-Verwaltung (create, exit, join, detach)
- Thread-Kontrolle (cancel, setcancel)
- Semaphore (init, destroy, lock, trylock, unlock)
- Bedingungs-Variablen (init, destroy, wait, signal)
- Verwaltung Thread-globaler Variablen (create, set, get)

6 Namensverwaltung

6.1 Namen in verteilten Systemen

Namen dienen der **Benennung** und **Identifikation von Objekten** (u.a. Computer, Dienste, Ports, Netzadressen, Informations-Objekten, Dateien, Prozessen, Benutzer), für eine eindeutige Identifikation (lokale / globale Namen) und Namensauflösung (Wiederauffinden, Verzeichnisstruktur). Aufgabe der **Namensverwaltung** ist es, Namen zu vergeben und zu administrieren, die Struktur vorzugeben und Namen auf die von ihnen repräsentierten Objekte abzubilden.
Namen können Auskunft über die Art des Objektes geben (z.B. Drucker), den Zugriff kennzeichnen (z.B. über logische Adressen) oder eine Objekt lokalisieren (z.B. eine logische Adresse). Namen unterteilt in *dienstbezogene Namen* (Dateinamen, Prozessidentifikatoren etc.) und *dienstübergreifende Namen* (z.B. Benutzer, Email-Adress). Bildungsvorschriften von Namen, u.a. Syntax (i.d.R. Betriebssystem), Pfadinformation (Lokalisierung), Metazeichen
Namen besitzen eine Reihe von Merkmalen. Die **Syntax** legt die Namensbildung fest und die Namenstaxonomie erlaubt die Einteilung in einfache Namen (bilden einen flachen Namensraum) und strukturierte Namen (bilden einen hierarchischen Namensraum). Innerhalb einer Namensdomäne ist eine Verwaltungsinstanz für die Namensverwaltung zuständig. Diese besteht aus **Namensdienst** und **Verzeichnisdienst**. Diese löst die Namensbindung wieder auf, die Namen Objekten zuordnet und lokalisiert Objekte anhand ihres Namens.

6.2 Namensdienste

Die **Namensverwaltung** ist zumeist auf einer verteilten Datenbasis installiert, die vom Namensdienst gebildet wird und die die Zuordnung von Namen zu Objekten und zu deren Attributen speichert. Seine Hauptaufgabe ist die Bildung des **Namensraumes** und **Auflösung** der **Namensanfragen** in die entsprechenden Attribute.
Da spezifische Dienste am besten separat angeboten werden (Separation) und Namensräume zusammenwachsen oder getrennt werden können sollen (Integration), ist der Namensdienst meist ein eigener Dienst. Er verwendet das gleiche Namensschema für unterschiedliche Dienste (Unifizierung), zum Beispiel für lokale und verteilte Dateisysteme.
Die Standardoperationen des Namensdienstes sind **Binden** (bind), **Auflösen** (resolve) und **Löschen** (clear). Außerdem bietet er Suchfunktionen auf der Datenbasis an, die entweder Namen als Suchschlüssel verwenden (**White Page Services**) oder Attribute (**Yellow Page Services**). Bei verteilten Systemen ist die Suche nach Namen eine gleichzeitige Traversierung der Datenbasis-Einträge. Dabei ist der Namensraum üblicherweise partitioniert:

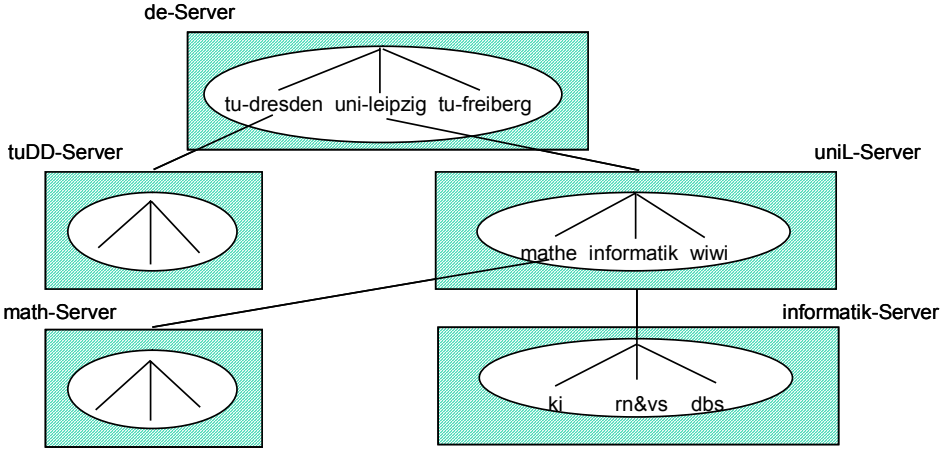


Abbildung 6.1: Aufbau eines partitionierten Namensraumes (Beispiel)

Zur **Namensauflösung** beauftragt der Client einen so genannten **User Agent** (UA), der durch die verschiedenen **Namensserver navigiert** und das Ergebnis zurückliefert. Dazu stehen ihm vier mögliche Vorgehensweise zur Verfügung:

Iterative Navigation:
Falls die Namensanfrage durch den UA bei einem Name Server nicht vollständig aufgelöst, wird die Frage an einen anderen Name Server weitergereicht oder zurückgegeben.

Multicast-Navigation:
UA schickt die Anfrage per Multicast an mehrere adressierten Name-Server.

Iterativ serverkontrollierte Navigation:
UA stellt Anfrage an einen Name Server, der iterativ die weitere Navigation übernimmt.

Rekursiv serverkontrollierte Navigation:
Der vom UA beauftragte Name Server reicht die Anfrage an andere Name Server weiter, wenn ein sein Namensdienst die administrative Grenze überschritten hat (keine Zugriffsberechtigung mehr). Das Ergebnis gibt er dem UA zurück.

Reduzierung der Zugriffshäufigkeit und Leistungserhöhung durch Caching und Replikation.

6.3 Verzeichnisse

Definition

Ein Verzeichnis ist eine Auflistung von Informationen über Objekte in einer gewissen Ordnung. Zu jedem Objekt kann man Detailinformationen abfragen. Informationen in bestimmter Ordnung gehalten für schnelle Wiederauffindung.

Verschiedene Verzeichnisarten

- Gedruckte Verzeichnisse („offline-Verzeichnisse“), u.a.
  - *Telefonbuch*: Auflistung Name, Adresse, Tel.-Nr. in alphabetischer Ordnung
  - *Versand-/ Einkaufs-Katalog, Gelbe Seiten*: Produkte, sortiert nach Bestell-Nr.
  - *TV-Zeitschrift*: zeitliche Auflistung von Programmfolgen
- Elektronische Verzeichnisse („online-Verzeichnisse“), u.a.
  Netzwerkbasiert, Anwendungsspezifisch, Standard-basiert, Datenbank-orientiert

Vorteil elektronischer Verzeichnisse gegenüber „Offline“-Verzeichnissen (Dynamik, Flexibilität, Sicherheit, Personalisierbarkeit).

Typen elektronischer Verzeichnisse

- NOS-basierte Verzeichnisse. Setzen auf ein Network Operating System auf. Beispiele: *Novell‘s NDS oder Microsoft‘s Active Directory.*
- Anwendungsspezifische Verzeichnisse: Eingebettet in Anwendung oder in Kombination mit einer Anwendung ausgeliefert. Abdeckung eines speziellen Anwendungsgebietes. Beispiele: *Lotus Notes Namens- und Adressbuc, Browser-Adressbuch.*
- Zweckspezifische Verzeichnisse: Für bestimmten Anwendungszweck optimiert, jedoch einzeln und anwendungs-unabhängig nutzbar. Beispiele: *Internet Domain Name System (DNS), UDDI (Web-Services).*
- Allzweck- und Standard-basierte Verzeichnisse: Entwickelt für eine breite Palette von Anwendungen. Beispiele: *X.500 basierte Verzeichnisse, LDAP Verzeichnisse*

Verzeichnisdienste

Verzeichnisdienst: Attributierter Namensdienst, Auflösung von Namensreferenzen (Rechnernamen, Adressen und weitere Informationen)

Auswahl bekannter Verzeichnisdienste:

- Internet Domain Name System (DNS)
- Global Name Service (GNS)
- X.500 Directory Service
- Lightweight Directory Access Protocol (LDAP)

6.4 Domain Name System (DNS)

DNS ist eine einfacher Verzeichnisdienst: Hostname, physikalische und logische Internet-Adresse (URL/URI) und Informationen zu Email-Server (optional: Host-Informationen). DNS wurde für die Nutzung im **Internet** entwickelt. Der ursprünglich flache Namensraum konnte nur noch sehr ineffizient gehandhabt werden. Dagegen kann DNS stark partitionierte Namensräume mit einer tiefen Hierarchie verwalten. Internet-Namensraum im DNS in USA organisiert (IESOC bzw. privat), außerhalb USA i.d.R. geographisch partitioniert, Endsuffixe sog. Top-Level-Domains (z.B. de).
DNS-Namen beschreiben von rechts nach links den Weg von der Wurzel zum jeweiligen Blatt des Namensbaumes. Die einzelnen Komponenten des Namens werden durch Punkte getrennt und veranschaulichen näherungsweise die hierarchische Partitionierung des Namensraumes. Zur Auflösung eines Namens ist stets der komplette Domain-Name notwendig. Client-SW (UA) hängtselbständig die sog. Default Domain an, z.B.

```
isun  =>  isun.informatik.uni-leipzig.de.
```

Zwei Standardanfragen für den Client unterstützt:

- Die Namensauflösung (Host Name Resolution) liefert zu einem **Rechnernamen** (z.B. isun.informatik.uni-leipzig.de) die entsprechende IP-Adresse (z.B. 139.18.12.1) zurück.
- Die Mail Host Location Anfrage liefert zu einer Email-Adresse eine geordnete **Liste mit Rechnernamen**, die für diese Adresse eine Mailbox enthalten.

Einige Implementierungen verarbeiten weitere Anfragen, wie Reverse Resolution (liefert Namen zu IP-Adressen), Host Information (Informationen über einen Rechner , aber Sicherheitsproblem) oder Well Known Services (gibt Auskunft über dort verfügbare wohlbekannte Dienste, wie telnet, ftp oder www und deren Protokolle).
Die Namensdaten sind in Zonen unterteilt. Eine Zone enthält folgende Daten:

- Attribute aller Namen der Domäne (außer der Subdomäne),
- Namen und Adressen der Namensserver, die für die Zone maßgebend sind,
- Namen der Server mit maßgebenden Daten von Subzonen,
- Managementparameter (u.a. Caching, Replikation).

Jede Zone muss mindestens einmal repliziert werden (**Primär- und Sekundärserver**), wobei die Sekundärserver die Daten periodisch vom Primärserver laden. Cache-Einträge auf den Servern besitzen ein Verfallsdatum (time-to-live), nach dem sie aus dem Cache gelöscht werden. Spezifische Implementierung: BIND (Berkeley Internet Name Domain).
DNS ist ein primitiver Namensdienst, der nur wenige Funktionen anbietet. Daher wird er oft mit anderen lokalen Diensten, wie dem Network Information Service (NIS) kombiniert, der dann unter anderem Benutzernamen und Passwörter liefert.

6.5 Global Name Service (GNS)

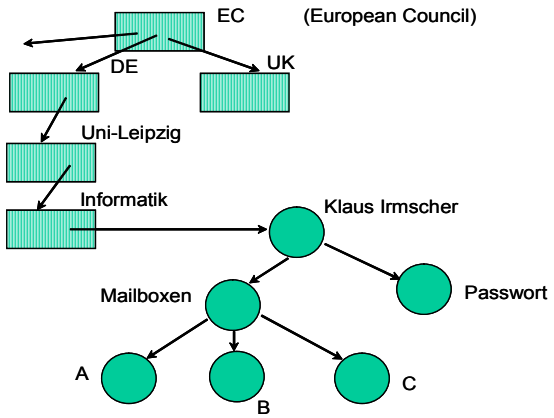


Abbildung 6.2: Aufbau des GNS-Verzeichnisbaumes (Beispiel)

GNS wurde als Teil von DCE von der OSF konzipiert und von der Digital Equipment Corporation (DEC) entwickelt. Es ist ein weltweit gültiger Namensdienst zur Ressourcenlokation, Mailadressierung und Authentisierung, der Caching und Replikation unterstützt.
Die Namens-Datenbasis ist ein Baum von Verzeichnissen (Directories), die jeweils einen eigenen Directory Identifier (DI). Namen bestehen aus Paaren <Directoryname, Wertename>, wobei der Wertename auf einen strukturierten Wertebaum zeigt. GNS unterstützt die Vereinigung zweier Namensbäume (Anhängen an gemeinsame neue Wurzel) bzw. Restrukturieren.

6.6 X.500 Directory Service

Steigender Bedarf an vielseitiger einsetzbaren Verzeichnissen führte zur Entwicklung unabhängiger Standards, u.a. X.500 und LDAP. Mitte der 80er Jahre: *White Pages* Verzeichnis der CCITT (später ITU) und *Name Service* der ISO für OSI Netzwerke. 1990 folgte die Entwicklung des X.500 Standards durch ISO und CCITT. X.500 erstes reines Multi-Purpose Directory, setzt auf den OSI Protocol Stack auf, offener Standard.
X.500 ist ein **hochwertiger**, **attributierter Verzeichnisdienst**. Er wurde durch CCITT (ITU-T) standardisiert und beschreibt eine Anwendung der 7. Schicht des OSI-Referenzmodells. Er wird zusammen mit GNS im DCE eingesetzt, genauso in einigen Implementierungen von OMG CORBA. Bis ca. 1995 auch als Basis für LDAP eingesetzt, später eigenständige LDAP-Server.

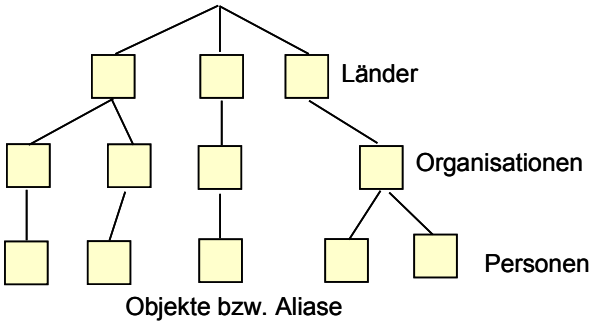


Abbildung 6.3: X.500 Directory Information Tree (DIT)

Der **Namensbaum** von X.500 (**Directory Information Tree,** DIT) ist als Datenstruktur mit allen Attributwerten in einer Datenbank, der **Directory Information Base** (DIB) gespeichert. Informationen des DIT sind Eintragungen über Objekte.
Jeder **DIB-Eintrag** besteht aus einem Namen, der zur Auflösung immer komplett (absolut) angegeben werden muss, und einer Menge von Attributen. Diese besitzen einen ausgezeichneten Typ und mehrere Werte und können zwingend oder optional sein. Typen sind durch Typnamen und eine Syntaxdefinition in ASN.1 spezifiziert.
Alle Einträge werden jeweils durch ein Object-Class-Attribut einer Objektklasse zugeordnet, wobei neue Klassen definierbar sind.

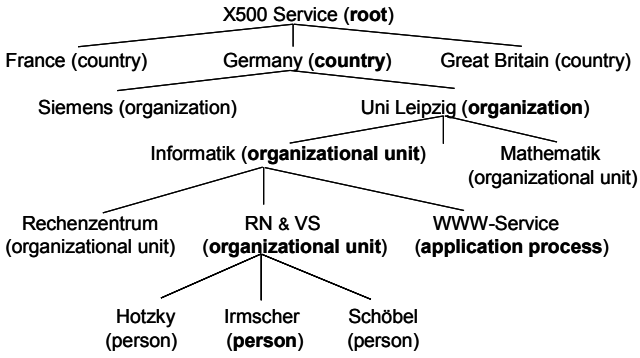


Abbildung 6.4: Ausschnitt aus einem X.500 DIT

Name eines DIB-Eintrages wird aus den vorhandenen Einträgen gewählt und als **Distinguished Attribute** zur Positionierung im Baum verwendet. Er dient als Navigationsanker für den gesamten Eintrag. Laut Standard soll es auf der Welt nur eine einzige DIB geben, deren Daten auf individuelle X.500-Server verteilt sind.
In der Nomenklatur von X.500 heißen die Server **Directory Service Agent** (DSA) und die Clients **Directory User Agent** (DUA). Ein DUA bietet 3 Schnittstellentypen: Lese-, Such-, Änderungs-Schnittstelle. Ist ein Verzeichnis auf mehrere DSAs verteilt, so kommunizieren diese untereinander, um eine Suchanfrage zu bearbeiten. Zur Nutzung des Clients sind 2 Operationen definiert: Lesen und Suchen.
Der **X.500-Standard** schreibt weder eine Implementierung noch die Navigationsform vor. Ein Nachteil des Standards ist, dass die Definition der Objektklassen auf nationaler und internationaler Ebene erfolgen muss, um weltweites Suchen zu ermöglichen. Bekannteste vollständige Implementation eines X.500-Verzeichnisdienstes als Multi-Purpose Directory von Quipu (University College London).

6.7 Lightweight Directory Access Protocol (LDAP)

Entwicklung LDAP
X.500 zu komplex und schwierig auf PC implementierbar ~> Beginn 90er Entwicklung zweier unabhängiger Protokolle für vereinfachten X.500-Zugriff (über IETF):

- *Directory Assistance Service (DAS, RFC 1202)* und
- *Directory Interface to X.500 Implemented Efficiently (DIXIE, RFC 1249).*

Beide Protokolle setzen als X.500 direkt auf TCP/IP auf und beruhen auf Prinzip eines *Übersetzungsservers*, der Befehle des einfachen Protokolls in das komplexe, X.500-eigene Directory Access Protocoll (DAP) übersetzt.
IETF: Verstärkung der Entwicklung, Fokusieren auf leichtere Protokolle -> Ergebnis: *LDAP.*
Erste Veröffentlichung 1993 (RFC 1487).
LDAPv2 war erste nutzbare Version (RFC 1777), LDAPv3 erschien 1997.

Verzeichnisdienst LDAP
LDAP (Lightweight Directory Access Protocol): standardisiertes Protokoll ~> regelt den Austausch zwischen Verzeichnisdiensten. Vereinfachung des X.500-Protokolls. Einfache Struktur ~> rasante Verbreitung und hohe Akzeptanz. Es definiert Datenaustausch und Datenformate sowie Aufbau von LDAP-Verzeichnissen und Zugriff auf die Dienste.
LDAP sichert Interoperabilität zwischen verschiedenen (LDAP kompatiblen) Verzeichnisdiensten: Datentransferformat LDIF, Programmierinterface LDAP API. Modelle im LDAP:

- Informationsmodell (Speicherung der Daten),
- Namensmodell (Strukturierung der Daten),
- Funktionsmodell (Ablage und Auslesen von Informationen),
- Sicherheitsmodell (Zugriffsschutz der Daten).

Standalone-Dienst (slapd)
Implementation von X.500 Server Software immer noch kompliziert. Zugriffsstatistik: nahezu 99% aller Anfragen auf X.500 Server erfolgen über LDAP.
Somit Entwicklung eines eigenen Verzeichnisservers auf Basis LDAP. Entfernen des zwischen Client und X.500-Server geschalteten LDAPD (LDAP Daemon), der nur LDAP-Anfragen in X.500-DAP-Anfragen und andersherum übersetzte. Dadurch Komplexität des Gesamtsystems drastisch reduziert. Außerdem wurde Ziel eines weltweiten X.500-Verzeichnisses immer unwahrscheinlicher.
1995 Entwicklung des *Standalone LDAP Daemons (SLAPD).* Unterstützung durch NSF.
Erste *slapd*-Version im LDAP Package 3.2 der Universität Michigan (1995). Damit LDAP nun unabhängig von X.500. Mit LDAPv2 Aufbau von SLAPD Verzeichnisserver-Netzen

möglich. 1996 Ernennung LDAPs zum *Internet Verzeichnisdienst-Protokoll erster Wahl* durch SW-Konsortium.

LDAP Version 3 (LDAPv3)
LDAP Version 3 wesentlich ausgereifter als Version 2, heute anerkannter Internet Standard. Verbesserung in vielen Bereichen:
- *Internationalität:* Verwendung des UTF-8-Zeichensatzes (erfaßt alle Sprachen).
- *Verweise:* Möglichkeit des Verweisens auf andere LDAP Server (s. Web-Server)
- *Sicherheit:* Unterstützung von SSL (Secure Socket Layer), TLS (Transport Layer Security Layer) und SASL (Simple Authentication and Security Layer).
- *Erweiterbarkeit*: Bietet die Möglichkeit, neue Befehle zu definieren oder vorhandene zu erweitern ~> dadurch zukunftssicherer.
- *Übertragung der Servereigenschaften*: an Clients werden vom Server nun die Eigenschaften und Schema des Servers übertragen (Nutzung des spezifischen Verzeichniseintrages root DSE - Directory Server Specific Entry).

LDAP Produkte
Viele heutige LDAP Directory Server basieren auf dem Konzept slapd der Universität Michigan, d.h. sie sind um einen Standalone-Server herum konstruiert, der eine eingebettete High-Performance-Datenbank enthält. Somit enthalten viele den U-M-Code (Univ. Mich.). Viele ehemalige X.500 Hersteller empfingen LDAP mit offenen Armen. Einige heute bekannter Mehrzweck-Verzeichnis Server, die auf LDAP basieren:
- Netscape‘s Directory Server
- Innosoft Distributed Directory Server
- Lucent Technologie‘s Internet Directory Server
- Sun Microsystem‘s Directory Services
- IBM‘s DSSeries LDAP Directory
- University of Michigan‘s SLAPD Server

Diese Implementationen unterstützen ein breites Spektrum an Internet-, Intranet- und Extranet-Anwendungen. Sie können infolge LDAP problemlos untereinander kommunizieren, vergleichbar mit SMTP oder HTTP Web-Servern.

Funktionsweise LDAP
LDAP ist ein Client/Server-Protokoll: Client sendet Anfrage an Server und erhält vom Server eine positive oder negative Antwort nach vorangegangener Bearbeitung. LDAP ist ein nachrichtenorientiertes Protokoll, d.h. Client erzeugt eine Nachricht, die den Request enthält, und schickt diese an der Server. Server verarbeitet den Request und antwortet dem Client in einer Serie von Nachrichten, u.a. einen Result Code. Es gibt insgesamt neun grundlegende Protokoll-Operationen, die in drei Kategorien zusammengefasst werden können.
- Anfrage-Operationen: *search, compare*
- Update-Operationen: *add, delete, modify DN (rename)*
- Authentifications- und Control-Operationen: *bind, unbind, abandon*

Mit *bind* und *unbind* meldet sich Nutzer nach einer anonymen Anmeldung am LDAP- Server an und ab (er „bindet“ quasi seine Identität an die vorhandene Verbindung). Durch Kombination dieser 9 Operationen sind komplexe Anfragen und Aufgaben erfüllbar.

LDAP Informationsmodell
Es definiert, wie Daten gespeichert werden: Datentyp und kleinste Informationseinheit. Kleinste Informationseinheit: Eintrag (entry). Eintrag beschreibt ein reales Objekt. Informationen werden als Attribute gespeichert. Attribute bestehen aus Attribut-Typ und Attribut-Wert. Beispiel: cn=Hans Müller. Attribute können single-valued oder multi-valued sein.

Zwei Typen von Attributen: Nutzerattribute (vom Nutzer festgelegt) und Operationale Attribute (vom Verzeichnisdienste festgelegt). Alle Informationen über erlaubte Attribute, Wertigkeit usw. werden im LDAP Verzeichnis-Schema (directory schema) definiert.

LDAP Namensmodell

Damit gegelt, *wo* Daten abgelegt und gefunden werden. LDAP bietet hierarchische Speicherung, dadurch sehr flexibel. Jeder Eintrag hat einen eindeutigen Namen – ***distinguished name*** (DN). Aufbau des DNs wie Verzeichnisangabe bei UNIX, nur rechts die Wurzel und links das Objekt. DN ist eine Aufreihung von Attributen, getrennt durch Komma (,).
DN setzt sich aus seinem Namensgebenden Attribut (*Relativer DN, RDN)* und dem DN des Elternobjektes zusammen. RDN muss in seinem Unterbaum einzigartig sein.

Beispiel:

RDN	cn=Hans Mustermann
DN des Elternobjekts	o=uni-leipzig, c=de
DN des Objektes	cn=Hans Mustermann, o=uni-leipzig, c=de

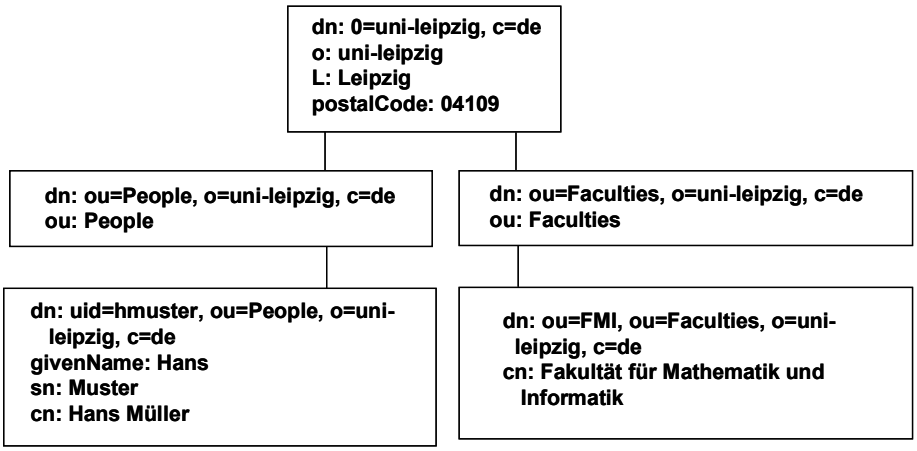


Abbildung 6.5: LDAP Namensmodell (Beispiel)

Namensmodell von LDAP muss dafür sorgen, dass jeder Eintrag einen eindeutigen Namen erhält, der ihn eindeutig identifiziert. Diese einzigartigen Namen sind im LDAP die *distinguished names* (DN). Sie setzen sich aus mehreren Komponenten zusammen, die jeweils ein Objekt im Baum bestimmen. *Beispiel: cn=Hans Mustermann, o=uni-leipzig, c=de*

7 Verteilte Dateisysteme

7.1 Anforderungen

Dateien dienen der **permanenten Speicherung** von **Daten**, meist als Reihen von Bytes, auf die über Dateizeiger zugegriffen wird. Das Dateisystem ist dann die Schnittstelle, die das Betriebssystem Anwendungen zu Festspeichermedien (Festplatten, CDs etc.) anbietet.
In **verteilten Dateisystemen** werden Dateien auf mehreren als Server fungierenden Rechnern gespeichert und den Clients ein Dateidienst angeboten, um auf sie zuzugreifen. Dabei sollen alle Transparenzeigenschaften, vorallem Replikationstransparenz und Nebenläufigkeitstransparenz, gewährleistet werden.

7.2 Architektur verteilter Dateisysteme

Ein verteiltes Dateisystem ist aus drei konzeptionellen Komponenten aufgebaut.

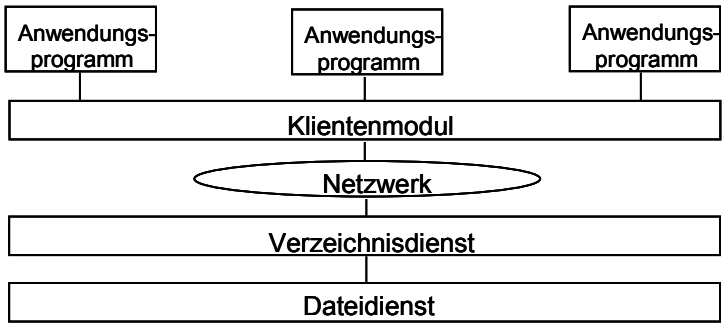


Abbildung 7.1: Komponenten eines verteilten Dateisystems

Dateidienst:
Der Dateidienst ist die unterste Komponente und stellt elementare Operationen wie Lesen/Schreiben, Plattenein- und -ausgabe und Pufferung bereit.

Verzeichnisdienst:
Ein Verzeichnisdienst dient zur Abbildung von Dateinamen auf Dateizeiger, dem Test und der Änderung von Zugriffsrechten und höherwertiger Dateiattribute, dem Verwalten der Dateien im Namensraum sowie der Unterstützung von symbolischen Verweisen oder Aliasen.

Klientenmodul:
Das Klientenmodul ist die Schnittstelle zum Anwendungsprogramm, über die Aufrufe an Verzeichnis- und Dateidienst erfolgen. Lokale Aufrufe im Dateisystem werden über das Modul auf das verteilte System abgebildet. Die Implementierung des Moduls kann als einfacher RPC-Stub erfolgen oder gar ein eigenes Dateisystem inklusive Caching erfolgen.

Namensbildung
Verschiedene Sichtweisen auf ein verteiltes Dateisystem liefern abhängig von der Namenbildungsvorschrift unterschiedliche Namen für die gleiche Datei.
Hat z.B. jeder Rechner eine von den andren unabhängige Verzeichnisstruktur und bildet Dateinamen als **Konkatenation von Rechner- oder Servername** und dem entsprechenden Pfad des jeweiligen Rechners, dann ist die Namensgebung orts- bzw. serverabhängig.
Eine andere Möglichkeit ist **Remote Mounting**, wobei die entfernte Verzeichnisstruktur lokal logisch eingebunden wird. Auch hier ist **keine Namenstransparenz** gegeben, da die Einbindung auf jedem Rechner individuell erfolgen kann. In den nachfolgenden Grafiken sind A, D und G jeweils die Wurzeln von drei lokalen Dateisysteme der Rechner R1, R2 und R3. Die Baumdarstellung zeigt Möglichkeiten der Namensbildung durch Mounting (aus Sicht der

Rechner R1 bzw. R3). Offensichtlich heißt die Datei (x) einmal A/C/D/E/x auf R1 und einmal G/A/C/D/E/x auf R2.

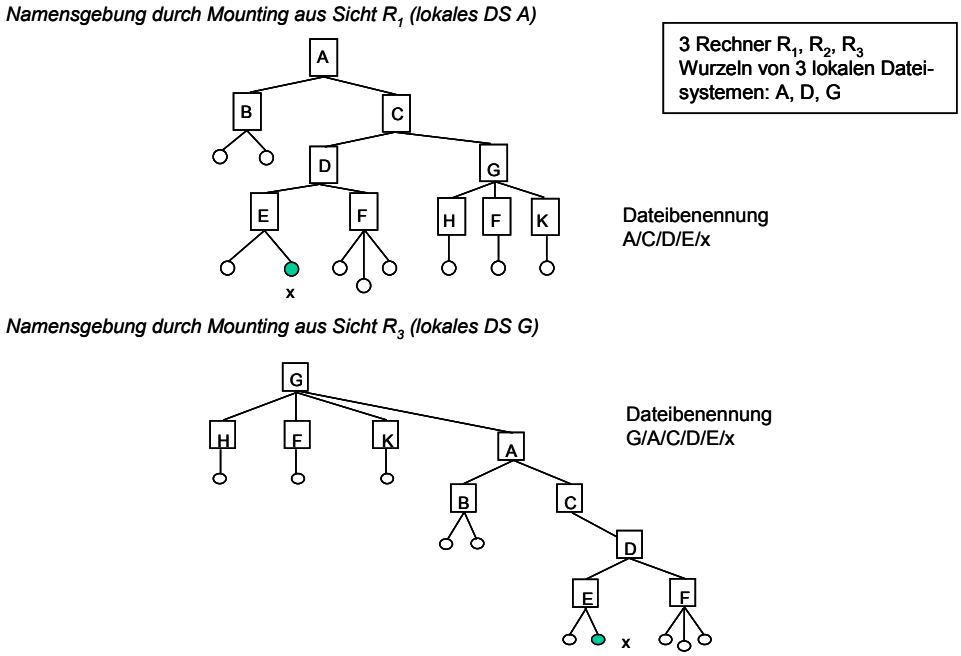


Abbildung 7.2: Remote Mounting (aus Sicht einzelner Rechner)

Abhilfe schafft hier die dritte Form der Namensbildung, ein **uniformer Namensraum** durch Einsetzen einer **übergeordneten Wurzel** über die lokalen Dateisysteme, einer **Superroot**. Dadurch wird Orts- und Namenstransparenz erreicht. A, D und G werden an die Wurzel / angehängt und es ergibt sich ein uniformer Pfad /D/E/x für alle Teilnehmer.

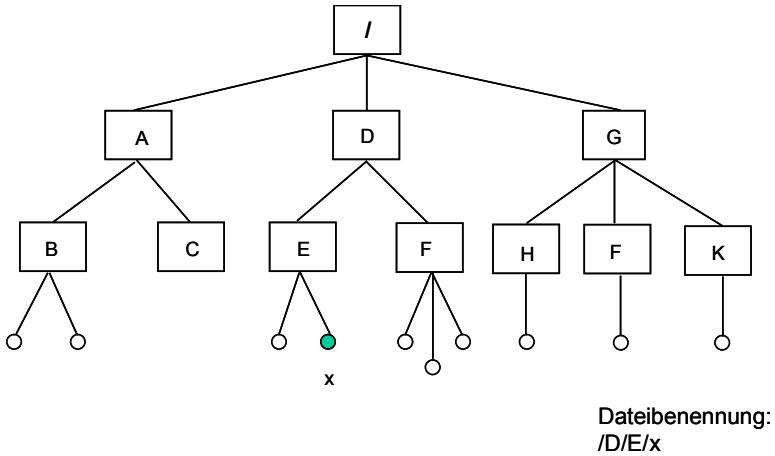


Abbildung 7.3: Remote Mounting (Superroot)

7.3 Semantik des Datei-Sharings

Greifen mehrere Prozesse auf eine Datei gemeinsam zu, können Probleme entstehen (kritische Bereiche, wechselseitiger Ausschluss, Synchronisation, Konsistenz). Auf Einzelrechnern wird ein gemeinsamer Zugriff auf eine Datei durch die zentrale Auslegung serialisiert. In einem verteilten Dateisystem mit zentraler Koordinierung ist dieses Prinzip leicht übertragbar, indem Dateien vom Client gesperrt werden, wenn er auf sie zugreift. Prinzip erschwert bei Nutzung von Caching und Replikation.

Generelle Lösung des Sharing-Problems: *Sperren von Dateien*: greift ein Client auf eine Datei zu, so sperrt er sie. Dadurch aber Einschränkung der Parallelität. Je nach Verfahren unterscheidet man verschiedene Semantiken zur Realisierung des konkurrenten Dateizugriffs:

Einzelkopie-Semantik (Unix):

Ein Lesezugriff liefert immer das **Ergebnis des** (zeitlich) **letzten Schreibzugriffes**. Mit einem zentralen Dateiserver ist die Implementierung einfach, Caching erfordert zwingend eine write-through-Semantik. In den meisten Fällen ist der Verwaltungsaufwand sehr hoch und die Effizienz gering.

Sitzungs-Semantik:

Beim Öffnen einer Datei erhält der **Client** eine **eigene Kopie**. Mit dieser arbeitet er bis zum Schließen der Datei, wonach sie auf den Server zurück geschrieben wird. Erst dann werden die Änderungen für andere Clients sichtbar.

Da jedoch mehrere Clients zur gleichen Zeit an derselben Datei Veränderungen vornehmen können, gibt es **vier Wege**, wie mit den Änderungen verfahren wird.

1. Mit dem Schließen werden alle vorherigen Versionen überschrieben.
2. Ein Abstimmungsprozess wird von den Klienten vor dem Zurückschreiben durchgeführt.
3. Der Server mischt die Versionen auf geeignete Weise (bei bekanntem Anwendungskontext, z.B. Gruppeneditoren, sinnvoll).
4. Unterschiedliche Versionen werden getrennt voneinander weitergeführt (Versioning, muss bei Namensgebung beachtet werden)

Unveränderlichkeits-Semantik:

Hierbei sind **Dateien nicht veränderbar**. Sie können gelesen oder neu erstellt werden. Das Ergebnis ist analog dem 4.Verfahren der Sitzungs-Semantik.

Transaktions-Semantik:

Alle Dateizugriffe sind Transaktionen, was eine serielle Ausführung garantiert. Die Dateien bleiben auf jeden Fall konsistent, zum Preis eines hohen Verwaltungsaufwandes.

7.4 Implementierungsaspekte

Implementierungen unterscheiden sich in den Aspekten Architekturtyp, Verzeichnis-Management und Art des Replikations- und Caching-Verfahrens.

7.4.1 Architekturtypen

Charakteristika: Art des Zugriffsmodells und die Realisierung des Servers.

Zugriffsmodelle

Upload/Download-Modell:

Dieses Modell entspricht i.w. dem Sitzungs-Modell, da eine Datei beim Öffnen durch den Client vom Server vollständig herabgeladen und dann lokal bearbeitet wird. Erst beim Schließen der Datei erfolgt eine Rückübertragung zum Server. Das Netzwerk wird nur während der Dateiübertragung belastet, dafür aber mit hohem Volumen

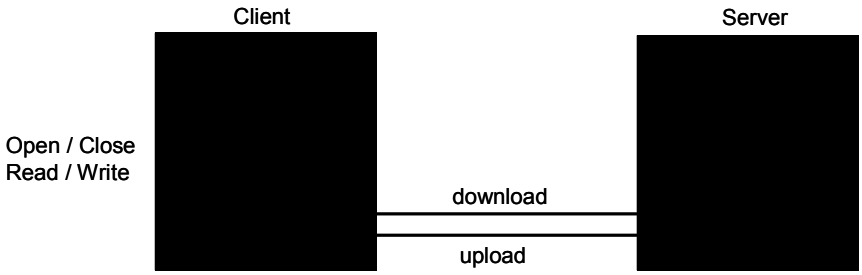


Abbildung 7.4: Zugriffsmodell (Upload/Download)

Remote-Access-Modell:

Großteil der Funktionalität des Dateisystems liegt auf der Serverseite. Das Klientenmodul hat i.w. nur eine Kommunikationsfunktion. Zugriffe des Clients werden einzeln auf dem Server ausgeführt. Das Netzwerk ständig belastet, aber nur mit einem kleinen Volumen.

Realisierung der Dateiserver
Für die Verwaltung von Dateizugriffen werden dynamisch Zustandsinformationen gespeichert, z.B. welche Dateien durch einen Client geöffnet sind, die Positionen der jeweiligen Dateizeiger und auf Dateien gesetzte Sperren. Verwaltung dieser Informationen kann

- *Server-seitig* (sog. **zustandsbehaftete Dateiserver**) oder
- *Client-seitig* (sog. **zustandslose Dateiserver**)

erfolgen. Beide Strategien mit Vor- und Nachteilen.

Zustandslose Dateiserver:

Da alle Zustandsinformationen vom Client verwaltet werden, entstehen beim Absturz desselben **keine Waisen** auf dem Server. Zudem entfallen die Operationen Öffnen und Schließen und Backup- bzw. Replikationsserver lassen sich einfach implementieren.

Zustandsbehaftete Dateiserver:

Da die Zustandsinformationen auf dem Server verwaltet werden, müssen sie nicht über das Netzwerk übertragen werden, was kurze Nachrichten ermöglicht. Dateizugriffe sind schneller, da Positionszeiger bereits auf die richtige Stelle zeigen. Auch intelligentes Precaching und die Unterstützung von Dateisperren sind möglich.

7.4.2 Verzeichnisdienst

Verzeichnismanagement

Hauptaufgabe: **Auflösen von Dateinamen.** Dazu wird Dateiname auf einen Zeiger abgebildet, der auf den physikalischen Ort der Datei auf Festplatte zeigt. Zur Auflösung des Dateinamens muss man den zugehörigen Pfadnamen durchlaufen, der auch über mehrere Rechner verteilt sein kann.

Zwei Prinzipien des Durchlaufens von Pfadnamen:

1. *Iteratives Durchlaufen:* Client fragt beim Server nach der Datei, für den der 1. Teil des Pfades passt. Kann dieser den Zeiger auf die Datei nicht liefern (weil er nicht den gesamten benötigten Verzeichnisbaum hält), so gibt er dem Client den Pfad und Server zurück, bei dem der Client weiter nachfragen kann. Vorgehen wiederholt der Client, bis er den Zeiger auf die Datei erhält. Ab hier kann dann Client direkt über den verwaltenden Server auf die Datei zugreifen. Navigationsalgorithmus ist vollständig auf Clientseite implementiert. Kommunikationsmodell synchron, u.a. RPC (entfernte Prozeduraufrufe).
2. *Rekursives Durchlaufen:* Client stellt an Server die Anfrage bezüglich einer Datei. Kann Server diese Anfrage nicht beantworten, gibt er sie mit der Information, welcher Client die Anfrage gestellt hat, an nächsten Server weiter. Ist gesamter Pfadname aufgelöst, so schickt letzter Server den gewünschten Dateizeiger zum Client zurück. Dateizugriffe können dann zwischen Client und Server erfolgen. Kommunikationsmodell: asynchron (wegen asynchron ist RPC nicht mehr möglich).

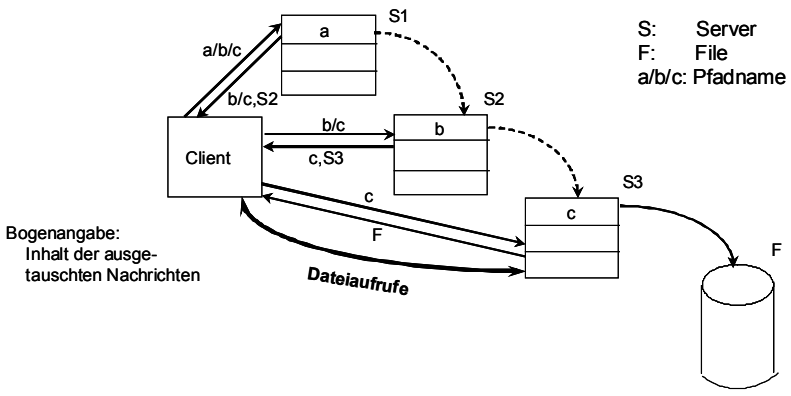


Abbildung 7.5: Durchlaufen von Pfadnamen (iterativ)

7.4.3 Caching und Konsistenz

Zur **Verbesserung** der allgemeinen **Leistung** des Dateisystems können Dateien, die sich physikalisch auf dedizierten Servern befinden, im Arbeitsspeicher des Servers, auf der Festplatte des Clients oder im Arbeitsspeicher des Clients **als Kopie** gehalten werden.
Um die **Konsistenz** bei Schreibzugriffen auf solchen Dateien zu sichern, sind **Cachekohärenz-Algorithmen** erforderlich:

Write-Through:
Schreibzugriffe erfolgen immer sowohl im Cache als auch in die Serverdatei. Das Verfahren ist ungeeignet für verteilte Systeme, da es eine völlig unzureichende Effizienz besitzt.

Delayed-Write:
Hierbei werden Änderungen über einen Zeitraum gesammelt und dann auf einmal (Burst) auf dem Server geschrieben. Das Verfahren hat eine indeterministische, zeitabhängige Semantik und ist daher nur selten einsetzbar.

Write-on-Close:
Dies entspricht der Sitzungs-Semantik.

Zentrale Koordinierung:
Eine zentrale Stelle kennt alle Cacheeinträge der Clients. Der Koordinator wird über jeden Schreibzugriff informiert und verteilt diese Information an die Clients, die die Datei im Cache halten. Jene können dann ihre lokale Kopie löschen. Das Verfahren ist durch seine zentrale Auslegung nicht robust und schlecht skalierbar.

Zur weiteren Verbesserung der Leistung eines VDS kann **Replikation von Daten** auf verschiedenen Servern eingesetzt werden. Dadurch bessere Verfügbarkeit und Zuverlässigkeit.

7.5 Sun Network File System (NFS)

NFS hat seine größte Verbreitung in **Unix-Umgebungen** und wurde speziell für heterogene HW/SW-Umgebungen entwickelt. Es ist der **de-facto-Standard** für verteilte Systeme.

7.5.1 NFS Architektur

NFS-**Client** und –**Server** sind jeweils im BS-Kern eingebunden und symmetrisch realisiert, d.h. jeder Rechner kann beide Rollen wahrnehmen. Die Kommunikation erfolgt über das **NFS-Protokoll**.
Anwendungsprozesse greifen über ein **virtuelles Dateisystem** (VFS), einer zusätzlichen Abstraktionsschicht, auf Dateien zu, wobei die Anfragen an das lokale DS oder einen entfernten NFS-Server weitergeleitet werden. NFS erlaubt es auch, plattenlose Clients zu betreiben.

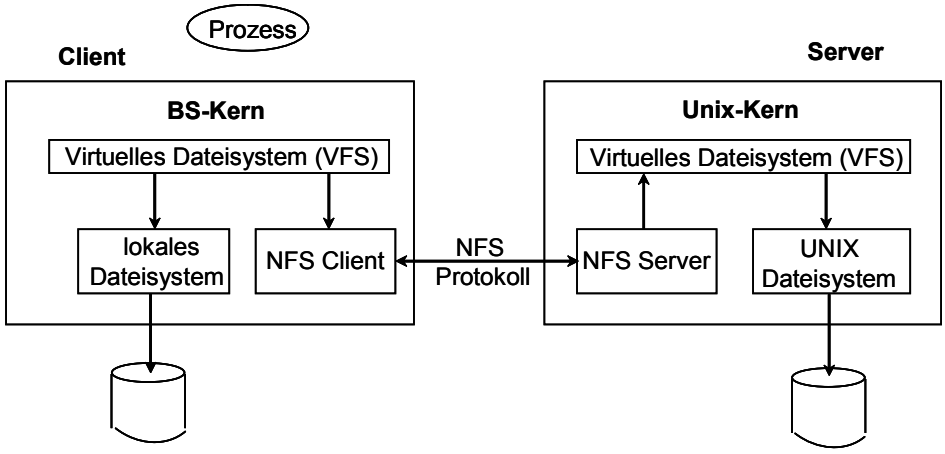


Abbildung 7.6: Architektur NFS

Das VFS enthält so genannte v-nodes als Datei-Handles, die entweder auf lokale i-nodes oder auf entfernte r-nodes zeigen. Der Anwendungsprozess erhält beim Öffnen einer Datei einen Dateideskriptor, der einer v-node im VFS zugeordnet ist.
NFS arbeitet nach dem Prinzip des **Remote Mounting**. Der Client hängt die gewünschten Verzeichnisse, die ein anderer (in der Rolle als Server) über eine Exporttabelle freigibt, an seine eigene Verzeichnisstruktur an. Mit der Ausführung des mount-Befehls wird der jeweilige Server kontaktiert, der dann die Dateihandles mit i-node-Nummer und Dateityp, sowie Schutzbits zurückliefert.

7.5.2 NFS Protokoll

NFS implementiert **zustandslose Server**, so dass jedem Dateizugriff alle notwendigen Zustandsinformationen beigefügt werden müssen. Neben Verwaltungsfunktionen sind drei Grundoperationen möglich:

lookup (*dirfilehandle*, *filename*):
Nach der angegebenen Datei wird im angegebenen Verzeichnis (rekursiv) gesucht. Dateihandles und Dateiattribute werden zurückgegeben.

read (*filehandle*, *offset*, *count*):
Aus der angegebenen Datei werden ab Position *offset count* Bytes geliefert.

write (*filehandle*, *offset*, *count, data*):
In die angegebene Datei werden ab Position *offset count* Bytes mit dem Inhalt von *data* be- bzw. überschrieben.

Die Protokolle für Mounting, Verzeichnis- und Dateizugriff laufen über Sun-RPC. Die Authentisierung der Clients erfolgt mittels öffentlicher Schlüssel aus dem NIS. Damit sind Daten stärker schützbar als mit den üblichen Zugriffskontrollen (Dateiattributen).

7.6 Andrew File System (AFS)

7.6.1 AFS Architektur

AFS verwendet aus Kompatibilitätsgründen die normalen Unix-Dateioperationen und das **NFS-Protokoll** für Dateitransfers zwischen Servern. Es gibt zwei wesentliche Unterschiede zu NFS. Zum einen arbeitet AFS nach dem Upload-/Download-Verfahren und zum anderen werden Dateien in großem Umfang auf der Festplatte des Clients gecacht.. Hierfür werden ein Least-Recently-Used-Verfahren und eine Cache-Größe von über 100 Megabytes eingesetzt.

AFS besteht aus der Client-Komponente **Venus** und der Server-Komponente **Vice**. Beide sind ursprünglich User-Level-Prozesse, Venus in neuen Versionen aber auch im BS-Kern.

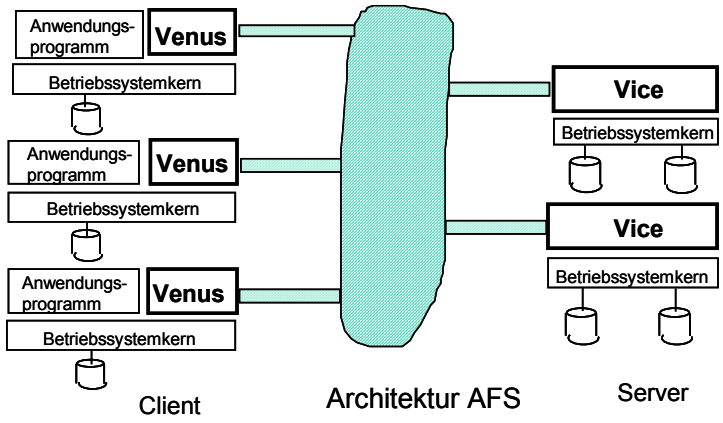


Abbildung 7.7: Architektur AFS

Aufrufe zum Öffnen und Schließen von Dateien werden im Kern abgefangen und an Venus weitergeleitet. Alle anderen Aufrufe gehen an das lokale Dateisystem. Venus und Vice sind multithreaded implementiert und kommunizieren mittels RPC über IP.

Der **Namensraum entspricht** im AFS dem des **Unix-Dateisystems**, wird aber in einen lokalen und einen gemeinsamen Dateibaum (Pfad /afs) aufgeteilt.
Die gemeinsamen Teile verwaltet AFS. Dort liegen alle Dateien, die lokal Konfigurationen speichern. Über symbolische Verweise können entfernte Verzeichnisse in den lokalen Dateibaum eingebunden werden.
Cache-Einträge legt Venus im Verzeichnis /cache ab. Dann erfolgen Zugriffe auf dort gespeicherte Dateien transparent, ohne dass der Name geändert werden muss und ohne Beteiligung von Venus.

7.6.2 Semantik und Implementierung

Der organisatorische Aufbau von AFS unterscheidet drei Elemente.

Cell:
Eine Zelle ist eine administrative Einheit, z.B. eine Abteilung oder Firma. Zellen können miteinander durch Mounting verbunden werden und beinhalten jeweils eine Ansammlung von Volumes.

Volume:
Ein (logischer) Datenträger (Volume) ist eine Sammlung zugehöriger Verzeichnisse, z.B. das User-Volume. Auf dieser Ebene können Zugriffsrechte verwaltet werden.

Directory, **File**:
Die eigentlichen Verzeichnisse werden durch einen 96-Bit-Dateibezeichner eindeutig identifiziert. Dieser besteht aus einer 32-Bit-Volume-Nr., einer 32-Bit-v-node und einer weiteren 32-Bit-Zahl, um Eindeutigkeit in Raum und Zeit herzustellen.

Beim Öffnen einer Datei wird diese in das /cache-Verzeichnis kopiert. Danach können lokale Prozesse darauf zugreifen, wobei eine Einzelkopie-Semantik verwendet wird. Beim Schließen der Datei wird die geänderte Version auf den Server zurückgeschrieben.

AFS-2 implementiert zustandsbehaftete Server. Vice protokolliert alle Dateien mit, die Venus zugreift und die daher in einem Cache gespeichert sind. Bei Schreibzugriffen werden so genannte Callbacks verschickt, damit alle anderen Clients ihre Kopie invalidieren können.
Stürzt ein Client ab, so ist nach dem Hochfahren sein AFS-Cache noch intakt. Allerdings ist keine vollständige Sitzungs-Semantik gegeben, da Callbacks im Netz verloren gehen können. Dateisperren werden von AFS unterstützt, sind aber durch einen timeout gegen Dauersperrung gesichert. Zur Latenzminimierung werden Dateien in 64KByte-Blöcken übertragen. AFS Server sind im Leistungsvergleich schneller als NFS-Server.

7.7 Weitere verteilte Dateisysteme

Coda ist eine Weiterentwicklung von AFS (1990). Drei weitere Ziele wurden unter der Maxime „Konstante Datenverfügbarkeit“ verfolgt. Fehlertoleranz soll Blockierungen des Dateizugriffs bei Serverausfällen verhindern. Auch schreibbare Dateien sollen replizierbar sein und portable Geräte sollen stärker unterstützt werden, indem möglichst alle benötigten Daten stets auf dem lokalen Rechner liegen. Die Architektur mit Venus und Vice blieb unverändert.
DFS basiert ebenfalls auf AFS und ist Teil der OSF DCE. Im DCE eingesetzte Verzeichnisdienste:
CDS (Cell Directory Service): DCE, innerhalb Zelle.
GDS (Global Directory Service): DCE, falls zellübergreifend.
X.500 Directory Service, Standard CCITT/ISO.
LDAP (Lightweight Directory Access Protocol): Internet Standard

8 Sicherheit in verteilten Systemen

8.1 Sicherheit und Schutz

Bedrohung eines verteilten Systems

Verteilte Systeme besitzen zahlreiche Angriffspunkte für gewollte oder unbeabsichtigte Störungen des Betriebes, sowohl in Quelle, Senke (Dateien, Ports, Speicherinhalte) als auch auf dem Weg von Quelle zu Senke (Datenpakete):

Unterbrechen:
Es handelt sich um eine bewusste Störung des Übertragungssystems, indem z.B. das Netz aufgetrennt wird. Dies ist ein **Angriff auf die Verfügbarkeit** des Systems.

Abhören und Mitschneiden:
Erfolgt dies durch nicht autorisierte Teilnehmer, so handelt es sich um einen **Angriff auf die Vertraulichkeit** der Informationen.

Fälschen:
Werden Nachrichten während des Transports verändert, findet ein **Angriff auf die Integrität der Informationen** statt.

Hinzufügen, Generieren:
Ein nicht autorisierter Teilnehmer ahmt das Verhalten eines autorisierten Teilnehmers nach und erzeugt falsche Daten, so ist das ein **Angriff auf die Glaubwürdigkeit** des Systems.

Angriffsformen

Besitzt jemand Zugang zum System (i.allg. über den Kommunikationskanal), so kann er auf fünf Wegen (1 passiv, 4 aktiv) unautorisiert auf das System zugreifen:

Lauschangriff (eavesdropping):
Dies ist eine passive Angriffsform, die die Daten nicht verändert. Unbemerktes Abhören und Mitschneiden von Informationen zählt dazu. Der Angreifer kann nicht beeinflussen, was er mithört.

Maskerade (masquerading):
Der Angreifer gibt sich als jemand anderes aus, z.B. als eine andere Person unter Zuhilfenahme eines falschen Passwortes oder ein Prozess verwendet eine falsche ID, um einen anderen Prozess nachzuahmen.

Intrigieren (tampering):
Nachrichten können bei der Übertragung verfälscht werden, so dass beim Empfänger ein anderes Verhalten bewirkt wird als vom Absender beabsichtigt ist.

Wiederholen (replay):
Nachrichten können (z.B. durch Lauschen oder Maskerade) mitgeschnitten und später unverändert erneut gesendet werden. Der Angreifer muss die Nachricht (z.B. übertragenes Zugangspasswort) nicht entschlüsseln können und kann sich unautorisiert Zugang zum System verschaffen.

Verweigerung (denial of service):
Eine eingeschleuste Komponente verhindert angeforderte Diensteerbringungen, weil sie Nachrichten nicht weiterleitet oder umleitet oder verfälscht. Auch die Netzüberflutung (flooding) mit willkürlichen Nachrichten (Überlast) kann zur Dienstverweigerung führen, z.B. Email „I love you“, Internet-Wurm, Blaster, Spam.

Für die vier aktiven Angriffsformen muss ein **attackierendes Programm** in das System gebracht werden, bevor es aktiv werden kann. Dazu kann ein dem Angreifer bekanntes **Passwort** oder eine Umgehung der Autorisierung eingesetzt werden. Ebenfalls möglich ist das Tarnen eines Programms als legitimes Programm und das Einbringen über legale Wege (z.B. Email). Diese nennt man dann auch **Virus**-Programme. Diese Programme können (für den Angreifer) sinnvolle Aufgaben ausführen oder einfach Zerstörung anrichten, z.B. indem sie

sich immer weiter vervielfältigen und selbst weiter verbreiten. Virus ist ein Programm, das an ein legitimes Programm angehängt ist, insbes. bei Shareware-Programmen (hohe Verbreitung) ~> führt dann unkontrollierte Aktionen aus. Viren replizieren sich selbständig und infizieren immer mehr legitime Programme. Übertragung der Viren über Netze (z.B. Email) oder Datenträger (z.B. Diskette, CD). Bekannte Viren: Stuxnet $_{2010}$ (iranische Atomanlagen), Flame $_{2012}$ (Finanzwelt). In diese Kategorie fallen auch so genannte **Würmer**. Es sind eigenständige Programme, die im unkontrollierte Aktionen durchführen. Bekannte Würmer: Internet-Wurm (Morris), Blaster-A, Mydoom, Sasser. Eine vierte Variante sind **Trojanische Pferde**, die als oft normales funktionsfähiges Programm auftreten, im Hintergrund aber noch anderer Funktionen ausführen. Sie öffnen Hintertüren im System oder täuschen Anmeldevorgänge vor (spoof login), um den Angreifer mit Zugangsinformationen zu versorgen.

Vertrauenswürdiges System
An ein Vertrauenswürdiges System werden folgende Minimal-Anforderungen gestellt:
Abgesicherte Kommunikationskanäle:
Diese verhindern das unautorisierte Abhören, erschweren die Interpretation und sind durch **Kryptographie** (Verschlüsselung) realisierbar.
Gegenseitiges Misstrauen:
Es findet immer eine **Authentisierung** der Kommunikationspartner statt, um zu sichern, dass der Client ein rechtmäßiger Vertreter seiner Klasse ist und der Server authentisch ist.
Frische (Aktualität) der Nachrichten:
Es muss sichergestellt werden, dass Nachrichten nicht unbemerkt veraltet sind, um Replay-Angriffe zu verhindern. **Zeitstempel**, **Tickets** oder sog. **Nonces** (angehängte Zufallszahlen, anhand derer sich Nachrichten bestimmten Sitzungen zuordnen lassen) leisten das.

Hauptsächlich drei **Mechanismen** für Vertrauenswürdigkeit eingesetzt (i.allg. Kombination):
Verschlüsselung (Cryptography):
Nachrichten werden **verschlüsselt** und nicht im Klartext übertragen. Diese Methode wird auch verwendet, um Nachrichten digital zu signieren und somit sowohl den Autor eindeutig zu identifizieren als auch die Nachricht vor unbemerkter Veränderung zu schützen.
Authentisierung (Authentication):
Identität beteiligter Komponenten soll verifizierbar sein (z.B. mit geheimen u./o. öffentlichen Schlüsseln); Authentisierung oft über Authentisierungs- und Schlüsselverteildienste implementiert, die Bestandteil der vertrauenswürdigen Basis sind („Trusted Third Party“).
Zugriffskontrollmechanismen (Access Control):
Hier werden die **Rechte** für bestimmte Klassen von Teilnehmern (roles) verwaltet. Zugriffe auf Ressourcen des verteilten Systems sind nur bestimmten Teilnehmern gestattet. Die Relation wird in **Zugriffskontroll-Listen** (access lists) bzw. **Befähigungen** (capabilities) gespeichert, die bereits zum Sicherheitsaspekt der lokalen Betriebssysteme gehören.

8.2 Chiffrierung (Kryptographie)

8.2.1 Verschlüsselung (Codierung)

Die **Verschlüsselung** erfolgt durch eine Funktion, die auf die Daten und den Schlüssel (K) angewendet wird. Eine dazugehörige Funktion (F) zur **Entschlüsselung** liefert entsprechend die Daten zurück. Schreibweise: F(K,M) -> $\{M\}_K$
Schlüssel müssen sicher verteilt und sicher gespeichert sein. Verschlüsselungsverfahren:
- **symmetrische Verfahren** haben nur einen Schlüssel, der für beide Funktionen verwendet wird (private key, geheimer Schlüssel),
- **asymmetrische Verfahren** arbeiten mit einem öffentlichen und einem privaten Schlüssel (public key, öffentliche Schlüssel).

Aus diesem Grund werden Schlüssel durch einen vertrauenswürdigen **Schlüsseldienst** verteilt und sicher gespeichert. Die Schlüssel müssen nicht unbedingt alle geheim gehalten werden. Verfahren mit **öffentlichen Schlüsseln** (public key, asymmetrische Verfahren) veröffentlichen jeweils einen der beiden Schlüssel, wohingegen Verfahren mit **geheimen Schlüsseln** (private key, symmetrisch) dies verbieten.

8.2.2 Blockchiffrierer

Diese Verfahren teilen den Klartext in **gleich große Blöcke** und verschlüsseln diese, ohne die Blockgröße zu ändern (im Gegensatz zu den bitweise arbeitenden Stromchiffrierern).
Vier Betriebsarten, Electronic Code Book (ECB) Modus, Cipher Block Chaining (CBC) Modus, Cipher Feedback Chaining (CFC) Modus und Output Feedback (OFB) Modus.
Beispiele für Blockchiffrier-Verfahren: Data Encryption Standard (DES, 1973), Triple-DES, IDEA, Blowfish (1993) oder Cast (1996). Sie finden Anwendung in großem Umfang und gelten als sicher, obgleich prinzipiell eine Rückberechnung durch Überprüfen aller Kodierungsformen möglich ist.
Bedeutsam ist bei diesen Verfahren vor allem auch die **Schlüssellänge**. Früher wurden Schlüssel mit 40 oder 56 Bit eingesetzt (vorallem bei DES), heute sind 168 Bit (bei 3DES, auch Triple-DES) der Standard. Eine Tabelle zeigt den Einfluss der Schlüssellänge und der Operationsgeschwindigkeit (Preise 1998).

Schlüssellänge	Langsamer Pentium (ca. 1.000 $)	Dedizierte Hardware (ca. 100.000 $)	Dedizierte Hardware (ca. 1.000.000 $)
40 Bit	1 Monat	0,3 Sekunden	0,02 Sekunden
56 Bit	5000 Jahre	6 Stunden	35 Minuten
Dabei ausgeführte Tests pro Sekunde	200.000	255.000.000	$3 * 10^{13}$

Zum Vergleich: Supercomputer „Roadrunner“ (IBM, USA) mit mehr als 1 Billiarde Op./sek. (Petaflops), Platz 1 in Rankingliste (2008).

8.2.3 Verfahren mit geheimen Schlüsseln

Mehrere Nutzer, die alle im Besitz **eines gemeinsamen Schlüssels** sein müssen, den keiner außer ihnen kennt, können vertraulich kommunizieren, indem sie ihre Nachrichten mit einer Funktion F und dem Schlüssel K verschlüsseln, dann versenden und anschließend entschlüsseln. Beide Teilnehmer benutzen den gleichen geheimen Schlüssel. Ein **geheimer Schlüsselaustausch** ist in jedem Falle vor der (ersten) Kommunikation nötig. Die verwendeten Funktionen können öffentlich sein. Die Hauptangriffspunkte geheimer Verfahren sind der Austausch und die Speicherung der geheimen Schlüssel.

DES (Data Encryption Standard)
Eines der bekanntesten Verfahren mit geheimen Schlüsseln (1977: National Institute of Standards and Technology, USA). Blockchiffrier-Verfahren.
Eingabe zum DES: 64-bit Klartext und 56-bit Schlüssel. In 16 schlüsselabhängigen Runden werden Bitrotationen und zusätzlich 3 schlüssel unabhängige Transpositionen durchgeführt.
Ausgabe des DES: 64-bit verschlüsselter Text. Mit dem gleichen Schlüssel kann die Entschlüsselung erfolgen. DES insbes. für Umsetzung auf HW-Bausteine konzipiert. Für DES gibt es spezielle VLSI-Chips für die NW-Karte, um Daten in Echtzeit zu ver-/entschlüsseln.

Bekannte Verfahren mit geheimen Schlüsseln:
Blockchiffrier-Verfahren:
- DES (Data Encryption Standard): 64-bit Daten, 56-bit Schlüssel.

- Triple-DES: 64-bit Daten, 168-bit Schlüssel (Stallings, 1995),
- IDEA (International Data Encryption Algorithm): 64-bit Daten, 128-bit Schlüssel (Lai/Massey, 1990).

Stromchiffrier-Verfahren:

- SEAL: 160-bit Schlüssel
- RC4 (Ron Rivest, RSA Data Security): variable Schlüssellänge bis zu 2048 bit.

Hash-Funktionen

Längere Nachrichten werden auf einen Hashwert fester Länge abgebildet. Hash-Funktion: aus einem Text wird ein beliebig langes Klartextstück ausgewählt und daraus eine Bitfolge fester Länge berechnet ~> einwegige Hash-Funktion, sog. Message-Digest.

Anforderungen an Hash-Funktion: mit polynomialem Aufwand berechenbar, unumkehrbar und kollisionsfrei, d.h. zwei Nachrichten dürfen nicht auf den gleichen Hashwert abgebildet werden. Beispiele:

MD4, MD5 (Ron Rivest);

SHA (Secure Hash Algorithm), entwickelt von NSA - US-Geheimdienst,
SHA-Hash-Code ist 32 Bit länger als MD5, 232-fach sicherer als MD5, langsamer;
SHA ist US-Regierungsstandard (überarbeitete Version: SHA-1);

RIPE-MD 160 (EU-Projekt).

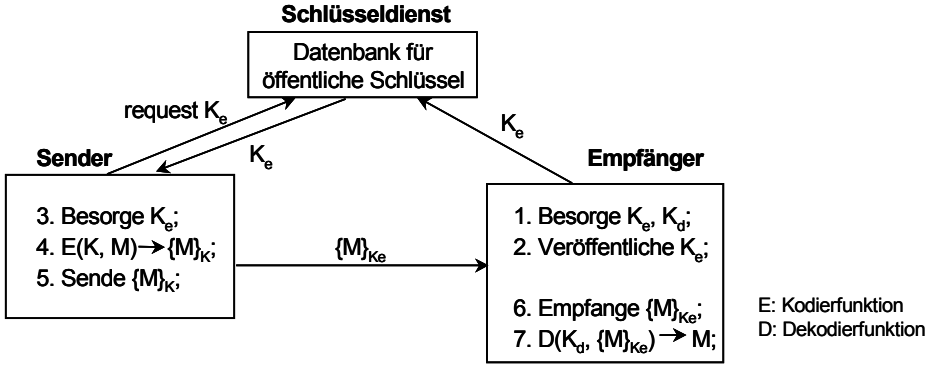
Zertifikate

Zertifikate dienen zur Authentisierung. Sie werden von einer Certification Authority (CA) signiert. Bei Bedarf fordert der Nutzer ein Zertifikat bei der CA an. Standardisierung wegen Interoperabilität erforderlich. Beispiel: X.509-Zertifikate.

8.2.4 Verfahren mit öffentlichen Schlüsseln (public key)

Man kann auf das Vertrauen zwischen den Kommunikationspartnern und den Schlüsselaustausch vor der Kommunikation verzichten, indem man **asymmetrische Verfahren** mit öffentlichem Schlüssel und privatem Schlüssel verwendet (Deering/Hinden, 1995, im Zusammenhang mit IPv6). Während der **öffentliche Schlüssel** verbreitet wird, muss der **private** geheim gehalten werden. Eine Anforderung an die Schlüssel ist, dass man den privaten nicht aus dem öffentlichen berechnen kann.

Ablauf einer Kommunikation mit öffentlichen Schlüsseln:



- Zunächst ermittelt Empfänger das Schlüsselpaar (K_e, K_d).
- Kodierschlüssel K_e dann beim Schlüsseldienst hinterlegt.
- Will ein Sender eine Nachricht zum Empfänger schicken, so besorgt er sich beim Schlüsseldienst diesen Kodierschlüssel und verschlüsselt mit der bekannten Kodierfunktion und dem Schlüssel die Nachricht.
- Verschlüsselte Nachricht kann nun nur noch beim Empfänger ausgepackt werden, denn er allein kennt den Dekodierschlüssel.

Abbildung 8.1: Verfahren mit öffentlichen Schlüsseln

Der Ablauf der Kommunikation ist dann leicht modifiziert. Aus seinem geheimen Schlüssel berechnet jeder Empfänger seinen öffentlichen Schlüssel (mittels einer Einwegfunktion) und stellt ihn den Kommunikationspartnern (öffentlich) zur Verfügung. Diese verschlüsseln ihre Nachrichten an ihn mittels des öffentlichen Schlüssels. Anschließend können die Nachrichten nur noch mit dem privaten Schlüssel entschlüsselt werden.

Mathematische Basis öffentlicher Schlüssel bilden das Produkt zweier sehr großer Primzahlen ($> 10^{100}$) und der Fakt, dass die Zerlegung in Primfaktoren sehr aufwendig ist und lange dauert. Ein potentieller Empfänger bildet Schlüsselpaar (K_e, K_d)

K_e: öffentlicher Schlüssel (e: encryption)

K_d: privater Schlüssel (d: decryption)

Mit K_e können die Sender Nachrichten kodieren, die nur der Besitzer des privaten Schlüssels (K_d) dekodieren kann Die beiden Schlüssel werden durch Verwendung einer Einwegfunktion (engl.: one-way-function) gebildet. Aus Kenntnis eines Schlüssels läßt sich der andere Schlüssel nur extrem aufwendig herleiten. Kodierfunktion E und Dekodierfunktion D dürfen öffentlich bekannt sein. Eine mit E und dem öffentlichen Schlüssel K_e verschlüsselte Nachricht läßt sich nur mit D und dem passenden privaten Dekodierschlüssel K_d entpacken.

RSA-Algorithmus

RSA: Verfahren nach Rivest, Shamir und Adleman (MIT/Boston, 1978). Anhand des **RSA**-Algotithmus soll hier die Funktionsweise von Verfahren beschrieben werden, deren Sicherheit auf dem Faktorisierungsproblem von großen Zahlen basiert. RSA verwendet öffentliche Schlüssel und Stromchiffrierung. Die Grundidee ist, dass es sehr leicht ist, das Produkt zweier Primzahlen zu berechnen, jedoch sehr schwierig, eine große Zahl in Primfaktoren zu zerlegen (Faktorisierung).

Erster Schritt:
Auffinden zweier Zahlen e und d, die als Kodier- und Dekodierschlüssel dienen.
Algorithmus:
1. Wähle 2 Primzahlen P und Q (jeweils $> 10^{100}$) und bilde
   $N = P \cdot Q$ und $Z = (P - 1) \cdot (Q - 1)$
2. Wähle d frei als eine Zahl, die relativ prim zu Z ist, d.h. wähle d so, dass
   ggT (d, Z) = 1 ggT: größter gemeinsamer Teiler
3. Bestimme e aus Lösung der Gleichung
   $e \cdot 1 = 1 \mod Z$ mit $e > Z$ d h . auch e muss relativ prim zu Z sein

Daraus 2 neue Schlüssel: Kodierschlüssel $K_e = < e, N >$
Dekodierschlüssel $K_d = < d, N >$

Zugehörige Funktionen: Kodierfunktion $E(e, N, M) := M^e \mod N = \{M\}$
Dekodierfunktion $D(d, N, \{M\}) := \{M\}^d \mod N = M$

Kodiert werden dann Nachrichtenblöcke, deren ganzzahlige Repräsentation kleiner N ist, d.h. ein Block mit k Bit kann kodiert werden, wobei $2^k < N$.

8.3 Authentisierung

Authentisierung (authentication, Beglaubigung) ist ein Vorgang, bei dem überprüft wird, ob ein Kommunikationspartner derjenige ist, der er vorgibt zu sein. Es handelt sich also um **Echtheitsnachweise** (Überprüfung der Nutzeridentität). Oft werden in verteilten Systemen separate **Authentisierungsdienste** eingesetzt, die auf eigenen Servern laufen, am besten in einem eigenen, separaten Dienst („3. Instanz“, „Trusted Third Party“, Authentisierungsserver, engl.: authentication server, auf Basis geheimer oder öffentlicher Schlüssel).

8.3.1 Needham-Schroeder-Protokoll

Verfahren

Kryptografisches Verfahren zur Authentisierung in verteilten Systemen.
Grundlage: ***Needham-Schroeder-Protokoll***

- sowohl für geheime Schlüssel (private key, symmetrisches Verfahren)
- als auch öffentliche Schlüssel (public key, asymmetrisches Verfahren – Nutzung eines Schlüsselpaares, bestehend aus geheimen und öffentlichen Schlüsseln).

Lit.: Needham, R.M.; Schroeder, M.D.: Using Encryption for Authentication in Large Networks of Computers. Communications of the ACM, vol. 21, 1978

Authentisierungsserver

Verwaltet eine Liste mit Tupeln der Art **<Name, geheimer Schlüssel>** für jeden Akteur (Benutzer u/o. Prozesse). Aufgaben des Servers:

- Sicheren Weg zum Erhalt eines Kommunikationsschlüssels bieten,
- Gegenseitige Authentisierung beider Kommunikationspartner.

Authentisierung mit geheimen Schlüsseln

Ein bekanntes Protokoll zur gegenseitigen Authentisierung zweier Teilnehmer ist das **Needham-Schroeder-Protokoll** für symmetrische Verschlüsselung (geheime Schlüssel). Der Authentisierungsdienst S kennt dabei die privaten Schlüssel der Teilnehmer A (Sender) und B (Empfänger). Verwendet wird außerdem ein Nonce N.

Aktion	Senderichtung	Nachricht	Erläuterung
1	A --> S	A, B, N_A	A möchte mit B kommunizieren, Anforderung eines Tickets. N_A ist eine beliebig gewählte, aber eindeutige Zahl.
2	S --> A	$\{N_A, B, K_{AB}, \{K_{AB}, A\}\}_{KA}$	Server sendet geheimem Kommunikationsschlüssel K_{AB} und ein mit B's Schlüssel K_B versehenes Ticket verschlüsselt an A. A traut S, weil außer A nur noch S den Schlüssel K_A kennt.
3	A --> B	$\{K_{AB}, A\}_{KB}$	A schickt das Ticket und Nachricht zu B, das nur B entschlüsseln kann.
4	B --> A	$\{N_B\}_{KAB}$	B entschlüsselt das Ticket und schickt seine Nonce, um zu prüfen, ob A authentisch ist oder Nachrichten wiederholt.
5	A --> B	$\{N_B - 1\}_{KAB}$	A bestätigt eine vereinbarte Rückantwort.

Eine **Schwachstelle** des Protokolls ist, dass Empfänger B in Aktion 3 nicht sicher sein kann, dass sein Ticket frisch ist. Ein Client könnte sich später als A ausgeben, falls er sich das Ticket besorgt hat. Als Lösung fügt man einen **Zeitstempel** zum Ticket hinzu: $\{K_{AB}, A, t\}_{KB}$. Diese Lösung wird beispielsweise im System Kerberos eingesetzt.

Authentisierung mit öffentlichen Schlüsseln

Verwendung des Needham-Schroeder-Protokolls für asymmetrische Verschlüsselung: jeder Teilnehmer hat einen öffentlichen Schlüssel (public key) und einen geheimen Schlüssel (private key). *Schwachstelle des Protokolls:* Teilnehmer A und B müssen glauben, dass die vom Authentisierungsdienst S kommenden öffentlichen Schlüssel (Nachrichten 2 und 5) frisch sind. Lösung: Absicherung durch Zeitstempel-Ergänzungen.
Verwendet werden die öffentlichen Schlüssel PK und geheimen Schlüssel SK, sowie A und B als Teilnehmer, S als Schlüsseldienst und N als Zufallszahl.

Aktion	Senderichtung	Nachricht	Erläuterung
1	A --> S	*A, B*	A möchte mit B kommunizieren, Anforderung öffentlicher Schlüssel von S.
2	S --> A	$\{P_{KB}, B\}_{SKS}$	S sendet öffentlichen Schlüssel von B (PK_B). Jeder kann mittels des öffentlichen Schlüssels PK_B entschlüsseln Die Übertragung wird verschlüsselt, um Tampering (Intrigieren) zu verhindern und um die Nachricht als von S kommend zu authentisieren.
3	A --> B	$\{N_A, A\}_{PKB}$	A schickt an B eine Nonce und Nachricht. Nur B kann das entschlüsseln.
4	B --> S	B, A	B fordert öffentlichen Schlüssel (zu A) von S.
5	S --> B	$\{PK_A, A\}_{SKS}$	B erhält öffentlichen Schlüssel von A.
6	B --> A	$\{N_A, N_B\}_{PKA}$	B prüft A durch die erhaltene und eine eigene Frischeinformation und Nonce.
7	A --> B	$\{N_B\}_{PKB}$	A ist tatsächlich A, weil nur A die Nachricht aus Aktion 6 entschlüsseln konnte.

8.3.2 Authentisierungsdienst Kerberos

Kerberos ist ein **verteilter Authentisierungsdienst** für offene und unsichere Systeme (z.B. Internet). Name der griechischen Mythologie entlehnt.
Entwickler: Steve Miller, Clifford Neuman (MIT), basierend auf Needham-Schroeder-Protokoll zur Authentisierung (1978). Aktuelle Version_{2010}: Kerberos 5, definiert in RFC 4120, Nutzung ASN.1 zur Codierung. Kerberos entstand im Rahmen des Athena-Projekts am MIT. Erst die Version 4 (Ende der 1980er Jahre) wurde auch außerhalb des MIT verwendet.
MIT bietet freie Implementierungen des Kerberos-Protokolls der Versionen 4 und 5 für Unix und Linux, mit verschiedenen Verschlüsselungsverfahren (wie DES, RC4 ...) und Prüfsummenverfahren (wie MD5, SHA-1, ...).
Kerberos bietet sichere und einheitliche Authentifizierung in einem ungesicherten TCP/IP-Netzwerk auf sicheren Hostrechnern. Die Authentifizierung übernimmt eine vertrauenswürdige dritte Partei (sog. Trusted Third Party), ein besonders geschützter Kerberos-5-Netzwerkdienst. Kerberos unterstützt Single Sign-on: Benutzer muss sich nur einmal anmelden, dann kann er alle Netzwerkdienste nutzen, ohne erneute Passwort-Eingabe. Kerberos übernimmt die weitere Authentifizierung (Lit.: Wikipedia).
Komponenten: Client, Server (der vom Client genutzt werden soll) und der Kerberos-Server (auf Client und Server muss jeweils ein Kerberos-Client installiert sein). Benutzer an Arbeitsplatzrechnern können gesichert die Dienste auf im Netz verteilten Servern nutzen.

Kerberos-Server: **Key Distribution Centre** (**KDC**)

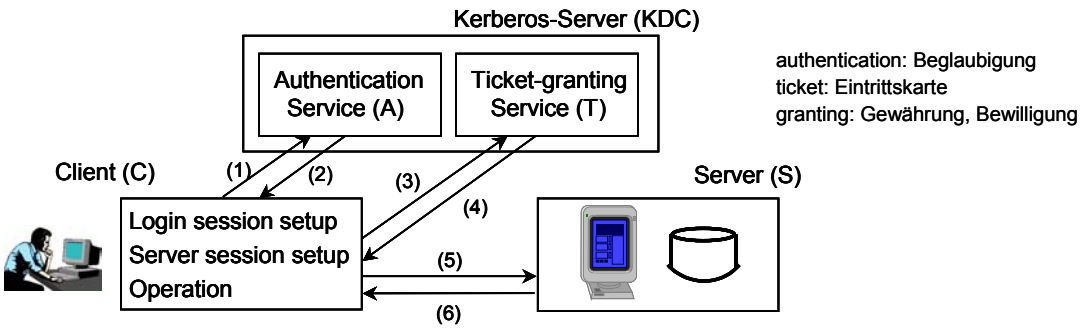


Abbildung 8.2: Architektur Kerberos

KDC wurde am MIT als Teil des Athena-Projektes entwickelt und basiert auf dem **Needham-Schroeder-Protokoll** mit geheimen Schlüsseln. Seine Bestandteile sind der Authentication Service (AS) und der Ticket-Granting-Service (TGS).
Funktionsweise: Client, Server (den Client nutzen will) und der Kerberos-Server.
Kerberos-Dienst authentifiziert sowohl den Server gegenüber dem Client, als auch den Client gegenüber dem Server (um Man-In-The-Middle-Angriffe zu unterbinden). Auch Kerberos-Server selbst authentifiziert sich gegenüber Client und Server und verifiziert deren Identität.. Kerberos verwendet Tickets zur Authentifizierung. Um den Kerberos-Dienst nutzen zu können, muss sich ein Client zuerst beim Kerberos-Server anmelden. Client fordert vom Kerberos-Server ein Ticket Granting Ticket (TGT) an (Passwort-Eingabe). Mit dem TGT kann Client auch weitere Tickets für weitere Dienste anfordern, ohne erneute Passwort-Eingabe.
•Für die Kommunikation zwischen Client und Kerberos-Server wird ein Session Key ausgehandelt, um den Datenverkehr zu verschlüsseln. Zur Nutzung eines Dienstes fordert Client ein weiteres Ticket an. Dieses Ticket sendet der Client dann an den Dienst. Dieser überprüft, ob er dem Client den Zugriff gestattet. Dazu wird der Sitzungsschlüssel vereinbart und die Identität von Client, Server und Kerberos-Server überprüft. Die Unterstützung von digitalen Zertifikaten auf Smartcards zur Authentifizierung im UNIX-Bereich ist in Entwicklung, bereits unterstützt von Windows 2000 und 2003 (stellt somit Hauptanwendung von Unternehmens-PKI‘s dar). Juni 2006: Verfahren von der IETF als Standard (RFC 4556) empfohlen. Durch Kerberos werden insbesondere Angriffe durch *passives Sniffing* unterbunden, aber auch *Spoofing, Wörterbuch-, Replay-* und andere *Angriffe* erschwert.

Es existieren drei **Sicherheitsobjekte** in **Kerberos**. Zum einen ist dies das **Ticket** $\{ticket(C, S)\}_{KS} = \{C, S, t_1, t_2, K_{CS}\}_{KS,}$ mit dem Namen des Clienten C, der Serveridentifikation S, dem Zeitintervall (t_1, t_2), während dessen das Ticket gültig ist und dem Session-Key für C und S, K_{CS}, der von Kerberos erzeugt wurde. Des Weiteren existieren der **Authenticator** $\{ C, t \}_{KCT} = \{ auth (C) \}_{KCT}$, der dem Server die Identität des Clients bestätigt (wird vom Client erzeugt) und der **Session Key** (geheimer Schlüssel), der zur verschlüsselten Kommunikation zwischen dem Client und einem Server dient.
Dreistufiges Protokoll:

- Login des Benutzers (Beginn),
- Authentisierungsphase pro Service und Server,
- Absicherung der Verwendung des Services (letzte Phase).

Die Authentisierung ist dabei vom Ablauf her äquivalent zu Verfahrensweise in der obigen Tabelle (Authentisierung mit privaten Schlüsseln). Während der Einlogphase tippt der Benutzer seinen Namen ein, der zum Kerberos-Authentisierungsdienst geschickt wird. Danach wird der Schlüssel KC aus dem Passwort des Nutzers gebildet, das beim Authentisierungsdienst gespeichert ist. Anschließend entschlüsselt das Login-Programm den enthaltenen Session-Key anhand des eingetippten Passworts. Dadurch erscheint das Passwort nicht auf dem Netz. Kerberos erfordert eine gemeinsame Zeitbasis (allerdings ohne hohe Genauigkeit), da jedes Ticket ein Gültigkeitsintervall enthält, das zeitabhängig ist.

8.4 Digitale Signaturen

Signaturen dienen zur Sicherstellung, dass eine Nachricht M tatsächlich vom Absender A kommt, also vor allem den folgenden Zwecken:

- bestätigen willentlichen Unterzeichnung (gleichgestellt der handschriftlichen Unterschrift),
- fälschungssicher (Authentifikation),
- nicht wiederverwendbar,
- das unterzeichnete Dokument ist nicht mehr veränderbar,
- kann nicht zurückgenommen werden.

Zur Bildung der Signatur wird ein geheimer oder öffentlicher Schlüssel verwendet. Eingesetzt wird dieses Verfahren vor allem bei gesicherter Email-Kommunikation (PEM: Privacy Enhanced Mail oder PGP: Pretty Good Privacy), elektronischem Handel, Bankverkehr udgl.
Oft ist nicht die Geheimhaltung, sondern die **Überprüfbarkeit der Authenzität** der Nachricht das Ziel. Hier werden häufig (polynomial berechenbare) **Hash-Funktionen** eingesetzt, deren Umkehrfunktion um Größenordnungen aufwendiger zu berechnen ist als sie selbst. Die Funktion wird auf den Klartext angewendet und das Ergebnis (der Hashwert, auch Digest) mit dem eignen Schlüssel verschlüsselt. Der Berechnungsaufwand ist weitaus geringer als bei vollständiger Verschlüsselung und daher auch auf rechenschwachen Geräten ohne Verschlüsselungshardware einsetzbar. Der Empfänger wendet dieselbe Funktion auf den empfangenen Klartext an, entschlüsselt den Digest und vergleicht die Hashwerte. Sind sie identisch, ist die Nachricht authentisch.
Beispiele für solche Digestfunktionen: MD5 (Ron Rivest), SHA (NSA).

8.5 Aspekte der Netzwerksicherheit

Pretty Good Privacy (PGP)
PGP ermöglicht die sichere Email-Kommunikation. Der Ursprung einer signierten Mail kann nicht geleugnet werden, da die Signatur mit dem privaten Schlüssel des Absenders erzeugt wird. Unter Annahme einer sicheren Verteilung der öffentlichen Schlüssel können nur der oder die Empfänger die Nachricht lesen.

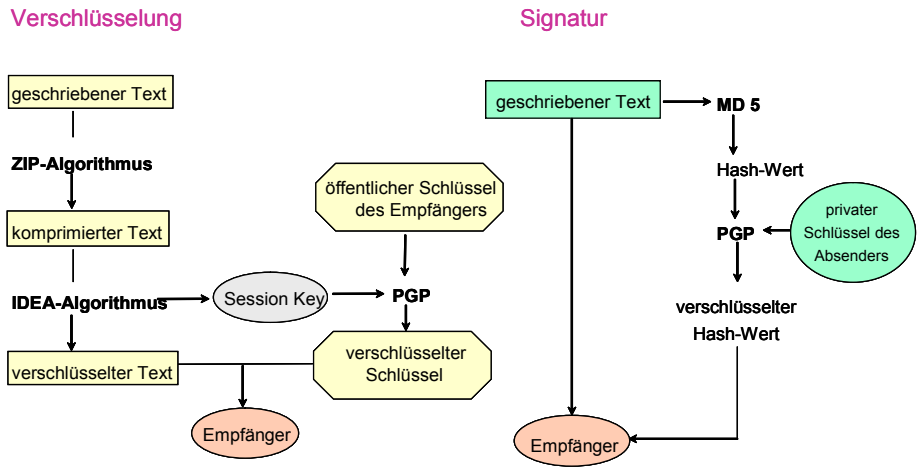


Abbildung 8.3: Pretty Good Privacy (PGP)

Verwendung finden symmetrische Verschlüsselungsverfahren (IDEA-Algorithmus) und Public-Key-Verfahren (RSA und DSS –Digital Signature Standard / Diffie-Hellmann)
PGP ist weit verbreitet und für private Nutzung kostenlos, bietet eine hohe Sicherheit durch anerkannte Verfahren, die Möglichkeit für eine eigne Schlüsselerzeugung und Wahl der Schlüssellänge. Trotz dieser Vorteile gibt es noch immer Hindernisse für eine flächendeckende Verwendung. So ist eine Schlüsselverwaltung notwendig und es ist schwierig, die Echtheit von Schlüsseln zu überprüfen.

Secure Socket Layer (SSL)
SSL ist eine **Technik zur sicheren Datenübertragung** über unsichere Verbindungen. Entwickelt wurde sie von Netscape und findet hauptsächlich in Web-Browsern Anwendung. Die Verbindung ist dann geheim, authentisch (über Zertifikate) und die Integrität der Datenpakete kann sichergestellt werden. In Zukunft soll das SSL Protokoll von **IPnG** ersetzt werden.

Im OSI-Modell wird SSL in einer zusätzlichen Schicht zwischen Transportschicht und Anwendungsschicht realisiert. Es besteht aus in zwei Subschichten mit **SSL-Handshake**-Protokoll und **SSL-Record**-Protokoll.
Eine **SSL-Sitzung** kann aus mehreren sicheren Verbindungen bestehen und enthält im Session State das Zertifikat des Partners, den Kompressionsalgorithmus, die Algorithmen zur Verschlüsselung und Berechnung der Message Authentication Codes (MAC) und das master secret, aus dem die benötigten Schlüssel berechnet werden. Der Connection State enthält dann u.a. noch einen geheimen Wert für die MAC-Operationen und die Schlüssel für die symmetrische Verschlüsselung.
Die **Record-Schicht** erhält uninterpretierte Daten von höheren Schichten in beliebig großen Blöcken. Diese werden dann in vier Schritten weiterverarbeitet (Fragmentierung, Komprimierung, MAC-Berechnung, Verschlüsselung).

8.6 Firewalls

Zugriffskontrolle und Überwachung
Eine Firewall ist ein Netzknoten, der ein gesichertes Netz an ein ungesichertes anschließt. Er überwacht den Datenstrom in beiden Richtungen und bietet Funktionen für die **Zugriffskontrolle**. Dabei soll die Firewall möglichst ungestörte Netzwerkzugriffe ermöglichen und trotzdem den gesamten Datenverkehr protokollieren. Die **Überwachung** umfasst das Erstellen von Zugriffsstatistiken (~> IDS), Anzeigen von Verbindungen und Melden von wiederholten Versuchen, die Firewall zu umgehen.
Beim Zugriff auf das Internet z.B. werden viele **Dienste mit Sicherheitslücken** eingesetzt, u.a. SMTP, FTP, HTTP und DNS. Durch geschickte Konfiguration der Firewall sollen diese Lücken geschlossen werden.
Nach außen hin bietet die Firewall Zugang zum Intranet über Schnittstellen, u.a. Telefon, ISDN, xDSL, paket- und datenvermittelte Leitungen. Für die Zugriffskontrolle gibt es nachgeordnete Kontrollmechanismen, anhand derer man Firewalls klassifiziert.
Unterschieden wird zwischen Firewalls, die auf der Netzwerkschicht, der Transportschicht oder der Anwendungsschicht aufsetzen. In dieser Reihenfolge nennt man sie **Paketfilter**, **Circuit-Relays** bzw. **Application Gateways**.

Paketfilter
Paketfilter filtern Datenpakete nach Kriterien wie Sende-, Empfangsadresse, Protokoll, Protokollport, benutzerdefinierte Bitmasken. Sie bewirkt bei korrekter Konfiguration einen ersten Schutz des internen Netzes. Bei komplexen Netzen werden Filtertabellen schnell unübersichtlich und fehlerbehaftet. Deshalb Paketfilter meist nur als Vorfilter für andere Firewall-Komponenten eingesetzt (wie Circuit-Relay, Application Gateways). Paketfilter filtern i.w. den Datenstrom der **Netzwerkschicht**.
Typen von Paketfilter-Firewalls:
* Begrenzungsrouter,
* Begrenzungsrouter mit gesichertem Zwischennetz,
* Dual Home Bastion Host mit Paketfilter.

Sowohl Begrenzungsrouter als auch Dual Home Bastion Hosts routen Pakete aus oder ins Internet. ***Einfache Begrenzungsrouter*** bieten i.allg. nur geringe Sicherheit. Begrenzungsrouter können durch ***abgesichertes Zwischennetz*** effektiver eingesetzt werden, in dem nur Systeme mit minimaler Ausstattung und gesperrten Diensten existieren, die die Funktionalität des Routers um Logging und Authentisierungsmechanismen erweitern.
Dual Home Bastion Hosts filtern Pakete nach IP-Adressen, Protokollart und Portnummer und nehmen dabei Pakete vor der Weiterleitung vollständig auseinander und prüfen dabei teilweise auch Nutzdaten. Sie bieten eine bessere Funktionalität und Sicherheit.

Weiterhin können Dienste, die häufig von außen genutzt werden (wie FTP oder WWW), auf Servern in einer sog. *Demilitarisierten Zone (DMZ)* zwischen äußeren Router und Bastian Host installiert werden

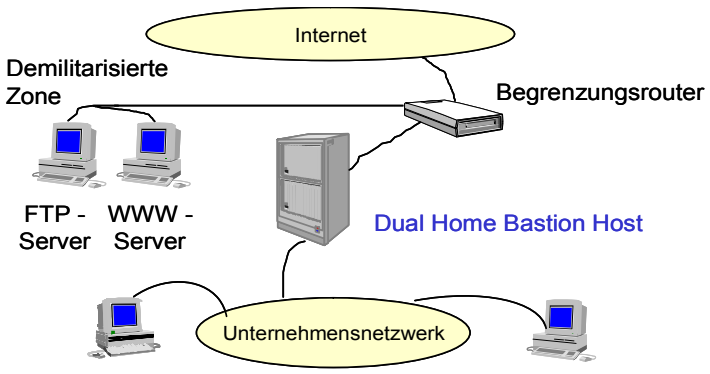


Abbildung 8.4: Paketfilter mit Dual Home Bastion Host

Circuit-Relays:
Diese Firewalls sind Endpunkte der Verbindungen aus beiden Netzen und bauen ihrerseits Verbindungen in das jeweils andere auf, um die Daten zu kopieren. Circuit-Relays sind i.allg in der **Transport-Schicht** angelagert und ermöglichen den Betrieb von TCP- oder UDP-basierten Applikationen, wie Web, Telnet u.a. Es handelt sich also um eine echte Trennung der Netze, da es keine durchgehenden Protokollverbindungen gibt. Als SOCKS sind leistungsfähige Produkte frei verfügbar und auch mit geringer Rechenleistung einsetzbar. Nachteilig wirkt sich aus, dass bei einem zwischengeschalteten Proxy (als solcher funktioniert die Firewall) die Client-Anwendungen angepasst werden müssen.

Application-Gateways:
Aus Client-Sicht verhält sich diese Firewall wie ein Server der jeweiligen Applikation, leistet aber dieselbe Trennung der Protokollverbindungen wie Circuit-Relays, nur ohne die Notwendigkeit, die Software anzupassen. Detaillierte Überwachungs- und Loggingfunktionen können zur Verfügung gestellt werden und werden mit hohen Rechenleistungen und komplexer Wartung bezahlt. Da dieser Firewall der Applikationskontext der übertragenen Daten bekannt ist, können die Daten effektiver klassifiziert und gefiltert werden. Application-Gateways bieten die beste Sicherheit.

Gegenwärtig verarbeiten gängige Firewalls 5 bis 10 MBit/s an Datentransfer. Neue Übertragungstechniken ermöglichen Transfers im Gigabit-Bereich und machen die Firewall ggf. zum Flaschenhals.

8.7 Intrusion Detection Systems (IDS)

Aufgaben von IDS
Es sind i.d.R. Hard- oder Software-Systeme zur Erkennung eines Missbrauchs von Computersystemen (Angriffs- und Einbruchserkennung: Intrusion Detection): Missbrauch (sog. *Angriff*) oder erfolgreicher Einbruch (*Intrusion*).
Aufgaben des IDS:
- Erkennung eines erfolgreichen Angriffs,
- Meldung des Angriffs, evtl. automatische Vereitelung des Angriffversuches.

Arten von IDS (Auswahl)
Netzwerkbasierte Intrusion Detection Systems (NIDS)
Analyse des Verkehrs auf dem anliegenden Netzwerksegment durch passives Abhören des Netzwerkverkehrs. Alle Pakete einer Verbindung werden gesammelt und auf Angriffsversuche untersucht. Typisches Beispiel: Portscans (auffällige Häufungen von TCP-Verbin-

dungsversuchen). NIDS überwachen i.allg. den Verkehr kompletter Netzwerke. Speziell: sog. HIDS (Hostbasierte IDS) überwachen den Verkehr an einer Maschine.
System Integrity Verifiers (SIV)
Überwachung Integrität wichtiger Systemdateien (z.B. /etc/shadow, Registry-Einträge). Meistens mittels Bildung von Prüfsummen über alle vom Administrator spezifizierten Dateien. Zusätzlich, ob ein normaler User root-Privilegien erreicht. I.allg. Warnmeldungen, z.T. Beenden verdächtiger Prozesse (z.B. root-Shell). SIV arbeiten hostbasiert (HIDS).
Beispiel: Tripwire.
Log File Monitors (LFM)
Überwachen der Log-Dateien relevanter Systemdienste. Beispiel: Parser, der Log-Dateien des HTTP-Servers nach bekannten Attacken durchsucht. LFM arbeiten meistens hostbasiert, können aber durch zentralisiertes Logging auch verteilte Systeme überwachen.
Beispiel: Swatch.
Deception Systeme (DCS)
Enthalten oder emulieren absichtlich Sicherheitslücken. Zielstellung: durch starke Überwachung versuchen, Angreifer zu fangen ~> einfache Programme zur Simulation eines Dienstes oder ganze Netzwerke von präparierten Servern.
Intention: Neue Angriffstechniken kennen lernen und sich dagegen abzusichern. Rechtsproblem, da sie als Angriffsplattform auf fremde Netzwerke dienen können.
Beispiele: Decoys, Lures, Traps, Honeyspots.

Passive Sicherheit
IDS als zusätzliche Sicherheitsmaßnahme, allein weniger sinnvoll. I.allg. passive IDS: Hilfe bei Erkennung, Meldung und Rückverfolgung von Angriffen. Auch aktive IDS verfügbar, aber mit potentiellem Risiko.
Ein umfassendes Sicherheitskonzept erfordert weitere HW/SW-Lösungen:
- Firewalls: Bilden meistens erste Abwehrlinie eines NW. Einfache Paketfilter bewerten nur Quelle und Ziel eines Paketes -> falls legitim, wird Verbindung erlaubt. IDS untersuchen dann Payload. Firewalls z.T. als aktive IDS eingesetzt.
- Authentisierung: Beglaubigung (Richtlinien bezüglich Passwörter, Netzwerkzugriff, ...).
- Verschlüsselung: Schutz vor Interpretation. Authentisierte, aber kompromittierte (d.h. abgehörte) Nutzerkonten sind vom IDS nicht mehr vom legitimen Verkehr unterscheidbar (analog VPN).
- Virenscanner: Regelmäßige Viren- und Trojanersuche (auf allen internen Host). Durch Trojaner initiierte, verschlüsselte Verbindungen zum Angreifer sind weder vom Firewall noch vom IDS erkenn- und verhinderbar.

Aufbau eines IDS
Da kommerzielle Systeme selten einen Einblick in die Architektur eines IDS erlauben, wurde deshalb „Common Intrusion Detection Framework (CIDF)“ entwickelt.
CIDF definiert Satz von allgemein üblichen Komponenten:
- Ereignisquelle (E-Box),
- Analyseeinheit (A-Box),
- Speicherung von Ereignissen (D-Box),
- Reaktions- und Gegenmaßnahmeeinheit (C-Box).

Diesem Rahmenwerk folgen die meisten der heutigen Implementierungen, auch freie Implementierungen, wie z.B. *Snort*.

Komponenten eines CIDF:
Ereignisquelle (E-Box): Sammelt „Ereignisse“ aus verschiedenen Systemschichten (am häufigsten NW-, Host-, Applikationsüberwachung).

Analyseeinheit (A-Box): Analysiert von E-Box gelieferte Daten, korreliert Daten aus verschiedenen Quellen, Herausfiltern relevanter Daten. Häufige Analysemethode: Erkennung von *Angriffssignaturen* und *Anomalien*.

Speichermechanismen (D-Box): E- und A-Box produzieren große Datenmengen, die schnell und zuverlässig zu speichern sind (z.B. für eine forensische Analyse eines Angriffs).

Gegenmaßnahmen (C-Box): IDS aus Sicherheitsgründen oft nur als passive Alarmsysteme konzipiert, aber viele kommerzielle IDS enthalten Gegenmaßnahmen (Angriffe verhindern/einschränken).

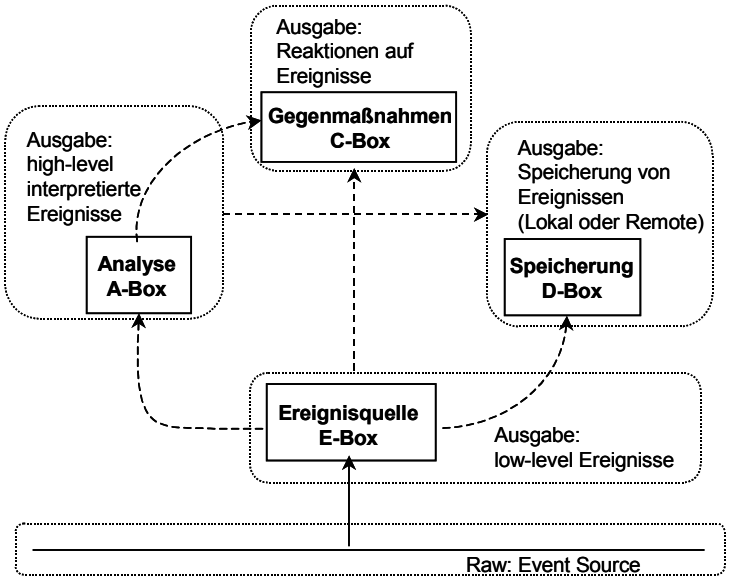


Abbildung 8.5: Common Intrusion Detection Framework (CIDF)

Einsatz eines IDS

IDS sind detailliert an den jeweiligen Standort anzupassen, um *Fehlalarme* („false positives“) und *unerkannte Angriffe* („false negatives“) zu minimieren.

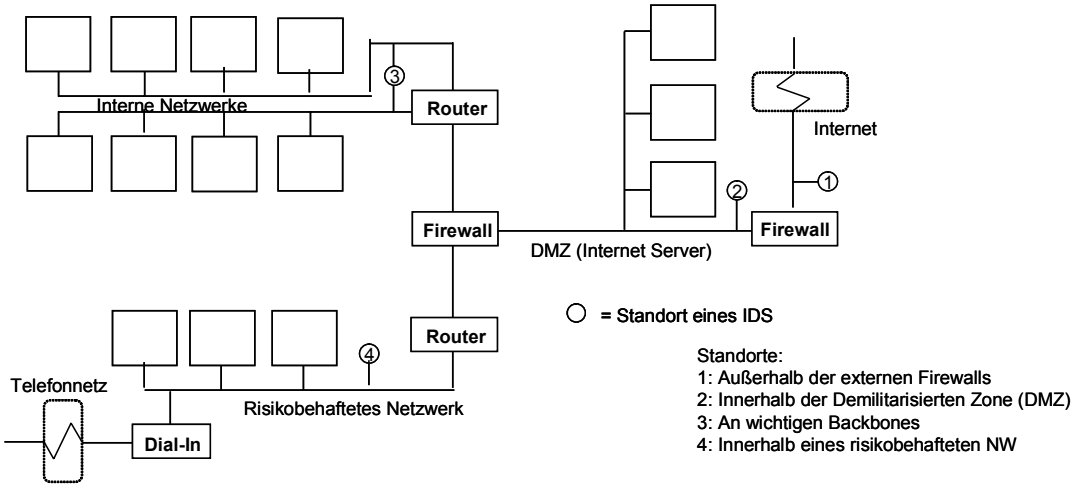


Abbildung 8.6: Einsatzszenario eines NIDS

9 Verteilungsplattformen

9.1 Middleware

Zielstellungen: Einheitliche Umgebung für Anwendungen, Verbergen der Heterogenität, Unterstützung heterogener Hardware / Software.
Realisierungen:

- Verteilte Betriebssysteme: Homogen, auf allen Rechnern installiert. Beispiele: Mach, Amoeba, Clouds, Birlix, ...
- Middleware (Verteilungsplattformen)
  * Plattform: Realisierung der Middleware-Technologien ~> SW-Komponente zwischen Basissystem (HW / SW, insbes. lokales Betriebssystem und Netzwerk) und Anwendung. Standardisierte Schnittstellen.
  * Dienste, u.a. Kommunikation und Koordination (-> RPC, RMI, MOM), zusätzlich Naming, Sicherheit, Persistenz, ... (-> DCE, CORBA, J2EE, .NET).
- Erweiterungen
  * Web-Technologien (AJAX), Web-Services.
  * Integrierte Technologien: SOA (Service Oriented Architecture).
  * Internet-Technologien: Cloud-Computing.

Im Rahmen dieses Scriptes werden ausgeführt (z.T. in extra Kapiteln, s. Inhaltsverzeichnis)
Entfernte Aufrufe (synchron/blockierend vs. asynchron/nicht-blockierend):
- Entfernter Prozeduraufruf: RPC (Remote Procedure Call). Sun, heute alle BS
- Entfernter Methodenaufruf: RMI (Remote Method Invocation): Sun, integriert in J2EE (Java 2 Platform Enterprise Edition), zuvor JDK (Java Development Kit)

DCE (Distributed Computing Environment): - historisch -
- Client/Server- / RPC- basiert ~> Ablösung durch objektorientierte MW (1995)
- Standard der Open Group / Open Software Foundation (OSF)

CORBA (Common Object Request Broker Architecture):
- Objektorientiert, sprachunabhängig: interoperabel (RMI-IIOP), dynamisch (DII)
- Standard der Object Management Group (OMG)

Componentware und Dienste-Architektur:
- MicroSoft: Active X, .NET (mit Komponentenmodell COM/DCOM, WCF)
- Sun u.a.: J2EE (mit Komponentenmodell EJB (Enterprise JavaBeans))
- SOA (Service-oriented Architecture); hier nicht explizit behandelt

Web- und Internet-Technologien:
- Web-Services (SOAP, WSDL, UDDI), Web-Technologien (JavaScript, AJAX), Cloud-Computing (s. auch Script_IntW3).

9.2 Distributed Computing Environment (DCE)

DCE ist eine Verteilungsplattform (Middleware) für heterogene verteilte Systeme, basierend auf einem **prozeduralen** Modell.

9.2.1 OSF - Open Software Foundation

Open Software Foundation (OSF) 1988 gegründet, Sitz in Cambridge (Massachusetts), offene Ausschreibungen RFT (Request for Technology). Bekanntestes Produkt:
DCE (Distributed Computing Environment), mit den Eigenschaften:
- Plattform für Client / Server-Anwendungen: dezentral, skalierbar
- Prozeduraler, RPC-basierter Ansatz, Interoperabilität
- Security Service, Transaktionsschnittstelle, einfache objektorientierte Erweiterungen

- Verfügbar für alle wesentlichen Systeme (Implementationen für die meisten gängigen Betriebssysteme, wie Unix (z.B. OSF/1 von DEC), VMS, Windows, OS/2).

9.2.2 DCE Architektur

Architekturmodell

DCE setzt auf lokalem Betriebssystem und zugehörigem Transportdienst auf.

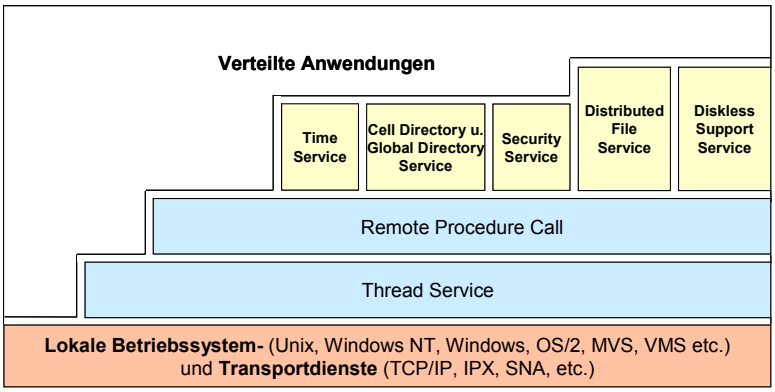


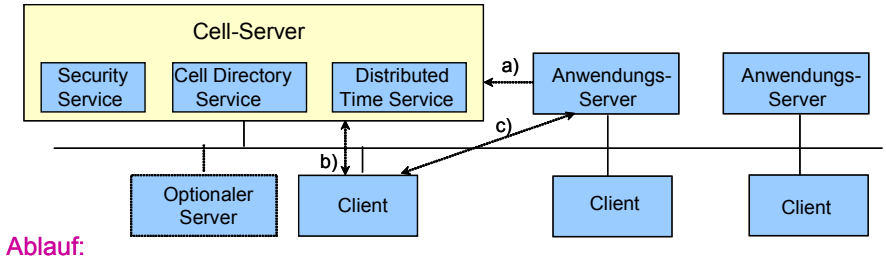
Abbildung 9.1: DCE Architekturmodell (Treppenmodell)

Es bietet auch einen Thread Service, falls das lokale Betriebssystem keinen Threaddienst enthält. Ansonst werden die DCE-Threads auf die Threads des lokalen BS abgebildet.

DCE Systemkonfiguration

DCE teilt alle Komponenten nach organisatorischen oder funktionalen Gesichtspunkten zueinander disjunkten Zellen zu, so dass jede dieser Zellen (Cells) eine Gruppe von Benutzern, Rechner und Ressourcen und somit die Basiseinheit der DCE-Infrastruktur bildet. Alle Knoten müssen Threads und RPC unterstützen.
Auf Cell-Ebene sind Namensverwaltung, Sicherheit und Administration implementiert, wobei zellenübergreifendes Arbeiten möglich, aber aufwendiger als zellinternes Arbeiten ist.
Struktur einer DCE-Cell:

- Mindestens ein Cell-Server mit den Basisdiensten CDS, Security Service, Distributed Time Service.
- Anwendungsserver, Clienten (Client-SW, Funktionsbibliotheken).
- Optionale Server, u.a. für GDS, DFS (Distributed File System), Diskless Support.



Ablauf:

a) *Service Export:* Registrierung eines Dienstanbieters / erbringers (Server) im CDS.
b) *Service Import:* Dienstnutzer (Client) erfragt Dienst mittels Identifikator (Service Request) und erhält Server-Interface (Service Identification).
c) *Binding:* Aufruf des Diensterbringers (Remote Procedure Call).

Abbildung 9.2: DCE Zellstruktur

9.3 Common Object Request Broker Architecture (CORBA)

9.3.1 OMG - Object Management Group

OMG: Konsortium namhafter Hersteller (SunSoft, DEC, HP, ObjectDesign, NCR, HyperDesk u.a.). Sitz in Cambridge (Massachusetts), offene Ausschreibungen (RFTs). Zielstellungen: Konzeption und Entwicklung verteilter Anwendungen auf Basis verteilter, objektorientierter Modelle, Spezifikation interoperabler SW-Systeme (heterogen) und Gestaltung eines ORB (Object-Request-Brokers) zur transparenten Kommunikation zw. verteilten Objekten.
Ergebnisse:
- **OMA** (Object Management Architecture) als Referenzarchitektur,
- **CORBA** (Common Object Request Broker Architecture) als ORB-Softwarebus.

9.3.2 CORBA Architektur

OMA-Konzept
Spezifikation für eine objektorientierte Middleware (context awared):
- ORB (Object Request Broker) als Kern sowie
- plattformübergreifende Protokolle und Dienste.

CORBA-Implementierungen vereinfachen das Erstellen verteilter Anwendungen in heterogenen Umgebungen.
- CORBA nicht an eine bestimmte Programmiersprache gebunden, somit ORB-Implementierungen verschiedener Hersteller möglich (unterschiedliche PS, Betriebssysteme).
- Client/Server-Architektur, insbes. für Java- und C++ -basierte Anwendungen.
- Formale Spezifikation der Schnittstellen (Daten, Methoden) mittels Interface Definition Language (IDL). Über IDL-Compiler und mitgelieferter Bibliotheken (enthalten u.a. ganze Interprozesskommunikation) werden Client-Stubs und Server-Skeletons generiert, die an die Client- bzw. Server-Implementationen anzubinden sind.
- Zusätzliche Bereitstellung von Diensten (Services & Facilities).

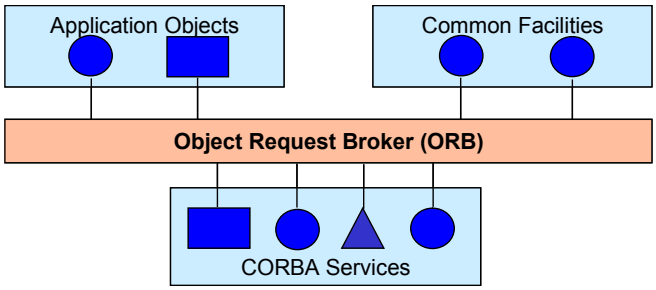


Abbildung 9.3: Allgemeine CORBA-Architektur

Merkmale
Objektorientierter Ansatz, Vererbung, Wiederverwendbarkeit. Implementierung teilweise auf DCE aufsetzend. Schnittstellen beschreibungssprache IDL (Interface Definition Language).
Language Mapping unter anderem für C, C++, Java, Smalltalk, Ada, Fortran, Cobol.
Durch Java (RMI) Möglichkeit der Anbindung von Intranet/Internet-Lösungen, insbes. Web.
Interoperabilität (GIOP, IIOP, RMI-IIOP). Dynamische Aufrufschnittstelle (DII).
CORBA 3.0: mit Komponentenmodell (CCM: CORBA Component Model).

Architekturkonzept
ORB (Object Request Broker)
- Softwarebus zur Interaktion zwischen den Objekten (lokale und entfernte Objekte, Services, Facilities).

- Sichert **Interoperabilität** zwischen Anwendungen auf verschiedenen Computern in *heterogenen, verteilten* Umgebungen und sichert Interoperabilität von Implementierungen in *verschiedenen Programmiersprachen* (u.a. C, C++, Smalltalk, Java, Ada, Fortran, Cobol).
- ORB-Implementierungen verschiedener SW-Hersteller möglich, unabh. von Programmiersprache und Betriebssystem (liefern ORB, IDL, IDL-Compiler, Objektbibliotheken) ~> Entlastung des Anwendungsprogrammierers.
- Kommunikation innerhalb ORB erfolgt mittels herstellerspezifischer Protokolle.
  Kommunikation zwischen unterschiedlichen ORB: IIOP (Internet Inter-ORB Protocol).

Objekte, Aufrufschnittstellen, Repositorium
Erweiterter Mechanismus zur Interaktion zwischen verteilten Objekten (Methodenaufruf).
Funktionalität (analog RPC): Entfernter Aufruf durch Client, Weiterleitung (ORB), Aufrufausführung (Server), Ergebnisrückgabe (ORB).
Programmentwicklung: IDL, IDL-Compiler (-> Stubs, Skeleton), Client/Server-Objekte

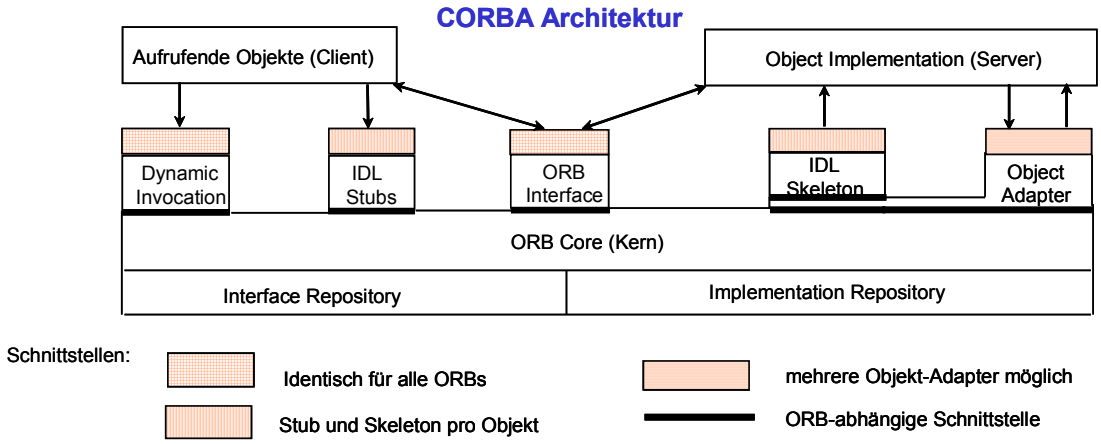


Abbildung 9.4: Aufrufschnittstellen in CORBA

Clients und Objekt-Instanzen
Clients sind rufende Objekte (Client-Rolle), sie greifen auf Methoden anderer Objekte zu. Objektimplementationen ("Server") sind gerufene Objekte.
Objektschnittstellen:
- statisch: spezifiziert über **IDL (Interface Definition Language)**.
- dynamisch: während Laufzeit (dazu IDL-Spezifikation im **Interface Repository** des ORB abgelegt, damit diese zur Laufzeit bekannt ist).

Ein Zugriff auf die Objekte durch den ORB erfolgt dann über Objekt-Adapter: BOA (Basic Object Adapter), POA (Portable Object Adapter, ab Corba 3.0).

Systemdaten
Zur Speicherung persistenter Objekte und der übersetzen Schnittstellenbeschreibungen dient das Interface Repository. Damit kann über das Dynamic Invocation Interface (DII) dynamisch zur Laufzeit eine Zuordnung von Interface und Implementierung erfolgen.
Das Implementation Repository speichert die physikalischen Netzadressen von Objektinstanzen, protokolliert Objektaktivitäten und enthält Management-Informationen.

9.3.3 Schnittstellen und Methodenaufrufe

Interface Definition Language (IDL)
Zur Nutzung des ORB als Mittler zwischen Client- und Serverobjekten ist das Objekt (Datenstrukturen und Methoden) in der CORBA-IDL zu beschreiben.

IDL: Interface Definition Language
- Syntax an C++ angelehnt
- Strenge Typisierung
- (Mehrfache) Vererbung, jedoch keine Redefinition von Operationen
- Synchrone und asynchrone Aufrufweise (auch asynchron mit Ergebnisrückgabe)
- Statische und dynamische Aufrufweise
- Ausnahmebehandlung

Verschiedene Sprachanbindungen (Language Mapping), u.a. für C, C++, Smalltalk, Java, Ada, Fortran, Cobol. Übersetzte Schnittstellenbeschreibungen werden in lesbarer Form im Interface Repository abgelegt ~> Unterstützung des dynamischen Aufrufmechanismus (DII: Dynamic Invocation Interface).
Mit der IDL wird eine formale Spezifikation der Schnittstellen (Daten und Methodensignaturen) erstellt, die eine Serveranwendung für entfernte oder lokale Zugriffe zur Verfügung stellt. Die IDL-Beschreibung wird dann mit einem IDL-Compiler in ein Objektmodell der ver-wendeten Programmiersprache umgesetzt (IDL-Compiler vom Hersteller mitgeliefert).
Aus der IDL wird der Quelltext generiert, der zur jeweiligen ORB-Implementierung passt. Quelltext enthält die Stubs und Skeletons.
Verwendetes Architekturmuster: sog. Broker Pattern
- verbirgt die Komplexität der Netzwerkschnittstelle und
- lässt Methodenaufruf wie einen lokalen Aufruf erscheinen.

Kommunikation innerhalb des ORB erfolgt auf Basis herstellerspezifischer Protokolle [Kommunikation mit anderen ORB: GIOP (seit Corba 2.0), insbes. IIOP ~> RMI-IIOP].

Programmentwicklung
Erzeugung von Quellcode: mittels IDL-Compiler (analog zu RPC). Compiler erzeugt zugehörige Clientstubs und IDL- (bzw. Server-) Skeletons (sie entsprechen den Serverstubs).
Beide Stubs werden mit entsprechendem Client- bzw. Servercode gebunden. Stubs verfügen dann über ORB-spezifische Schnittstellen, um mit diesen zu kommunizieren (somit Art der ORB-Implementierung dem jeweiligen Hersteller überlassen). Die meisten CORBA-Implementierungen unterstützen Java und C++ bzw. auch andere PS.
Außer der Generierung von Stubs kann die IDL-Spezifikation auch in das Interface Repository des ORB abgelegt werden, um diese dynamisch während der Laufzeit zu verwenden. Somit unterstützt CORBA sowohl eine *statische* als auch *dynamische* Aufrufvariante.
Stubs der generierten Klassen können wie normale lokale Objekte genutzt werden. Die Aufgaben der Interprozesskommunikation übernehmen die generierten Klassen und die von der CORBA-Implementation mitgelieferten Bibliotheken. Beispiel:
- Entwickler einer C++ -Server-Anwendung definiert zuerst seine IDL-Schnittstellen,
- danach erzeugt er mit dem IDL-Compiler die C++-Skeleton-Klassen,
- danach erweitert er die Skeletons mit der notwendigen Implementierung der Logik.

Ein Client-Entwickler benutzt die IDL-Schnittstelle des Server-Entwicklers und erzeugt mit seinem IDL-Compiler die Stubs im Quelltext seiner Programmiersprache (er kann die Instanzen der generierten Klassen wie normale Objekte nutzen). Damit Aufwand der Client/Server-Entwicklung reduziert, Details der IPC bleiben für Client und Server verborgen.

Methodenaufrufe
Für den CORBA-Aufrufmechanismus stehen bereit:
- IDL Stubs, IDL Skeleton (Server Skeleton): Stubs wie bei RPC.
- Dynamic Invocation Interface (DII): Aufruf ohne vorherige Stub-Generierung, Schnittstelleninformation zu Laufzeit.

- Object Adapter: Abbildung von Objektreferenzen auf konkrete Implementierung. Dynamische Methodenauswahl (BOA: Basic Object Adapter ~> POA: Portable Object Adapter, ab Corba 3.0).
- ORB Interface: Zugang zu Hilfsdiensten (z.B. Naming, Schnittstelleninformationen).

CORBA unterstützt 2 Aufrufvarianten

Statischer Objektaufruf

Bei statischen Aufrufen, muss zur Compilezeit feststehen, welche Schnittstellen ein Client verwenden will und er muss sich vor dem Aufruf über Namens- und Verzeichnisdienste eine Objektreferenz für einen passenden Server beschaffen.
Client verwendet die aus IDL in seiner Implementierungssprache abgebildeten Objektaufrufe, um Methoden des Serverobjekts aufzurufen. Aufruf an ORB weitergeleitet. Für jede Objektmethode steht ein spezifischer Aufruf zur Verfügung.
Statische Aufrufe sind synchron und blockieren, d.h. Client blockiert bis Ergebnis zurückgeliefert ist. Ausnahme: Einwegnachrichten (diese stellt ORB nach dem *Best-Effort-Prinzip* zu). Client erhält dazu keine Quittung. Einwegnachrichten werden durch das *one-way-Attribut* in der IDL gekennzeichnet.

Dynamischer Objektaufruf

Neben Clientstub kann Client auch die dynamische Aufrufschnittstelle des ORB verwenden: Dynamic Invocation Interface (DII):
- Schnittstelle bietet objektunabhängige Aufrufe an, um zur Laufzeit Aufrufe an Objekte dynamisch zusammenzustellen.
- ORB nutzt dazu die im Interface Repository abgelegte IDL-Spezifikation.

Client kann damit auf neue Objekttypen und Methoden zugreifen, die zur Laufzeit noch nicht bekannt waren.
Auf diese Weise kann auf neue Server zugegriffen werden, ohne erneut Stubs zu übersetzen, sofern sie die gleiche Schnittstelle exportieren.
Dynamische Objektaufrufe können nach 3 verschiedenen Semantiken abgewickelt werden: synchron, verzögert synchron (macht ein Polling der aktiven Requests notwendig) oder als Einwegnachricht (one-way-Attribut in IDL).

Um den Aufruf zusammenzustellen, muss der Client zunächst ein Requestobjekt erzeugen

```
CORBA_Object_create_request
```

und mit den entsprechenden aktuellen Parametern füllen. Die Parameter werden dazu in einer Liste von Paaren der Art <Name, Wert> abgelegt.
Zu einem Objekt kann Client über die DII-Routine

```
CORBA_ORB_create_operation_list
```

diese Liste aus dem Interface Repository auslesen.
Auch möglich, die Liste sukzessive zu ergänzen, um z.B. Werte dynamisch einzulesen.

```
CORBA_request_add_arg
```

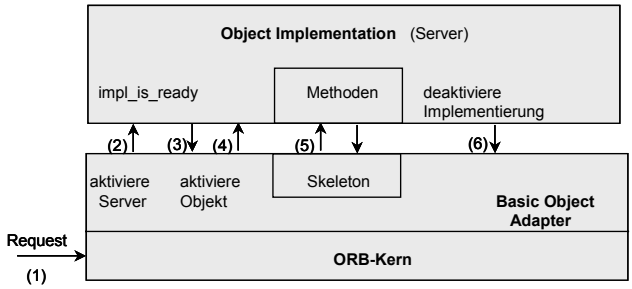
9.3.4 ORB, Object Adapter und Interoperabilität

ORB (Object Request Broker)

Anfragen aller Clients nicht direkt an Server geleitet, sondern über ORB: entweder über Clientstub oder über dynamisches Interface durch Nutzung DII. ORB ist Komponente, die Client-spezifische und Server-spezifische Aufgaben erfüllt. Für den Client bearbeitet der ORB die Anfragen. Zu jedem Request prüft ORB die Parameter gegen die IDL-Spezifikation. Der Request wird dann einem entsprechenden Server zugeordnet; falls dieser nicht läuft, wird er vom ORB gestartet. Falls mehrere Objektimplementierungen vorliegen, wählt ORB dientsprechende aus (“Trading”).

Je nach Aufrufsemantik regelt ORB den synchronen bzw. asynchronen Aufruf (im asynchronen Fall muss ORB die Antwort bereithalten, bis Client diese abholt). Am ORB-Interface werden den Clients generische Funktionen angeboten, um z.B. Name-Werte-Listen zu erzeugen. Die Client-spezifischen Teile des ORB werden üblicherweise zum Client gebunden.
Auf Serverseite empfängt ORB Objektaufrufe und aktiviert dazu die entsprechenden Objektmethoden. Dazu Parameter aus der Nachricht entpackt (engl.: demarshalling) und dann wird im IDL-Skeleton (Server-Skeleton) die entsprechende Methode angesprungen. Liegen Rückkehrparameter vor, so werden diese verpackt (engl.: marshalling) und entsprechend der Aufrufsemantik zum Client zurückgeleitet.

Object Adapter
Der ORB stellt den Servern Objektadapter zur Verfügung, mit Funktionen z.B. für Generierung und Interpretierung von Objektreferenzen, Methodenaufruf, Sicherheitsfunktionen, Objektaktivierung und- deaktivierung, Zuordnung von Referenzen zu Implementierungen, Registrierung von Implementierungen im Interface Repository. Jeder ORB muss einen BOA (Basic Object Adapter) besitzen, weitere Adapter können vom Hersteller ergänzt werden. Ab CORBA 3.0 unterstützt der POA den Persistenzmechanismen (Portable Object Adapter, Erweiterung des BOA).



Typischer Ablauf:
(1): ORB empfängt einen Request vom Client.
(2): BOA aktiviert die entsprechende Objektimplementierung, falls diese inaktiv ist.
(3): Objektimplementierung (d.h. Server) führt Initialisierungsschritte aus und meldet dem BOA, dass sie bereit ist (CORBA_BOA_impl_is_ready).
(4): Die Objektreferenz wird vom BOA an Server übergeben.
(5): Server wickelt daraufhin über das Skeleton und seine zugehörigen Methoden den Auftrag ab.
(6): Danach kann Server das Objekt deaktivieren (CORBA_BOA_deactivate_obj) oder komplett herunterfahren (CORBA_BOA_deactivate_impl).

Abbildung 9.5: BOA-Funktionen beim Objektaufruf

ORB Interoperabilität
Die Kommunikation innerhalb eines ORB erfolgt mittels eines herstellerspezifischen Protokolls. Aber: ORB nicht nur für Interaktionen zwischen Client und Server auf lokalen Systemen. Verteilte heterogene Anwendungen erfordern transparente Interaktionen zwischen den Clients und Servern über unterschiedliche ORB's.
Ab CORBA 2.0 ist das **IIOP (Internet Inter-ORB Protocol)** definiert. Es ermöglicht Kommunikation zwischen unterschiedlichen ORB's. Somit Requests von einem ORB an anderen ORB weitergeleitet, bei denen ein Server Objekte im Implementation Repository anbietet.
IIOP auch als Kommunikationsprotokoll in RMI/Java integriert (RMI-IIOP). Damit Kommunikation zwischen CORBA und Java / Web möglich.
IDL und IIOP bilden den Kern des CORBA-Standards.

Jeder ORB muss das General Inter-ORB Protocol (GIOP) unterstützen, das insgesamt sieben Protokollelemente anbietet, um Requests und zugehörige Antworten zu prüfen und zu übermitteln. Das GIOP benötigt ein verbindungsorientiertes, zuverlässiges Transportsystem und legt die Kodiervorschrift verbindlich auf Common Data Representation (CDR) fest.

Standardisiert ist das Internet Inter-ORB Protocol (IIOP). Dadurch auch Java-Anwendungen integrierbar: Über IIOP können Java-Objekte die Corba-Objekte, -Dienste und -Facilities nutzen. In Java-RMI ist das RMI-IIOP-Protokoll integriert. Mittels IIOP kann Anbindung von Internet und Intranet in einer CORBA-Umgebung realisiert werden.

CORBA Component Model (CCM)
Fortgeschrittene Anwendungen werden mit CORBA 3.0 möglich, mit Hilfe der Möglichkeit, Legacy-Anwendungen einzubinden. Kern der Neuerungen ist das Object Packing Schema für EJB (Enterprise Java Beans). Es spezifiziert, wie aus EJB CORBA-Komponenten werden. Weitere Ergänzungen sind Realtime-Erweiterungen, Wertübergabe eines Objektparameters (Object by Value), Mehrfachschnittstellen und Verbesserungen am IIOP.

9.3.5 CORBA Services und Facilities

CORBA Services

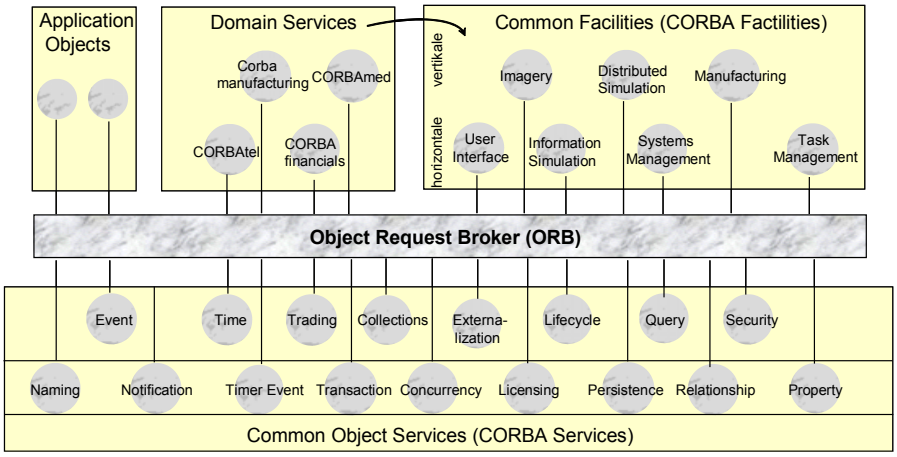


Abbildung 9.6: CORBA Services

CORBA Facilities
Es handelt sich um allgemeine Dienste auf der Anwendungsebene, unterteilt in anwendungsübergreifende Horizontal Facilities und anwendungsspezifische Vertical Facilities.
Horizontal Facilities:
User Interface (Benutzerschnittstelle): Dienste an Benutzerschnittstelle (grafische OF).
Information Management (Informationsmanagement): Dienste, um Informationen zu modellieren, zu definieren, zuzugreifen und auszutauschen (z.B. Dokumentverwaltung, WfMS).
System Management: Zur Systemverwaltung komplexer, heterogener Rechnerinfrastrukturen.
Task Management (Aufgabenmgt.): Dienste zur Automatisierung von Arbeitsprozessen.
Business Object Facility: Spezialisierung verschiedener Verteilungsplattformen bezüglich Architekturmodell, Technologie und Beschreibungsmechanismus.
Vertical Facilities: Telekommunikation (CORBAtel), Gesundheitswesen (CORBAmed), Finanzwesen (CORBAfinancials), Produktion (CORBAmanufacturing), Geschäftsbereiche, Öl- und Gasindustrie u.v.a.m.
CORBA Produkte *(Auswahl)*
ORBIX (Iona), Visibroker (Inprise), Distributed Smalltalk (HP), ORBeline (Netscape), ...

Einschätzung CORBA
Leistungsfähige Plattform für komplexe verteilte Anwendungen, viele Dienste. Aber viele der Services nicht realisiert – dagegen in Java- und C#-Bibliotheken angeboten.
Andere Möglichkeiten: Componentware (EJB, DCOM), SOA, Web-Services, Cloud-Computing.

10 Web-Services

10.1 Web-Technologien

Web-Services (Überblick)

Web-Services unterstützten die Gestaltung von Web-Anwendungen auf Knoten eines verteilten Web-Systems. Zugriff auf Web-Service wird über das Middleware-Protokoll **SOAP** realisiert, ein XML-basiertes Protokoll auf der Grundlage eines entfernten Prozeduraufrufes. Dienste der Web-Services werden über Schnittstelle gekapselt (WSDL). Standards:

- SOAP: Zugriffsprotokoll für Web-Services.
- WSDL: Schnittstellenbeschreibungssprache.
- XML: Deskriptive Sprache zur Beschreibung und Austausch komplexer Datenstrukturen, eingesetzt für SOAP und WSDL.
- UDDI: Verzeichnisdienst zur Veröffentlichung von Web-Services.

Realisierungsplattformen für Web-Services: Java (JAX-RPC), .NET, PHP. Seit ca. 2005 auch Werkzeuge für Entwicklungen mit C, C++ verfügbar.
Erweiterung der Web-Technologien mit AJAX (Asynchronous JavaScript and XML).

Charakteristika

Web-Services erbringen heterogene verteilte Dienste, benutzbar in beliebigen Anwendungen. Nutzung der *Infrastruktur* des Internets (z.B. HTTP, SMTP, Web-Server) für Transport von Daten und der *Metasprache* XML für Beschreibung der Daten.
Web-Services bilden eine der Basismethoden zur Realisierung der integrierten Diensteplattform für Geschäftsmodelle: **SOA (Service Oriented Architecture)**. Neben Web-Services auch andere Verfahren einsetzbar.
Web-Services bilden ein Netz von Softwarediensten, erstellt und verwendet unabhängig von Betriebssystem, Programmiersprache und binärem Übertragungsprotokoll.
Wesentliche Unterschiede zu anderen oo-Modellen:

1. Web-Services definieren für die Kommunikation kein neues Binärprotokoll (wie IIOP oder TCP/IP), sondern verwenden das XML-basierte Protokoll SOAP.
2. SOAP definiert kein eigenes verteiltes Objektmodell, sondern unterstützt beliebige in XML beschreibbare Datenstrukturen.

Generationen von Web-Anwendungen

1. Generation (Anf. 90er): statische HTML-Seiten.
Inhalte mit HTML formatiert und auf Web-Server gespeichert. Anfragen vom Browser an Server geleitet. Web-Server wählt passende HTML-Seite aus, navigiert über Links, schickt Seite an Browser zurück, der diese auflöst und anzeigt (Kommunikationsform: synchron, blockierend).
2. Generation (Mitte 90er): dynamische Web-Seiten (HTML-Format).
Mechanismus CGI (Common Gateway Interface): Browsereingaben (Formulare), am Server Bearbeitung in Abhängigkeit der Eingabe (CGI-Skripte, in verschiedenen Sprachen), Ergebnisrückgabe als HTML-Seite. Nachteil: schwergewichtige Prozesse ~> Engpass am Server.
Ergänzung durch *Clientseitige Interaktionen*

- dHTML (dynamic HTML).
- Skriptsprachen für Web-Browser, z.B. JavaScript, JScript: Browser-eigene Interpreter.
- Abarbeitung von Applets (vom Server) durch die JVM (Java Virtual Machine).

3. Generation (Ende 90er): serverseitige Web-Technologien.
Java Servlets, Java Server Pages (JSP), PHP, Active Server Pages (ASP), ASP.NET. Prinzip wie CGI: Aufrufe vom Browser am Server bearbeitet, Folgeseiten dynamisch in Abhängigkeit vom Ergebnis aufgebaut.

Vorteile:
- Jeder Aufruf startet einen Thread (leichtgewichtiger Prozess) ~> weniger Ressourcen.
- Anwendungen können alle Funktionalitäten der jeweiligen Plattform nutzen (Java, Windows-Technologien, .NET).

Vorteilhafter Einsatz von PHP (Personal Hypertext Pre-Processor): Scriptsprache im Bereich der Internet-Technologien.
Serverseitige Web-Technologien bilden Basis vieler Web-Anwendungen.

Bisherige Generationen: für Mensch-Maschine-Kommunikation entwickelt (Anwender interagieren über Browser mit der Anwendung am Web-Server).
Ergänzung durch *Clientseitige Interaktionen* (z.B. JavaScript).

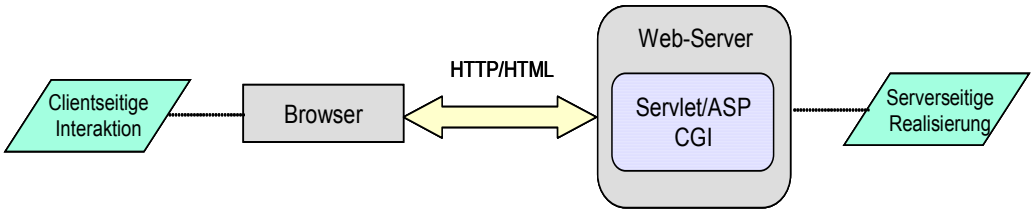


Abbildung 10.1: Mensch-Maschine Kommunikation im Web

Typische Anwendungen: Web 1.0 Applications, datenzentrierte Speicherungen / Retrievals (DBS), konventionelle Suchmaschinen (Google, Yahoo).

4. Generation (ab 2000er): Web-Services.
Erweiterung um Maschine-Maschine-Kommunikation. Web-Services bieten Anwendern komplexe Dienste an und suchen eigenständig nach geeigneten Teildiensten im Netz.

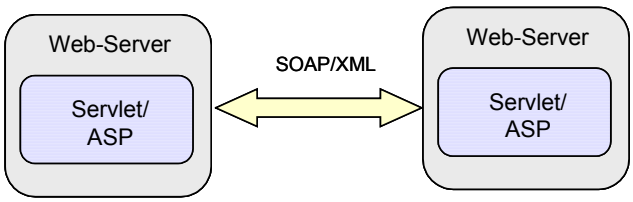


Abbildung 10.2: Maschine-Maschine-Kommunikation durch Web-Services

Beispielanwendung: Während Reisebuchung werden im Hintergrund weitere Dienste genutzt, wie Hotel- und Flugbuchung sowie Autoreservierung.

5. *Generation (Mitte 2000er):* asynchrone Web-Technologien.
Einsatz AJAX (Asynchronous JavaScript and XML): Seit Mitte der 2000er Erweiterung der synchronen Web-Abarbeitung (Browser bleibt blockiert bis zum Erhalt der HTML-Seite) durch asynchrone Kommunikationsform (nicht-blockierend). Nutzung JavaScript und XMLHttpRequest-Objekt: in gängigen Web-Browsern enthalten (steuert Kommunikation zw. Client und Server, stellt entsprechende send-Methoden und Callback-Funktionen bereit).
JavaScript, AJAX, XML und Web-Services bilden Basis der gegenwärtig modernen Web-Technologien: dynamisch, interaktiv, asynchron.
Beispiele:
- Web 2.0, u.a. Read-Write-Web (Wiki, Weblogs), Pull-Dienste (RSS-Feeds), Plattformen
  speziell: Social Network (YouTube, Flickr, Facebook, Google+, StudiVZ, Twitter, ...)
- neue Suchmaschinen (Google Search, Bing, Technorati),
- Online-Plattformen, u.a. Produktanbieter (Amazon, eBay), Paypal.

HTML5 ab 2009 als nächste Generation für Auszeichnungssprachen im Web festgelegt.

10.2 Standards zu Web-Services

Standards und Gremien

4 Standards und Standardisierungsgremien:

SOAP (ursprünglich „Simple Object Access Protocol“, ab Version 1.2 ohne Namen):
Zugriffsprotokoll für Web-Services, aufsetzend auf Transportprotokollen (HTTP, SMTP). SOAP nutzt deren Strukturen zur Übertragung von Datenpaketen in XML-Format. Verwaltung durch W3C (früher Microsoft). Aktuelle Version $_{2004}$: SOAP 1.2.
Vorgänger: ***XML-RPC***, Dave Winer, 1989 (Microsoft): Möglichkeit, um XML-Nachrichten über HTTP zu verschicken. Bildete Grundlage für SOAP. Anwendung auch bei Twitter (RoR, Web 2.0).

WSDL (Web-Services Description Language):
Schnittstellenbeschreibungssprache für Web-Services. Definiert Schnittstelle der Web-Services in XML und kann veröffentlicht werden. Verwaltung durch W3C.

XML (eXtended Markup Language): Deskriptive Sprache zur Beschreibung und Austausch komplexer Datenstrukturen. SOAP und WSDL verwenden XML zur Beschreibung von Schnittstellen, Datenstrukturen und Übertragungsformaten.
Verwaltung durch W3C. Aktuelle Version $_{\text{Febr. 2004}}$: XML 1.1.

UDDI (Universal Description, Discovery, and Integration): Verzeichnisdienst zur Veröffentlichung der Web-Services (Namensdienst für Web-Services).
UDDI definiert die Schnittstelle eines Verzeichnisdienstes für Web-Services. Entwicklung seit 2000. Verwaltung durch OASIS (Organization for the Advancement of Structured Information Standards). Aktuell: UDDI-Version 3, i.allg. noch Version 2 im Einsatz.
Vorgänger von UDDI: Spezifikation ***DISCO*** (DISCOvery, Microsoft) ~> WSDL-Beschreibung von Web-Services und zusätzliche Beschreibung in einer XML-Datei (DISCO-Datei) zusammengefasst. DISCO-Spezifikation beschreibt Struktur der DISCO-Datei, nicht aber deren Verwaltung in Registry oder Verzeichnisdienst.

Erweiterungen:

- Web Services Composite Application Framework (**WS-CAF**):
  Standard zur Koordination von Applikationen, z.B. für ein Transaktionsmanagement sowie Sicherheitsmanagement (auf Basis SAML: Security Assertion Markup Language).
  Spezifikation bei W3C und OASIS zur Standardisierung eingereicht.
- **Semantic Web Services**:
  Erweiterung von Web Services um Semantik, die das Auffinden (Discovery), Auswählen (Selection), Ausführen (Invocation) und Komposition mit anderen Web Services ermöglicht – analog der Idee des Semantic Web.
- **REST** (Representational State Transfer, Zustandsrepräsentationsübertragung):
  Beschränkung des Kommunikationsinterface (SOAP) auf eine Menge definierter Stan-dard-Operationen (an HTTP angelehnt: GET, PUT, POST, DELETE), insbes. Interaktion von zustandsbehafteten Ressourcen.

10.3 Grundkonzepte

10.3.1 Implementation

WS sind Anwendungskomponenten, die auf den Knoten eines web-basierten System liegen. Technologisch eher den entfernten Aufrufen zuzuorden: über das Protokoll SOAP (Simple Object Access Protocol, Basis XML) werden web-basierte entfernte Aufrufe definiert.

Realisierungen von Web-Services:

- mit.NET-Plattform.
- mit Java-Plattform: JAX-RPC (Java APIs for XML-based RPC) ist ein Standard zur Entwicklung Java-basierter Web-Services. Entwickelt unter Verantwortung der JCP (Java Community Process), ein Standardisierungsgremium unter Sun Microsystems.

- mit PHP-Plattform (PHP: Hypertext Pre-Processor, Skriptsprache; eine serverseitige Web-Technologie).
- SW-Tool xampplite: lokaler Apache-Server mit PHP, Web-Services (*freeservices*) u.a.

Unterstützung von Web-Anwendungen, u.a. bei Google, Amazon. Standardisierung W3C.
Dazu Anwendung auch sog. Lightweight Programming Models: Nicht die komplexen Web Services großer Firmen sondern eher Einsatz einfacher XML-Strukturen, z.B. RSS oder REST (Representational State Transfer). Damit Anwendungen einfach und effektiv mit vorhandenen Komponenten zusammengebaut.

Kommunikationsablauf:
Zugriff auf Web-Service erfolgt über SOAP (Modell der entfernten Prozeduraufrufe). Schnittstelle des Web-Services wird mit WSDL definiert und veröffentlicht. Zur Veröffentlichung wird Schnittstellenbeschreibung direkt an einen Client übergeben oder in einem UDDI-Verzeichnisdienst registriert. Nutzung XML zur Dokumentenbeschreibung.

10.3.2 XML (eXtended Markup Language)

Dokumentenstruktur
XML (eXtended Markup Language) als gemeinsame Sprache in Web-Services für Datenaustausch zwischen beliebigen Systemen. XML ist eine deskriptive Sprache zur Abbildung komplexer Datenstrukturen mit allen Abhängigkeiten in einem Dokument. Interpretation und Bearbeitung von XML-Dokumenten mit Open-Source-Werkzeugen.
XML-Dokument in 2 Bereiche aufgeteilt:
- Kopf: enthält allgemeine Metainformationen zu XML-Version und Zeichensatz,
- Rumpf: konkrete Daten.

Die Kopfinformationen benötigt der XML-Parser zur Bearbeitung der Rumpf-Daten.

Aufbau eines XML-Dokuments (Beispiel)
Kopf: XML-Version 1.0
Zeichensatz UTF-8 (Unicode Transformation Format-8)
Rumpf: Kundendaten einer Bank (Name, Vorname, Kontonummer)

```
<? xml version="1.0" encoding="UTF-8"?>
<Bank xmlns="http://www.beispielbank.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.beispielbank.com
Bank.xsd" >
<Kunde>
        <Name>Feuerstein</Name>
        <Vorname>Fred</Vorname>
        <Kontonummer>123456789</Kontonummer>
</Kunde>
        <Name>Geröllheimer</Name>
        <Vorname>Barney</Vorname>
        <Kontonummer>987654321</Kontonummer>
</Kunde>
...
</Bank>
```

Wohlgeformte XML-Dokumente:
XML-Dokument ist nach festen Regeln aufgebaut: hierarchisch, bestehend aus Elementen, mit einem Wurzelelement. Regeln für Elementstruktur (Auswahl):
- Elemente immer paarweise (öffnendes und schließendes Element)

```
<Kunde>....</Kunde>
```

- Verschachtelungen erlaubt, aberElemente nicht überlappend (öffnendes und schließendes Element nicht überschneidend): `<Kunde><Name>....</Name></Kunde>`
- 2 Alternativen zur Darstellung paarweiser Elemente:
  Leere Elemente: `<Kunde></Kunde>` zusammengezogen zu `<Kunde/>`.
  Ebenfalls Verwendung einer inline-Schreibweise. Untergeordnete Elemente werden als Attribute im Element definiert: `<Konto nummer="987654321"/>`

Namensräume: um XML-Elementen global eindeutige Namen zu geben.
Beispiel: Namensraum `http://www.beispielbank.com`. Alle Elemente des XML-Dokuments haben implizit den Namensraum als Präfix. Damit Vermeidung von Konflikten.

Gültige XML-Dokumente:
Wohlgeformtheit eines XML-Dokuments sichert noch nicht seine Gültigkeit. Es muss auch einer Strukturbeschreibung genügen. Zur Strukturfestlegung 2 Techniken verfügbar:
- DTD (Data Type Definition),
- XML-Schema.

Beide ähnlich: Festlegung der Informationen je Element, z.B. Name und Wertebereich. Bei XML-Schema zusätzlich auch Festlegung der Datentypen möglich. Beide Techniken alternativ einsetzbar, z.B. HTML basiert auf DTD-Beschreibung, neueres SOAP-Protokoll basiert auf flexibleren XML-Schema. W3C stellt ein Standard-XML-Schema mit allen Basisdatentypen zur Verfügung, u.a. *string, integer, element, complexType, type, sequence.* Es bildet die Grundlage aller XML-Dokumente.

10.3.3 WSDL-Schnittstellendefinition

Schnittstellendefinition eines Web-Services erfolgt als *XML-Dokument*. Gültigkeit des Dokuments aber nicht mehr durch W3C-XML-Schema vorgegeben, sondern durch ein *Schema des WSDL-Standards*.
WSDL-Schnittstellendefinition: nach festgelegter Struktur
<definitions> ... </definitions>
Wurzelsegment eines WSDL-Dokuments, definiert Namen und Namensraum des Services sowie Namensraum der verwendeten Standards. Klammert alle folgenden Definitionen.
<types> ... </types>
Enthält alle Datentypdefinitionen, die für Aufruf des Services nötig sind und nicht im Standard XML-Schema des W3C definiert sind.
<message name="Nachricht1"> ... </message>
Definiert die Nachrichten, die bei einem SOAP-Aufruf übertragen werden. Falls mehrere Nachrichten (z.B. Eingabeparameter, Rückgabewerte), sind mehrere Nachrichten zu definieren. Nachricht kann aus logischen Teilelementen (sog. Parts) bestehen. Parts definieren Name-Wert-Paare zu den Parametern einer Nachricht.
<portType> ... </portType>
Beschreibt die vom Web-Service angebotenen Methoden. Zur Definition der Parameter werden die oben definierten Nachrichten verwendet. WSDL unterstützt 4 Aufruf-Kommunikationstypen:
- **One-way**: Client sendet eine Nachricht an Webserver, Antwort wird nicht erwartet.
- **Request-Response**: Client sendet Nachricht und erhält vom Web-Server eine Antwort.
- **Solicit Response**: Server sendet eine Nachricht und erhält vom Client eine Antwort.
- **Notification**: Server sendet Nachricht an Client, Antwort wird nicht erwartet.

<binding> ... </binding>
Definiert Nachrichtenformate und Transportprotokoll, die zur Übertragung der Aufrufe verwendet werden. Häufig wird SOAP-Binding eingesetzt.
<service> ... </service>
Definiert alle für Zugriff auf den Dienst notwendigen Informationen, u.a. Netzwerkadresse,

Portnummer.
WSDL-Spezifikation i.allg. nicht von Hand. WS-Plattformen bieten Tools, die aus der Schnittstellendefinition in einer spezifischen Programmiersprache eine WSDL-Schnittstellendefinition generieren.

10.3.4 Verzeichnisdienst UDDI

Veröffentlichung mit UDDI
Zur Nutzung eines Web-Services ist seine Schnittstelle zu veröffentlichen, i.w. im UDDI-Verzeichnisdienst. Im Standard 2 API definiert:
- Publishing-API: Anbieter können damit ihren Web-Service in einem UDDI-Verzeichnisdienst veröffentlichen.
- Inquiry-API: Servicenutzer können damit inhaltlich nach Web-Services suchen.

Standard schreibt Internet-weite Veröffentlichung und Nutzung vor ~> erfordert übergreifende Organisation des UDDI-Verzeichnisdienstes. Nicht jeder UDDI-VD muss universell und global sein. UDDI-Standard unterscheidet 2 Typen von Dienstanbietern: UDDI-Operatoren und private UDDI-Verzeichnisdienste.
UDDI-Operatoren sind Betreiber von UDDI-Verzeichnisdiensten für bestimmte Anforderungen: Garantiezusicherung für Verfügbarkeit, Sicherheit und Performance. Nur wenige Firmen können dies erfüllen. Aktuelle Operatoren: Microsoft, IBM, SAP. Private UDDI-Verzeichnisdienste bisher weitgehend unabhängig.
UDDI-Verzeichnisse der UDDI-Operatoren arbeiten eng zusammen und werden in ihrer Gesamtheit als Universal Business Registry (UBR) bezeichnet. UDDI Version 2 ermöglicht noch keine Interoperabilität zwischen UBR und privaten UDDI-Verzeichnissen ~> dazu Version 3.
Neben den Betreibern von UDDI-Verzeichnissen sieht Standard zusätzliche Registrare vor. Es sind Firmen, die Schnittstellen zur Registrierung von Services an UBR anbieten.
UDDI-Schema: Definiert Struktur und Funktionalität eines UDDI-Verzeichnisdienstes. Das Schema definiert das Modell aller Entitäten, die zur Modellierung von Informationen zu Web-Services zur Verfügung stehen. Folgende Entitäten im UDDI-Schema unterstützt:
- **businessEntity**: Repräsentiert ein Unternehmen oder Geschäftseinheit, das Web-Services anbietet.
- **businessService**: Beschreibt eine Menge zusammengehöriger Web-Services, die von einer businessEntity angeboten werden.
- **bindingTemplate**: Beschreibt technische Informationen für Zugriff auf Web-Service.
- **tModel (taxonomy model)**: Dient zur Kategorisierung von businessEntity und Service. tModels definieren eine Taxonomie im Verzeichnisdienst, innerhalb der er business-Entity und businessService abgelegt und gesucht werden können.

Um Web-Service an einem UDDI-Verzeichnisdienst anzumelden, wird eine businessEntity erstellt. Dieser Entity wird die WSDL-Schnittstellenbeschreibung des Web-Services zugeordnet. Damit ist Web-Service veröffentlicht und kann von Nutzern abgefragt werden.

10.3.5 Kommunikationsablauf

Kommunikation über SOAP
Hat Client die Schnittstellendefinition des Web-Services vom UDDI-Dienst importiert, kann Kommunikation mit dem Service beginnen. Aus WSDL-Beschreibung wird ein Stub generiert, über den der Client seine Nachrichten an den Web-Service sendet. Grundlage der Kommunikation i.allg. SOAP (Simple Object Access Protocol). SOAP gibt aber lediglich Struktur der Nachrichten vor, die dann über Kommunikationsprotokolle (i.d.R. HTTP, aber auch SMTP oder FTP) verschickt werden. SOAP definiert die Struktur der Nachrichten. Eine SOAP-Nachricht ist ein XML-Dokument, in dem der Aufruf an den Web-Service mit den benötigten Parametern definiert ist. Falls Antwortnachricht, dann auch Ergebnisdaten enthal-

ten. Wie bei WSDL und UDDI gibt es auch für SOAP ein XML-Schema, das Struktur und Bedeutung der SOAP-Elemente festlegt. Definition von Datentypen: Standard-XML-Schema.

Aufbau einer SOAP-Nachricht
SOAP-Body enthält die zu versendenden Daten. SOAP-Header erlaubt zusätzliche Informationen mitzuschicken, u.a. Sicherheitsinformationen oder Transaktionskontext. SOAP-Envelope bildet die Klammer um Body und Header. Abhängig vom jeweiligen Kommunikationsprotokoll wird der SOAP-Envelope in einer spezifischen Nachricht des Protokolls verpackt und verschickt.

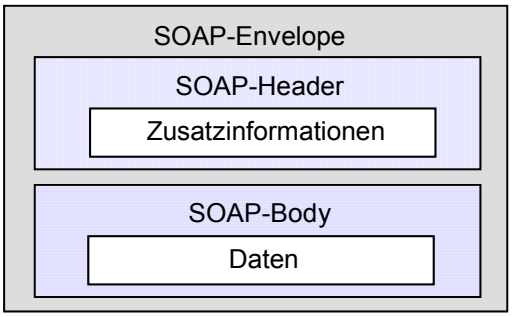


Abbildung 10.3: Aufbau SOAP-Nachricht

10.4 Sicherheit, Prozess- und Transaktionsverwaltung

Sicherheit
Anfangs geringes Sicherheitskonzept, insbes. fehlender Standard ~> Entwicklung von Web Services Security.
WS-Security: Bekanntester Vorschlag eines Sicherheitsstandards für Web-Services. Setzt auf SOAP-Mechanismus; dafür Erweiterungsraum im SOAP-Header genutzt. Grundelement von WS-Security sind Token - in XML formulierte Sicherheitsinformation.
Bisher 2 Token im Standard:
- UsernameToken: zur Übermittlung von Authentifizierungsinformationen, wie Benutzername und Passwort.
- BinarySecurityToken: zur Übertragung von Binärdaten, z.B. verschlüsselte Zertifikate.

Zur Verschlüsselung und Signierung von Daten werden 2 weitere Standards genutzt:
- XML-Signature: Standard für digitale Signaturen in XML, z.B. für Signierung von XML-Nachrichten, um Identität des Absenders zu sichern.
- XML-Encryption: Zur Verschlüsselung der Nachricht (allerdings nicht der Algorithmus, sondern Vorgabe der Struktur einer verschlüsselten XML-Nachricht).

SOAP-Header-Erweiterung und beide Token für WS-Security im Standard zwar veröffentlicht, aber praktisch noch wenig umgesetzt, Tendenz zunehmend. Nutzung auch anderer Sicherheitsmechanismen, z.B. **SSL** (Secure Socket Layer). Standard Version 1.0 (2002), an OASIS zur Weiterentwicklung übergeben.
Zur Verwaltung und Prüfung von Zugriffsrechten kann weiterer Standard eingesetzt werden: **SAML** (Secure Assertion Markup Language). SAML arbeitet mit Zusicherungen (Assertions): Datenstruktur, die Informationen eines beliebigen Objekts repräsentiert. Struktur der Zusicherung in XML-Schema definiert.
- Mittels Zusicherungen und weiterer Verwaltungseinheiten wird in der SAML-Architektur eine virtuelle Vergabestelle für Rechte nachgebildet.
- Daten anhand von vorgegebenen Regeln geprüft.
- Anwendung zur Authentifizierung und Autorisierung.

SAML ermöglich auch sog. **Single-Sign-On**: Anwender meldet sich einmal über eine Zugangsseite an und hat von da an Zugriff auf verschiedene Systeme und Anwendungen.

Prozess- und Transaktionsverwaltung
Ziel von Web-Services: Realisierung komplexer Geschäftsmodelle im Internet ~> erfordert Zusammenarbeit mehrerer Web-Services (sog. "Choreographie"). Basisstandards SOAP und WSDL für komplexe Anwendungen nicht mehr ausreichend ~> erforderlich: Service-übergreifende Prozess- und Transaktionsverwaltung.
Erarbeitung von Standard-Vorschlägen ~> IBM und Microsoft, u.a.
- Prozessbeschreibungsprache BPEL (Business Process Execution Language).
- Transaktionsstandard BTP (Business Transaction Protocol).

Häufig noch proprietäre Prozessbeschreibungssprachen in den Workflow Engines.

BPEL** und **WS-T
BPEL (Business Process Execution Language): Prozessbeschreibungssprache für Web-Services. Damit können konkrete Interaktionen zwischen Web-Services oder die Reihenfolge von Interaktionen komplexer Abläufe beschrieben werden. Beschreibung in XML, Prozessbeschreibung in WSDL 1.1 und Standard-XML-Schema. Unterstützung durch IBM und Microsoft, Standardisierung durch OASIS. Standard-Vorschlag WS-T:
Erweiterung BPEL um Transaktionsverwaltung mit 2 Transaktionstypen:
- Atomic Transaction (AT): für kurzläufige Transaktionen: mit ACID-Eigenschaften und 2-Phasen-Commit, allerdings sehr restriktiv.
- Business Acticities (BA): für lange Transaktionen, besser anpassbar. Verwaltung durch IBM, Microsoft, BEA; Standardisierung durch W3C.

BTP
BTP (Business Transaction Protocol): Transaktionsstandard (OASIS). Kern bilden sog. BTP-Elemente für jede Anwendungskomponente zur Koordination der Transaktion.
2 Transaktionsmodelle:
- Atomic Business Transactions: entspricht Atomic Transaction des WS-T: ACID-Prinzip.
- Cohesive Business Transactions: geschachtelte Transaktionen.

Flexible Anpassung an die jeweiligen Abläufe möglich.

10.5 Web-Services mit Java

JAX-RPC (Java APIs for XML based RPC)
Web-Service-Lösung der Java-Plattform - ein Standard zur Entwicklung Java-basierter Web-Services. Entwickelt unter JCP (Java Community Process). Erweitert den Web-Service um eine Java-spezifische RPC-Schnittstelle. Wesentliche Erweiterungen sind Abbildungsregeln zwischen Java und den WSDL-Datentypen, die auf dem Standard-XML-Schema basieren.
Auf Serverseite definiert JAX-RPC ein Programmiermodell für die Realisierung des Web-Services. Dabei wird die Schnittstelle des Web-Services (sog. Service Endpoint Interface) mit Java definiert. Aus der Schnittstelle wird dann die WSDL-Schnittstelle generiert und der Web-Service kann veröffentlicht werden.

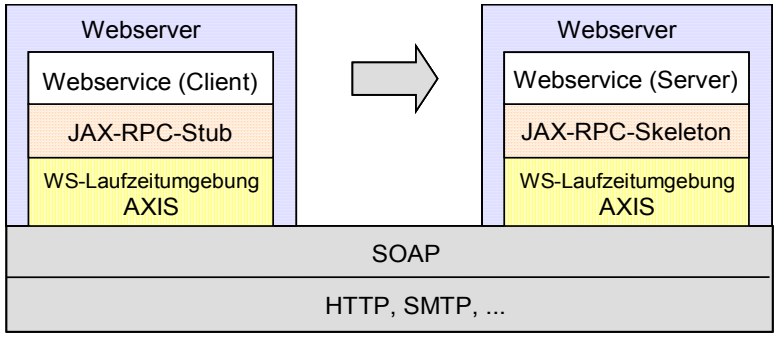


Abbildung 10.4: Web-Service-Plattform, mit JAX-RPC und AXIS

WSDL-Schnittstelle dient zusätzlich als Grundlage zur Generierung von Stub und Skeleton. Implementierung des Web-Services bleibt frei von technischen Aspekten (so kann bspw. jede Komponente zu einem Web-Service erweitert werden).

JAX-RPC-Standard definiert nur die Programmierschnittstelle für Java-Anwendungen. Zusätzlich ist eine Web-Service-Plattform wie bspw. Apache AXIS notwendig, um Laufzeitaspekte abzudecken (AXIS: SOAP-Implementierung im Rahmen des Apache-Projekts).

JavaScript
Skriptsprache in Web-Browsern; ermöglicht clientseitige Interaktionen zwischen Browser und Nutzer. Optionaler Interpreter in allen gängigen Browsern. Zusätzlich existieren auch zahlreiche eigenständige Implementierungen, z.B. für Mono/.NET und Java ~> erlauben Skripting innerhalb von eigenständigen Anwendungen.

AJAX: Asynchronous JavaScript and XML
Konzept zur asynchronen Kommunikation zwischen Web-Client und Web-Server. Damit besteht die Möglichkeit, eine Webseite zu aktualisieren, ohne diese erneut vollständig abzurufen. Voraussetzungen für diese Web-Anwendungen sind:
- einerseits JavaScript und
- andererseits die Möglichkeit, aus dieser Skriptsprache heraus mit dem Web-Server zu kommunizieren.

Letzteres ist über das `XMLHttpRequest`-Objekt in allen gängigen Web-Browsern möglich. Alternativ bietet die bei Mozilla.org entwickelte Gecko-Engine die Möglichkeit, per SOAP mit dem Web-Server zu kommunizieren.

10.6 Web-Services mit PHP

PHP: Personal Hypertext Pre-Processor
Weit verbreitete Scriptsprache im Bereich der Internettechnologien. Es ist eine serverseitige Technologie, entwickelt für die Erstellung dynamischer HTML-Seiten.
Ähnlich C/C++, angelehnt an Java, C und Perl, aber einfacher zu programmieren und mit hoher Funktionalität. PHP erlaubt die prozedurale und objektorientierte Nutzung (~> leichter Einstieg für Programmierer von C und C++).
PHP ist eine interpretierte Scriptsprache, die sowohl an der Kommandozeile als auch als Modul eines Webservers läuft ~> dadurch einfaches Debuggen möglich.
Online-Quellen:
- Professionelle Software-Entwicklung mit PHP 5 – Objektorientierung. Entwurfsmuster. Modellierung. Fortgeschrittene Datenbankprogrammierung: Sebastian Bergmann, 1. Auflage, 2005.
- PHP 4 – Webserver-Programmierung für Einsteiger. Thomas Theis, 1. Auflage, 2005.
- PHP Handbuch.

PHP-Programmierung
PHP-Programme werden direkt in die HTML-Seite durch das Tag <?php und ?> eingebettet. Dadurch keine separaten Script-Dateien erforderlich. HTML-Dateien, die PHP-Code enthalten, werden nicht direkt an den anfordernden Client übertragen, sondern erst durch einen sog. PHP-Interpreter gefiltert. Dieser gibt das HTML unverändert an den Client weiter, führt aber zuvor die PHP-Programmzeilen aus.
Mit PHP somit dynamische Dokumente erzeugbar, ohne den HTML-Code verändern zu müssen, z.B. Einfügen der aktuellen Uhrzeit, Datenbankeinträge usw.

PHP und SOAP
PHP Version 5.x verfügt über eine leistungsfähige SOAP-Implementierung. Im Verzeichnis freeservices werden Beispieldienste per SOAP angeboten, z.B. String-Reverse, Echo, ...
Die Beschreibung der Dienstleistung erfolgt über WSDL.
URL der Dienstleistung:

```
http://<webserver>.<ip-adresse>/freeservices
```

URL der Dienstbeschreibung in WSDL:

```
http://<webserver>.<ip-adresse>/freeservices/?wsdl
http://<webserver>.<ip-adresse>/freeservices/freeservices.wsdl
```

Praktikum Web-Technologien („Einfacher Taschenrechner“)
Es ist eine prozedurale (oder eine objektorientierte) Implementierung eines einfachen Taschenrechners für die 4 Grundrechenarten mit jeweils 2 Operanden zu erstellen. Ergänzung der Implementation durch eine Konsolenanwendung sowie eine formular-gesteuerte Webanwendung, die die Funktionen (bzw. Klasse) nutzt.
Voraussetzungen: Webserver, ausführbares Home-Verzeichnis, Verzeichnis public_html im eigenen Home-Verzeichnis.
Web-Praktikum BA Gera: Aufgabenstellung und Musterlösung siehe Skript WebPraktikum_IntW3.
Verwendung des SW-Entwicklungswerkzeuges **xampplite**:
- es stellt neben dem Web-Server (Apache-Server) auch eine PHP-Implementation mit Web-Services bereit (Verzeichnis `xampplite/hotdocs/freeservices`).
- Abruf der Web-Services über SOAP.
- Installation an BA Gera, ET-L: `http://192.168.41.10/pi0x/pki/freeservices`

10.7 Einsatz von Web-Services

Anwendungsgebiete und Einschätzung
Einsatzgebiete: Geschäftsmodelle im Internet, Bereitstellung kombinierter Dienstleistungen im Web, Anwendungsintegration (~> Basis für SOA). Bisheriges Problem: ungenügend ausgereifte Sicherheits- und Transaktionsstandards.
Vorteile der Web-Services gegenüber konkurrierenden Middleware-Technologien:
- Web-Service für Internet-Anwendungen prädestiniert: Infolge Verwendung HTTP als Kommunikationsprotokoll nicht auf Intranet beschränkt, sondern über Firewall-Grenzen hinaus erreichbar.
- Einfach zu implementieren (jede Web-Service-Plattform bietet ausreichend Tools an).
- Offener Standard, d.h. interoperabel (zumindet in Zukunft).
- Kostengünstig und einfach zu realisieren. Es wird lediglich eine geeignete Web-Infrastruktur (Web-Server, Netzwerk, Internetprotokolle) benötigt.

Nachteile:
- XML-Formatierung der Nachrichten vergrößert Daten um 30% ~> Performance-Engpass
- z.T. Client und Server mit verschiedenen XML-Schema bzw. Schema-Versionen.

11 Microsoft-Componentware .NET

11.1 .NET Plattform (Entwicklung, Konzepte, Werkzeuge)

Microsoft .NET

.NET: Middleware-Plattform für verteilte Anwendungen in Microsoft-Umgebung. Entwicklung durch Microsoft (Redmond), nur z.T. offene Standards.
Strategiewechsel von Microsoft: produktorientierte Desktopwelt ~> dienstorientierte Komponentenwelt. Öffnung zu anderen Technologien, z.B. Unterstützung der offenen Webservice-Standards (HTTP, SOAP, WSDL); direkte Interoperabilität.
.NET-Plattform: Sammlung historisch gewachsener Technologien, ergänzt durch .NET Framework mit Laufzeitumgebung (Runtime) und Klassenbibliothek (objektorientiert).
Unterstützte Betriebssysteme: MS Windows, LINUX (Open-Source-Projekt Mono, Novell).

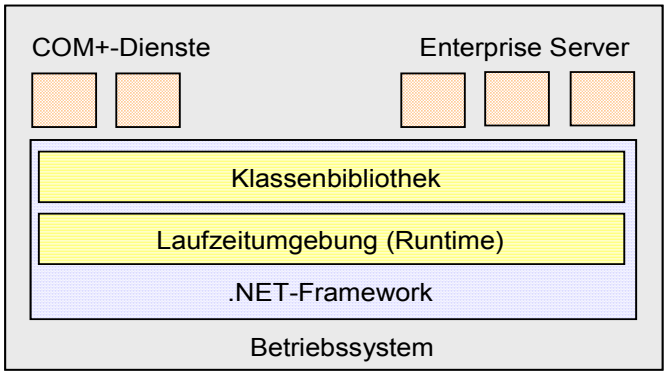


Abbildung 11.1: .NET-Plattform

.NET Framework, Dienste und Server setzen auf Betriebssystem (Windows, Linux) auf und bilden zusammen die .NET-Plattform. Kern der Plattform ist das Framework: (objektorientierte) Laufzeitumgebung, Klassenbibliothek (BCL) und diverse Hilfsprogramme (u.a. für Sicherheit, Rechteverwaltung). Dabei Nutzung der Dienste der Enterprise Server, der COM-Technologien und des COM+ -Transaktionsdienstes.
.NET Gegenstück zu J2EE und CORBA.

Entwicklung

Ursprung: Unterstützung von Serverplattformen durch einen speziell für Windows angepassten Transaktionsdienst MTS (Microsoft Transaction Server): bildet zusammen mit dem erweiterten COM-Komponentenmodell DCOM eine Umgebung für serverseitige Komponenten (COM: Component Object Model, DCOM: Distributed Component Object Model).
MTS und DCOM später in der COM+ -Technologie zusammengefasst. COM+ und .NET sind unabhängige Technologien, wobei nach Angaben von Microsoft COM+ auf lange Sicht nicht weiter unterstützt wird. .NET nutzt insbes. den Transaktionsdienst von COM+.
Ende 90er: Entwicklung eine umfassenden Serverplattform DNA (Distributed iNternet Architecture) bzw. NGWS (Next Generation Windows Services); nicht durchgesetzt ~> später zur .NET -Plattform ausgebaut (Ende 2000: Betaversion von .NET 1.0). Mit .NET werden zuvor entwickelte SW-Entwicklungskonzepte (z.B. COM) abgelöst.
NET ist eine SW-Plattform von Microsoft zur Entwicklung und Ausführung von Programmen. Große Teile davon, insbes. CLR und BCL, als ECMA-Standard unter der Bezeichnung Common Language Infrastructure (CLI) normiert. NET in vollem Umfang nur für Windows verfügbar. Durch Standardisierung der Laufzeitumgebung besteht Interoperabilität für SW aus .NET bzw. für .NET (Wrapper). Somit laufen viele Programme, die unter .NET entwickelt wurden, mit Hilfe des Monoprojektes auch auf Unix-basierten Betriebssystemen .

Versionen:

.NET Framework 1.0	05.01.2002	Windows 98, NT, XP
.NET Framework 1.1	01.04.2003	Windows Server 2003 (incl. IPv6)
.NET Framework 2.0	07.11.2005	Windows 2000, Windows 98, Windows ME
.NET Framework 3.0	06.11.2006	Windows XP SP 2, Windows Vista (incl. WCF)
.NET Framework 3.5	09.11.2007	Windows Vista (incl. AJAX), Quellcode z.T. ab 17.01.08
.NET Framework 3.5 SP1	11.08.2008	Windows Vista
.NET Framework 4.0	12.04.2010	Windows Vista, Windows 7 (Microsoft Visual Studio 2010)
.NET Framework 4.5	[14.09.2011	Vista, Windows 7, 8 (Microsoft Visual Studio 2011), angek.]

Konzept von .NET

NET ist die Umsetzung des ECMA-Standards Common Language Infrastructure (CLI). Entwicklung und Ausführung von Programmen, die mit unterschiedlichen Programmiersprachen auf verschiedenen Plattformen erstellt wurden.

.NET von Anfang an für die Programmierung mit unterschiedlichen PS entwickelt. Statt direktkompilierende Sprachen (insbes. C, C++) nun Zwischensprache (Common Intermediate Language, CIL). Mit dieser Technologie will Microsoft der Marktmacht von Java bzw. dem Erfolg der Scriptsprachen begegnen ~> mehr Effizienz in Programmentwicklung.

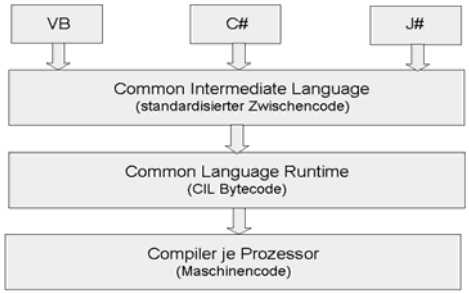


Abbildung 11.2: Basisprinzip von .NET

.NET-Programme bestehen nicht aus direkten Prozessorbefehlen, sondern aus dem Zwischencode CIL (Common Intermediate Language). Die CLR (Common Language Runtime) stellt den Interpreter für den standardisierten Zwischencode CIL dar (ursprünglich MSIL: Microsoft Intermediate Language, umbenannt durch ECMA International). Das .NET-Framework enthält jeweils auch einen Compiler, der CIL-Code für den jeweiligen Prozessor übersetzen und ausführen kann. Für die CIL wurde ein sprachübergreifendes System von objektorientierten Datentypen definiert, so dass alle Hochsprachen, die sich an den CIL-Standard halten, gültigen CIL-Bytecode erstellen können.

Compiler für .NET-Sprachen erzeugt keinen Maschinencode, der direkt vom Betriebssytem ausgeführt werden kann. Es wird ein prozessorunspezifischer Zwischencode erzeugt, sog. IL-Code (Intermediate Language Code): besteht aus Befehlen, die auf der stackbasierten virtuellen Maschine (VM) ausgeführt werden können.

Die resultierenden Programme haben wie die üblichen Programme unter Windows eine „.exe“-Erweiterung. Ermöglicht wird das durch eine kleine Routine am Programmanfang, die die virtuelle Maschine startet, die wiederum den Zwischencode ausführt.

Zur Programmausführungszeit übersetzt ein JIT-Compiler (Bestandteil der Laufzeitumgebung Common Language Runtime, CLR) den Zwischencode in Maschinencode, der dann vom Prozessor direkt ausgeführt wird.

Werkzeuge

.NET in verschiedenen Formen angeboten:

- als Framework (reine Laufzeitumgebung inkl. aller Klassenbibliotheken),
- als kostenloses SDK (Software Development Kit) für Entwickler,
- *Microsoft Visual Studios .NET*: kostenpflichtige integrierte Entwicklungsumgebung (IDE).

Für Einsteiger und Studenten (kostenfrei)
- Microsoft Visual Studio Express Editions (Einschränkungen gegenüber Standard-Variante),
- IDE im Open-Source-Projekt SharpDevelop bzw. im DreamPark-Programm.

In Betriebssystemen
- seit Windows Server 2003 in Server-Betriebssystemen,
- .NET ab Windows 98 einsetzbar, in Windows Vista Kernbestandteil des Systems (analog Win7, 8),
- mit eingeschränktem Funktionsumfang auch in Linux (Mono-Projekt),
- für Mobiltelefone unter Windows CE bzw. Windows Mobile: .NET Compact Framework (zusammen mit dem kostenpflichtigen Visual Studio .NET 2003 ff),
- für embedded Systems: .NET Micro Framework (seit Sept. 2006) – eigenständig bzw. mit Windows.

11.2 .NET Framework

Architektur .NET Framework

.NET Framework: bildet den Kern der .NET-Technologie, bestehend aus
- einer Laufzeitumgebung (Common Language Runtime, CLR) und
- einer objektorientierten Klassenbibliothek (Base Class Library, BCL), die alle Bereiche der Windows- und Web-Programmierung abdeckt.

In BCL auch ehemalige Subsysteme ASP, ADO sowie Web-Services. Ergänzung durch neue Programmiersprache C# ("C sharp"), die auf .NET abgestimmt ist.

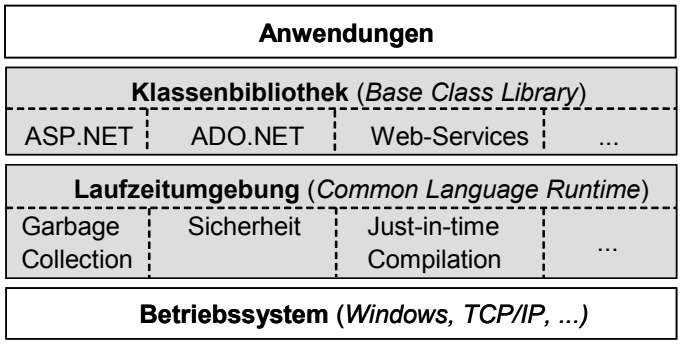


Abbildung 11.3: Architektur des .NET Framework

Common Language Runtime (CLR)

Laufzeitumgebung für Anwendungen in beliebiger Sprache. Unterstützung von
- Microsoft-Sprachen C#, C++/CLI (Managed C++), Visual C++, Visual Basic :NET, Microsoft-Java (J#) und Jscript .NET;,
- Smalltalk, Cobol, Tcl/Tk, Perl, Python, Delphi.

Sprachen auf eine gemeinsame Zwischensprache abgebildet: *Common Intermediate Language* (CIL), ursprünglich Microsoft Intermediate Language (MSIL). CLR bildet dabei die Laufzeitumgebung und den Interpreter für den CIL-Code. CIL ist keine Hochsprache und keine Maschinensprache. CIL ist binär, objektorientiert und kennt Datentypen ~> als Zwischensprache bezeichnet. Gemeinsames Typsystem (*Common Type System*) sichert einheitliche Interpretation der Datenstrukturen auf MSIL-Ebene (vgl. CORBA: Common Data Representation). Weitere Aufgaben der CLR:
- *Effiziente Speicherverwaltung*: geschützte Speicherbereiche, Freigabe von Speicherbereichen (Garbage Collector).
- *Threadverwaltung:* Nebenläufigkeit, leichtgewichtige Prozesse
- *Sichere Ablaufumgebung:* analog Java ~> vertrauenswürdiger Code (immer ausführbar) / nicht vertrauenswürdiger Code (für geschützte Speicherbereiche).
- Einheitliches Modell zur Ausnahmebehandlung (*Exceptions*) für alle Sprachen.
- Verwaltung des Zugriffs auf *Betriebssystemressourcen*.
- Durch MSIL einheitlicher *Zugriff auf Klassenbibliothek*.
- *Gemeinsames Typsystem* unterstützt die Interoperabilität zwischen den Sprachen.

managed / unmanaged Code
Compiler wandelt Hochsprachen in CIL-Programme, sog. Managed Code (= alle Programme, die unter Verwaltung der CLR ablaufen). Anwendungen in C++ können als Unmanaged Code laufen; hierbei stellt das Betriebssystem die Laufzeitumgebung.
CIL ist kein lauffähiger Code, muss vor Ausführung in Maschinencode übersetzt werden:
- *Just in Time* (JIT): CIL-Code zur Laufzeit übersetzt, JIT-Kompilation auf Ebene der einzelnen Methoden.
- zur *Installationszeit*: CIL-Code vollständig zur Installationszeit in Maschinencode übersetzt.

Nur wenige Sprachen, die sowohl *managed* als auch *unmanaged* Code in einer einzigen Programmdatei unterstützen: C++/CLI (sog. Managed C++) und Delphi (ab Version 8).

Interoperabilität und Sprachneutralität
Mit Hilfe der sog. *Interop-Technik* lassen sich traditionelle (Microsoft) COM-Programme mit .NET-Kapseln (sog. Wrappern) versehen und somit deren Programmfunktionen wie „normale“ .NET-Programmfunktionen aufrufen. Umgekehrt lassen sich auch .NET-Funktionen wie COM-Funktionen aufrufen.
Common Language Specification (CLS): sie definiert als eine gemeinsame Untermenge den Bytecode der CIL, der von der virtuellen Laufzeitumgebung (VM) in den Maschinencode der Zielmaschine übersetzt und ausgeführt werden kann. Interoperabilität wird durch das Common Type System (CTS) gesichert.
.NET ermöglicht Programmieren mit verschiedenen an die CLR angepassten Sprachen. Im Visual Studio sind mitgeliefert: C# , C++/CLI, das proprietäre Visual Basic .NET, J# (Microsoft Java), JScript .NET und ab 2010 die funktionale Programmiersprache F#.
Hinzu traditionell C++, Basic (über CTS) sowie Sprachen von Drittanbietern, z.B. Delphi.

Common Type System (CTS)
CTS für alle .NET-Sprachen verbindliches Typsystem ~> sichert Interoperabilität zwischen den Sprachen. Einschränkung des Typsystems durch die Common Language Specification (CLS) auf vom CTS unterstützte Sprachen. CLS: Regelwerk zur Abbildung der CLR-Sprachen auf die IL (Intermediate Language).
Vorteile durch ausschließliche Verwendung von Typen des eingeschränkten Typsystems:
- *Interoperabilität zur Compilezeit*: jede Klasse auf die IL abgebildet. Interoperabilität somit auf IL-Ebene (u.a. Vererbung).
- *Interoperabilität zur Laufzeit* (über Rechnergrenzen): Voraussetzung ist jeweils eine CLR als Laufzeitumgebung.

CTS sichert Nutzung der Vorteile von Java bzw. CORBA, u.a. in Remoting-Technologie.

Klassenbibliothek (Framework Base Class Library, BCL)
BCL fasst alte und neue Windows-Techniken zusammen. Steht allen Anwendungen der CLR als objektorientierte Bibliothek bereit.
Namensräume
Namensraum enthält logisch und funktional zusammengehörige Klassen. Namensräume hierarchisch in Bäumen organisiert. .NET-Klassenbibliothek umfasst 2 Baumhierarchien, Wurzelelemente sind die Namensräume *System* und *Microsoft*.

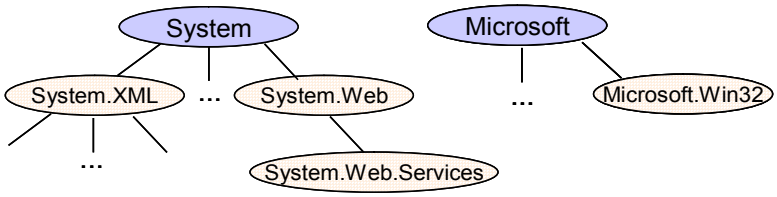


Abbildung 11.4: Namensraumhierarchie der Klassenbibliothek

System-Baum:
Für Dienste, die von Microsoft-Umgebung unabhängig sind.
- Namensraum *System*: Klassen mit Basisfunktionalität zur Ereignis- und Ausnahmebehandlung sowie einfache und komplexe Datentypen.
- Namensraum *System.Web*: Webtechnologien.
- Namensraum *System.XML*: Funktionalität zur Bearbeitung von XML-Dokumenten.

Microsoft-Baum:
Für Microsoft-spezifische Dienste, u.a. Zugriff auf Windows-Registry. Trennung der Bäume wegen Portierbarkeit des System-Bereiches auf andere Plattformen.

11.3 Microsoft Komponentenmodelle

11.3.1 Komponentenbasierte Middleware-Plattform

Komponentenmodell
Konzeptionelle und technische Basis zur komponentenbasierten Entwicklung verteilter Anwendungen (Erweiterung des objektorientierten Paradigmas). Dient der Unterstützung effizienter Softwareentwicklung verteilter Systeme (wiederverwendbar, time-to-market).
Componentware: Anwendungsobjekte werden als Komponenten definiert (Schnittstellen) und für eine Komponenten-Laufzeitumgebung bereitgestellt.
Bekannteste Entwicklungsumgebungen (mit Komponentenmodellen):
- Java 2 Platform Enterprise Edition (J2EE), Komp.-modell EJB (Enterprise JavaBeans).
- .NET-Plattform mit Komponentenmodell COM bzw. DCOM und WCF.
- CORBA 3.0 mit Komponentenmodell CCM.

(im Detail siehe Kap. 2.3.3.).

11.3.2 Component Object Model (COM)

Entwicklung Komponentenmodell
OLE (Object Linking and Embedding): Erste Technologie zur Bearbeitung von Verbunddokumenten. OLE setzte auf dem komplexen und wenig effizienten Mechanismus DDE (Dynamic Data Exchange) auf.
COM (Component Object Model): 1993 als Interaktionsmechanismus für lokale Softwarekomponenten eingeführt; löste DDE als Basistechnologie von OLE ab. COM ursprünglich zur Unterstützung des dokumentenzentrierten Arbeitens entwickelt: Dokumente unterschiedlichen Typs werden in einem Gesamtdokument zu einem Verbunddokument zusammengestellt. COM nicht nur für Dokumentenbereich geeignet. Heute zentrale Rolle in Windows-Betriebssystemen und vielen Technologien wie ActiveX oder ODBC, ebenfalls in .NET-Plattform integriert. COM nur zur lokalen Nutzung entwickelt, nicht geeignet für entfernte Kommunikation. Weiterentwicklung ~> DCOM, COM+ und .NET.
Mit .NET werden zuvor eingesetzte SW-Entwicklungskonzept (wie COM) abgelöst.

COM-Standard
Legt Regeln zur Struktur und Entwicklung von COM-Komponenten fest. Zielstellungen an die Komponenten: interoperabel, wiederverwendbar, sprachunabhängig, dynamisch zur Laufzeit austauschbar. Kommunikation zwischen Instanzen erfolgt ausschließlich über ihre Schnittstellen. Standard legt die Schnittstellen fest, die eine COM-Komponente zu implementieren hat (Entwickler kann weitere Schnittstellen definieren) und ein allen Komponenteninstanzen gemeinsames Binärformat. COM-Library bietet zusätzliche Hilfsfunktionen an.
COM-Spezifikation: *Schnittstellen*, *GUID* (Globally Unique Identifier), *Komponentenserver*.
COM-Schnittstellen
Eine Komponente kann beliebig viele Schnittstellen implementieren. Jede SS ist durch einen eindeutigen Identifikator gekennzeichnet: IID (Interface ID).

Wichtigste Schnittstelle: *IUnknown* ~> definiert 3 Operationen:
- *QueryInterface():* Anfrage an Instanz zur Laufzeit, ob bestimmte Schnittstelle unterstützt wird. Ergebnis: Zeiger auf Schnittstelle.
- *AddRef() / Release():* zur Verwaltung von Referenzen auf Komponenteninstanzen. Add: inkrementieren Referenzzähler, Release: dekrementieren. Falls Zähler Null, wird Instanz gelöscht.

Jede Schnittstelle erbt direkt oder indirekt von *IUnknown*. Definition der Komponentenschnittstellen über Schnittstellensprache MIDL (MS Interface Definition Language), orientiert an CORBA-IDL und C (Datentypen).
MIDL-Definition der Schnittstelle *IUnknown* (Beispiel):

```
Interface IUnknown {
HRESULT QueryInterface(
    [in] RFIID riid,
    [out] iid_is(riid) void **ppvObject);
ULONG AddRef();
ULONG Release();
}
```

COM-Schnittstellen dürfen nach Veröffentlichung nicht mehr geändert werden. Änderungen bewirken Generierung eine neuen IID und damit neue Schnittstellenversion.
Weitere verbindliche Schnittstelle: IClassFactory – es ist eine Fabrik als COM-Komponente zur Erzeugung von Instanzen. IClassFactory definiert 2 Methoden:
- CreateInstance(): bei Aufruf erhält Aufrufer eine Referenz auf die Komponenteninstanz.
- LockServer(): bei Methodenaufruf kann Aufrufer explizit angeben, ob Fabrik längere Zeit im Speicher bleiben soll.

Über Methode *CoGetClassObject()* der COM-Bibliothek erhält Client Zugriff auf eine Fabrik. Der Methode wird ein eindeutiger Identifikator GUID (Globally Unique Identi-fier) mitgegeben. Rückgabewert ist eine Referenz auf die Fabrik der Komponente. Erweiterung der ursprünglichen Schnittstelle durch Schnittstelle *IClassFactory2* ~> unterstützt Lizenzierung von Komponenten. Weitere Bibliotheksmethode *CoCreateInstance()* liefert direkt eine Zeiger auf die *IUnknown*-Schnittstelle. Instanziierung der Fabrik erfolgt im Hintergrund.

Globally Unique Identifier (GUID)

Weltweiter eindeutiger Identifikator. Einsatz GUID
- als Interface ID (IID) in COM-Schnittstellen,
- als Class Identifier (CLSID) bei COM-Komponenten.

Ein GUID wird über lokalen Algorithmus unter Berücksichtigung der aktuellen Zeit eindeutig generiert. Algorithmus von OSF verwaltet, eingesetzt in der DCE-RPC-Plattform zur Generierung von Universal Unique Identifiers (UUID). Aufbau einer GUID:
- 48 Bits Kennzeichnung für Rechner,
- 60 Bits für Zeitstempel,
- 4 Bits für Versionsnummer,
- 16 Bits werden anhand der Systemzeit errechnet.

Komponenten werden nach Erstellung unter ihrer CLSID in der Registry angemeldet; sind danach für Anwendungen auffindbar.

Komponentenserver

Komponentenserver sind die Verpackungseinheiten von COM-Komponenten. Inhalt: eine oder mehrere COM-Komponenten, die jeweiligen Fabriken sowie zusätzliche Registrierungs- und Verwaltungsfunktionen.
Verfügbar in 2 Formen (gemäß Instanziierungsprozess):
- als Dynamic Link Library (DLL), auch als In-Process-Server bezeichnet,
- als ausführbare exe-Datei, auch als Out-of-Process-Server bezeichnet (benötigt auch Mechanismus zur Interprozesskommunikation).

Instanziierung der COM-Komponenten erfolgt durch Aufruf der Bibliotheksmethode *CoCreateInstance()*. Sie greift im Hintergrund auf Service Control Manager (SCM) zu ~> realisiert die Instanziierung in der Registry:
- In-Process: Lädt im In-Process-Server die DLL in einen existierenden Prozess. Zugriff auf Komponente erfolgt lokal.
- Out-of-Process: Im Out-of-Process-Server wird exe-Datei in einem neuen Prozess gestartet. Zugriff auf COM-Komponenten remote. Nutzung RPC, Marshalling usw.

11.3.3 Erweiterungen DCOM, COM+, .NET und Dienste in .NET

Weiterentwicklung Komponentenmodell
1996 COM zum **verteilten Komponentenmodell DCOM** (Distributed COM) erweitert, insbes. zur entfernten Kommunikation über entfernte Prozeduraufrufe (RPC).
Erweiterung COM zu DCOM durch folgende Aspekte:
- entfernte *Prozeduraufrufe (RPC)* -> dazu Ausbau der IPC (InterProcess-Communication) für entfernte Prozesse,
- Verwaltung und Zugriff der Komponenten über *Sicherheitsmechanismen* geschützt,
- Komponentenserver können als *Dienste* angemeldet werden.

2000: Erweiterung des COM-Komponentenmodells durch Transaktionsdienst zu **COM+.** Erstmalige Auslieferung mit Windows 2000. COM+ ist integraler Bestandteil aller Windows-Systeme.

Distributed COM (DCOM)
DCOM ist die Erweiterung von COM zu einem Komponentenmodell für verteilte Komponenten. Dazu Basismechanismus der IPC so erweitert, dass Prozesse auf unterschiedlichen Rechnern arbeiten ~> RPC (Remote Procedure Call). Allen Bibliotheksmethoden zur Instanziierung von Komponenten kann mitgeteilt werden, wo angeforderte Komponente zu finden ist (Aufgabe der SCM (Service Control Manager) der beteiligten Rechner).
Sicherheitsmechanismus unterstützt 2 Arten von Zugriffsschutz auf Komponenteninstanzen:
- Activation Security: Festlegung des Rechts zur Instanzverwaltung über ACL (Access Control Lists), verschlüsselt in Registry.
- Call Security: Festlegung zum Zugriffsrecht auf eine Komponenteninstanz, erfolgt über Sicherheitsattribute (für Prozess oder Schnittstelle).

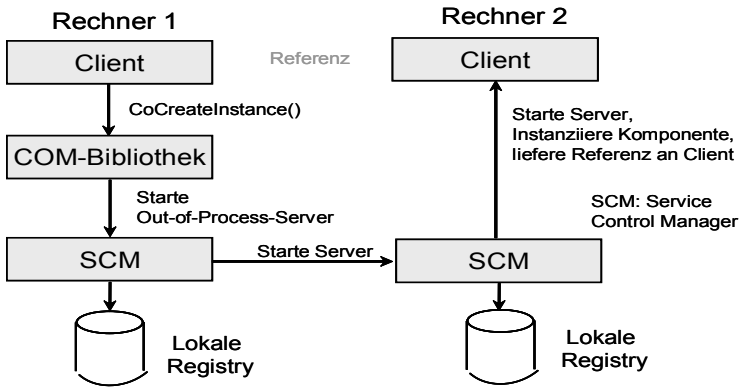


Abbildung 11.5: Instanziierung entfernter Komponenteninstanzen

COM+ ist dieErweiterung des COM-Komponentenmodells durch Transaktionsdienst (2000).
Es bildet den Oberbegriff für einen transaktionalen Anwendungsserver mit
- DCOM als verteiltes Komponentenmodell,
- MTS (MS Transaction Server) als transaktionale Laufzeitumgebung und
- MSQS (MS Message Queue Server) als nachrichtenorientierte Middleware.

COM+ in alle Windows-Systeme intergiert, aber nicht direkt für .NET (zu schwerfällig).

.NET:
Basiert i.w. auf COM-Technologien, benötigt aber Hilfsklassen zur Interoperabilität zwischen COM und .NET. Anwendungen, mit .NET-Framework entwickelt, laufen als Managed Code in der CLR, COM-Komponenten dagegen als Unmanaged Code (unter Laufzeitumgebung des Betriebssystems).
2 Wrapper-Klassen zur Interoperabilität zwischen COM-Komponenten und CLR-Klassen verfügbar:
- COM-callable Wrapper kapseln CLR-Klassen ~> von COM-Komponenten aufrufbar,
- Runtime-callable Wrapper kapseln COM-Komponenten, von CLR-Klassen aufrufbar.

Dienste in .NET
Dienste sind über längeren Zeitraum entstanden, können in unterschiedlicher Form auftreten:
- Enterprise Server: darunter viele bekannte Microsoft-Server zusammengefasst, u.a. Exchange Server, SQL Server, BizTalk Server.
- COM-Dienste: Dienste der COM-Technologie auch den Komponenten der .NET-Plattform verfügbar, u.a. Microsoft Transaction Server (MTS), Message Queue Server (MSQS).
- Subsysteme: ehemalige Microsoft-Technologien, deren Funktionalität in die Framework-Klassenbibliothek wurde, u.a.
  ASP.NET: Technologie zur Entwicklung von Web-Anwendungen, Nachfolger von Active Server Pages (ASP), für serverseitige Web-Interaktionen,
  ADO.NET: Bibliothek zur Unterstützung der Persistenz von Daten bei Datenbank zugriffen, Nachfolger von ActiveX Data Object-Technologie (ADO).

11.4 Kommunikationsmechanismen

11.4.1 Kommunikationstechnologien in .NET

Für die verteilte Programmierung unterstützt .NET (historisch bedingt) mehrere unterschiedliche Kommunikationstechnologien, die z.T. inkompatibel und jeweils mit Vor- und Nachteilen behaftet sind :
- ASP.NET: für Web-Anwendungen, auch als Webservice-Umgebung (Einsatz von Web Services, auch als „XML Web-Services“ bezeichnet).
- DCOM: Technologie für entfernte Prozeduraufrufe (RPC-Technologie), die nicht CLR-Mechanismen (Common Language Runtime) nutzen.
- .NET-Remoting: Technologie für TCP-basierte entfernte Methodenaufrufe. Sie bilden ähnlich Java RMI die Kommunikationsinfrastruktur des .NET-Frameworks.
- MS Message Queue: nachrichtenorientierter Middleware-Server. Message Queue ist auch Teil von COM+ und zählt zu den Enterprise Servern (.NET Enterprise Services).
- WCF (Windows Communication Foundation) für eine dienste-orientierte Architektur.

In den neueren Versionen von .NET sind vorrangig folgende Mechanismen unterstützt:
Web Services, Windows Communication Foundation, DCOM / RPC, NET Remoting Service, NET Enterprise Service (letztere Services als Konkurrenztechnik zum erfolgreichen J2EE).
Die Technologien sind auf jeweils bestimmte Anwendungsgebiete fokusiert, weitgehend unabhängig eingesetzt. Neuere Microsoft Betriebssysteme:
Windows XP Service Pack 2 und Windows Server 2003, Windows XP SP 3
Windows Vista (WCF integriert) und Server-BS Longhorn, Windows 7, Windows 8,
Windows Azure (für Cloud Computing Entwickler).

11.4.2 Windows Communication Foundation (WCF)

Windows Communication Foundation / Indigo
WCF (früherer Bezeichnung Indigo): Konzept von Microsoft für eine dienste-orientierte Kommunikationsplattform für verteilte Anwendungen.

- Vereinheitlichung der verschiedenen Kommunikationstechnologien in .NET, standardisiert, einheitliche dienstorientierte Architektur. Plan: 2006 auf Markt.
- Durch WCF werden die Kommunikationstechnologien DCOM, Enterprise Services, MSMQ, WSE und Web Services unter einer einheitlichen API zusammengefasst.
- WCF gehört zum .NET Framework 3.0, konzipiert für Windows Vista und das Windows Server-Betriebssystem Longhorn (verfügbar auch für die Vorgänger-Version Windows XP und Windows Server 2003).
- Komponente von .NET Framework 3.0 (06.11.2006). Aktuell: .NET Framework 3.5 (2009, enthält auch AJAX).
- WCF: Interoperabilität zu Java Web Services (über Web Services Interoperabilitiy Technology).

Longhorn / Vista / Windows 7 / Windows 8

Window Server-Betriebssystems: Longhorn (2008), mit den Neuerungen

- Avalon: für die Gestaltung der Benutzeroberflächen,
  (~> XAML: Extensible Application Markup Language – Microsoft-Beschreibungssprache für Oberflächengestaltung von Anwendungen der Windows Presentation Foundation (WPF, u.a. .NET Framework 3.5), der Windows Workflow Foundation (WF) und Web-Anwendungen (Silverlight)).
- WinFS: neues, objektorientiertes Dateisystem,
- Indigo (~> WCF): neue Technik für Kommunikation und Web-Services (Komponente von .NET Framework 3.0). Integriert und ausgeliefert in Windows Vista (06.11.2006).

Windows-Betriebssysteme (mit WCF)

- Windows XP Service Pack 3 (2008), Vista (Nachfolger von XP).
- Windows 7 (Arbeitsbezeichnungen: Blackcomb ~> Vienna): Nachfolger von Vista. BS für PC, Laptops, Tablet PC's. Vorgestellt durch Steve Palmer (MS, 09.01.2009). Auslieferung in DE: 22.10.2009.
- Windows 8: optimiert für Tablets und Cloud Computing. Auslieferung Herbst 2012.

Indigo/WCF:

- Anwendungen als Dienste, Kommunikation zw. Diensten erfolgt über sog. Connector: bildet Kern der Indigo-Architektur, mit 4 Konzepten: *Ports, Channels, Nachrichten, Dienste*.
- Formatierung der Nachrichten mit SOAP. Ports sind Kommunikationsendpunkte für Dienste, Channels repräsentieren die verschiedenen Kommunikationskanäle durch Ports, Jedem Kanal kann Typ zugeordnet werden; synchrone und asynchrone Kommunikation.
- Bildet das künftige Framework für die Kommunikation in verteilten Anwendungen:
  - vereint die besten Features von .NET Remoting, ASMX und .NET Enterprise Services zu einem konsistenten Programmiermodell,
  - unterstützt bekannte Standards wie XML und SOAP.

Realisierung des Endpunkt-Konzeptes in WCF durch Trennung in **A**ddress, **B**inding und **C**ontract (ABC-Prinzip):

- Address (Adresse): Address ist ein URI zur Lokalisation des Dienstes (kennzeichnet Ort des Dienstes und Zugriff für Dienstkonsumenten).
- Binding (Anbindung): beschreibt Art der Kommunikation, u.a. Protokoll (HTTP, TCP, UDP und Windows-eigene Protokolle), Kodierung (binär, SOAP-Dokument, eigenes Format), Sicherheitsaspekte (Verschlüsselung, Authentisierung).
  .NET-Framework stellt vorgefertigte Bindings zur Verfügung (noch konfigurierbar) und Möglichkeiten für eigene Bindings (z.B. für XML-RPC-Protokoll).
- Contract (Vertrag): Dienstdefinition, insbes. zu den bereitgestellten Methoden.
  Kontrakte zur Entwicklungszeit als Schnittstellen (Interfaces) in beliebiger .NET-Sprache verfasst u. zur Laufzeit durch WCF in SOAP-Contract umgesetzt (~> plattformunabhängiger Dienstzugriff).

12 Sun-Componentware Java Platform, Enterprise Edition (J2EE / Java EE)

12.1 Java-basierte Middleware-Plattform

Konzeption J2EE (seit 2006 **Java EE)**
Standard für eine verteilte Java-basierte Middleware-Plattform. Sun Microsystems (1998). Erweiterung der Java-Plattform zu einer verteilten Plattform für Enterprise-Anwendungen.
Einordnung/Vergleich mit anderen Plattformen bzw. Technologien:
- **CORBA**: Unterstützt Zugriff auf Objekte und Dienste, auch über unterschiedliche ORB's. Gutes Konzept, aber: hohe Komplexität durch Sprachunabhängigkeit (IDL, Language Mapping) und nur wenige der eigentlich innovativen Dienste umgesetzt ~> diese heute in den Bibliotheken von Java und C# realisiert.
- **COM/DCOM** (Microsoft): Komponentenbasierte Entwicklung ~> Plattform **.NET**. Unterstützung MS-Sprachen, wie oo-Sprache C#, Visual C++, MS-Java u.a. sowie C++, Python, Smalltalk u.a. (als sog. Unmanaged Code). Aber: MS-Abhängigkeit.
- Dagegen **J2EE/Java EE**: Java als gemeinsame Sprache und eine Java-Laufzeitumgebung (Java Runtime), die als Teil der MW-Plattform auf jedem Knoten des VS liegt.

Java Platform, Enterprise Edition (J2EE / Java EE)
Spezifikation einer SW-Architektur für transaktionsbasierte Ausführung von in Java programmierten Anwendungen, insbes. Web-Anwendungen. In der Spezifikation werden Komponenten und Dienste definiert (i.allg. in Java). Daraus werden modulare Komponenten für verteilte Anwendungen entwickelt. Klar definierte Schnittstellen sorgen für Interoperabilität der SW-Komponenten unter-schiedlicher Hersteller und für gute Skalierbarkeit.
Konkurrent: .NET/Microsoft.
Sun integriert Komponentenmodell EJB (Enterprise JavaBeans) in J2EE-Plattform. Plattform stellt Dienste bereit, ein Zugriff ist nicht mehr explizit zu implementieren.
Java Community Process (JCP): Ursprünglich J2EE unter alleiniger Verantwortung von Sun ~> Einbezug der Öffentlichkeit ~> JCP: Gremium: Standardisierungen, Spezifikationen.

12.2 Standard J2EE / Java EE

Versionen
- J2EE ist Weiterentwicklung des J2SE (ursprünglich JDK: Java Development Kit).
  1998: J2EE v1.1 / EJB 1.0: EJB als einzige Komponente (sog. J2EE-Application Server).
  1999: J2EE v1.2 / EJB 1.1: erste stabile Version. XML Deployment Descriptor.
  3 Bean-Typen (Stateful Session Beans, Stateless Session Beans, Entity Beans).
  Aber: unvollständige Persistenz, keine lokalen Bean-Aufrufe.
  2001: J2EE v1.3 / EJB 2.0: Neues Persistenzmodell, Beziehungen zw. Beans möglich, auch asynchrone Kommunikation (Message Driven Beans) und lokale Aufrufe zwischen Beans.
  Nov. 2003: J2EE v1.4 / EJB 2.1: wie Vorgänger, Ergänzung durch Unterstützung von Web-Services, neue Komponenten zur Bearbeitung von XML-Dokumenten und Implementierung JAX-RPC-Protokoll (Java APIs for XML based RPC): Abbildungsregeln zw. Java und WSDL-Datentypen, die auf Standard-XML-Schema beruhen.
- Seit 2006: Java Platform, Enterprise Edition (Java EE), ersetzt die Abkürzung J2EE.
  11. Mai 2006: Java EE v5, 10. Dez. 2009: Java EE v6 (aktuelle Version).
  Erweiterungen, u.a.
  - erweiterte Unterstützung von Web Services, XML, Java, Datenbankanbindung (SQL),
  - Einbindung von Komponenten in Web-Seiten (JSF: Java Server Faces),
  - Verbindung EJB mit JSF (v6).

Weitere Java-Plattformen:
- Java Platform, Standard Edition (J2SE)
- Java Platform, Micro Edition (J2ME)

Architektur J2EE / Java EE
Standard J2EE/Java EE definiert das Architekturmodell einer Java-basierten Middleware-Plattform mit Spezifikation, Referenzimplementierung und Kompatibilitätstest. Kernkomponenten (bezügl. VST):
- RMI (über CORBA-Protokoll IIOP ~> RMI-IIOP) für Kommunikation zwischen verteilten Objekten (und für Integration mit CORBA).
- EJB (Enterprise JavaBeans): ermöglicht, dass Anwendungslogik einfach in Form von Komponenten (Beans) in Anwendungsserver auf Middle-Tier integriert werden kann.

J2EE überträgt das objektorientierte Konzept der entfernten Methodenaufrufe zwischen verteilten Objekten auf entfernte Methodenaufrufe zwischen Komponenten.
Zentrale Bestandteile: Java RMI-IIOP, Komponentenmodell EJB und Container (Laufzeitumgebung für Komponenten). Container verwaltet Komponenteninstanzen und überwacht die Zugriffe. Komponenten in einem Container interagieren nie direkt miteinander. Sie nutzen ausschließlich Kommunikationsmechanismen, Dienste und Protokolle des Containers.

Java-EE-Komponenten erfordern als Laufzeitumgebung eine spezielle Infrastruktur, einen sog. Java EE Application Server. Dieser stellt folgende Funktionalitäten bereit:
- Sicherheit (Security),
- Transaktionsmanagement,
- Namens- und Verzeichnisdienste,
- Kommunikation zwischen Java-EE-Komponenten,
- Persistenzdienste zum langfristigen Speichern von Java-Objekten,
- Management der Komponenten über den gesamten Lebenszyklus (inkl. Instanziierung),
- Installationsunterstützung (Deployment).

Weiterhin kapselt der Application Server den Zugriff auf die Ressourcen des Betriebs-Systems (Dateisystem, Netzwerk, ...).
Weitere Infrastrukturkomponente für die persistente Speicherung von Daten: DBMS (Datenbank-Managementsystem), z.B. relationales DBMS oder ooDBMS. Die Anbindung der DBMS erfolgt meist über einen JDBC-Treiber.

Der clientseitige Zugriff auf Java-EE-Anwendung erfolgt meist über einen Browser oder über Application Clients (Java-Applikationen, CORBA-Komponenten, Web-Service-Clients).
Ein Java-EE-Server wird in verschiedene logische Systeme (sog. Container) unterteilt. Aktuelle Version Java EE v6 erfordert folgende Container:
- EJB-Container als Laufzeitumgebung für Enterprise JavaBeans (EJB),
- Web-Container als Laufzeitumgebung für Servlets und Java Server Pages (JSP),
- JCA-Container als Laufzeitumgebung für JCA Connectoren (JCA: J2EE Connector Architecture - für transparente Integration anderer Systeme, z.B. EAI):
  JCA-Container zwar explizit nicht definiert, aber durch Hersteller immer zu implementieren.
  Im EJB und Web-Container sind Restriktionen definiert, die für die JCA-Laufzeitumgebung nicht gelten, z.B. Starten von Threads, Lesen/Schreiben in Dateien ...
- JMS-Provider als Verwaltungssystem für Nachrichtenwarteschlangen.

Zahlreiche Implementierungen für Java-EE-Server verfügbar, teils frei (Open-Source), Oracle stellt auch eine Referenzimplementierung zur Verfügung:
• Komplette Java-EE-Server (Auswahl)
  - Open-Source-Server: Apache Geronimo, JBoss Application Server, JOnAS, GlassFish, ...
  - Kommerzielle Server: Oracle WebLogic (Übernahme von BEA), IBM WebSphere Application Server (WAS), SAP NetWeaver Application Server, Oracle Application Server, ...

- Separate Web-Container (Servlet-/JSP-Container): Apache Tomcat, Jetty, Resin
- Separate EJB-Container: Apache OpenEJB (als Open Source)

Architektur J2EE v1.4

J2EE v1.4-Standard definiert 4 unterschiedliche Container mit Komponententypen:

- EJB-Container mit Enterprise JavaBeans: EJB-Container läuft auf Middle-Tier im Application Server und stellt die Anwendungslogik in Form von Enterprise Beans bereit.
- Web-Container mit Servlets: Container im Web-Server und unterstützt Webanwendungen.
- Application-Client-Container mit Application Clients: Container läuft auf Client-Tier und unterstützt Java-Clients, die mit EJB-Anwendungen kommunizieren.
- Applet-Container mit Applets: Container entspricht dem Java-Plug-In eines Browsers und läuft auf der Client-Tier einer Web-Anwendung.

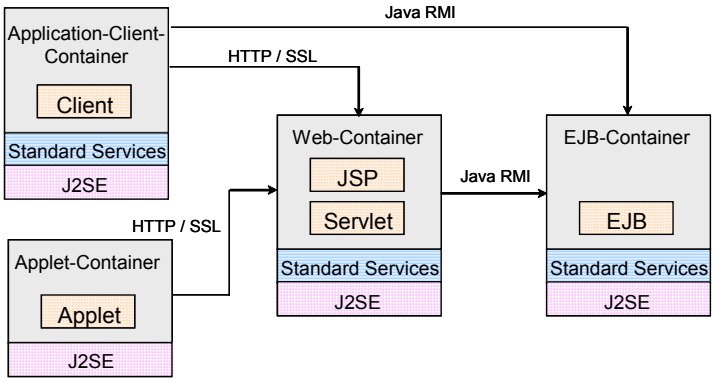


Abbildung 12.1: J2EE-Containerarchitektur

Client-Seite: J2SE (Java 2 Platform Standard Edition) dient als Basislaufzeitumgebung (am Desktop bzw. im Browser). Serverseite (Webserver bzw Anwendungsserver (Application Server)): J2EE somit J2SE, erweitert um Web-Container bzw. EJB-Container und Services.

12.3 Java (2) Platform, Enterprise Edition

J2EE-Komponenten

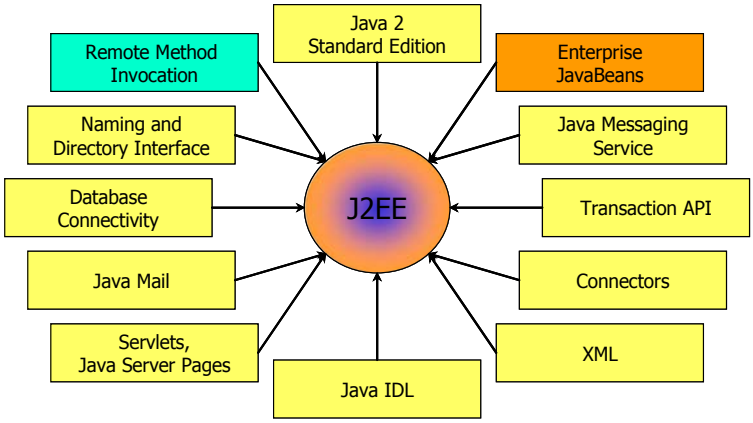


Abbildung 12.2: J2EE-Komponenten

Weitere Komponenten in Java EE, z.B.

- JAX-WS für Web-Service-Einsatz (v5)
- JavaServer Faces (JFS). Einbinden von Komponenten für Benutzerschnittstellen in Web-Seiten (v5)
- Context and Dependency Injection (CDI): Einsetzen von Feldern; verbindet JSF mit EJB (v6).

Bestandteile der Java 2 Plattform, Enterprise Edition (J2EE):

- Java 2 Standard Edition (J2SE): stellt Kernfunktionalität für den clientseitigen Einsatz der Programmiersprache Java bereit.
- Enterprise JavaBeans (EJB): Kernstück von J2EE; es definiert ein serverseitiges Komponentenmodell, welches komplexe Middlewarefunktionalität ohne zusätzlichen Aufwand für den Anwendungsentwickler bereitstellt.
- Java Messaging Service (JMS): JMS-API bietet verteilten Objekten die Möglichkeit, asynchron und zuverlässig über Nachrichten zu kommunizieren.
- Java Transaction API (JTA) und Java Transaction Service (JTS): JTA und JTS Spezifikation bietet Komponenten zuverlässige Unterstützung eines Transaktionskonzeptes.
- Remote Method Invocation (RMI): erlaubt Interprozesskommunikation. RMI-IIOP ist eine Erweiterung von RMI, die kompatibel zum Internet-Inter-ORB-Protokoll ist und für die Integration mit CORBA genutzt werden kann.
- Java Naming and Directory Interface (JNDI): Der Java-Namens- und Verzeichnisdienst lokalisiert Komponenten und andere Ressourcen im Netz.
- Java Data Base Connectivity (JDBC): JDBC-API wird benutzt, um mit Datenbanksystemen zu kommunizieren.
- Java Mail: Versenden plattform- und protokollunabhängig Emails in Javaprogrammen.
- Servlets und Java Server Pages (JSP): Technologien zur Request/Response-basierten Kommunikation mit Clientprogrammen über das HTTP-Protokoll. JSP ~> serverseitige Interaktion (like CGI bzw. PHP in WWW).
- Java IDL: Java-basierte Implementation von CORBA. Sie erlaubt verteilten Objekten, CORBA-Dienste zu nutzen.
- eXtended Markup Language (XML): Metabeschreibungssprache für Dokumente in J2EE.

12.4 Enterprise JavaBeans (EJB)

Serverseitiges Komponentenmodell

Enterprise JavaBeans ist eine serverseitige Komponentenarchitektur. Es ermöglicht das Entwickeln von verteilten Anwendungen, ohne dass sich Entwickler um Implementierung von Details kümmern muss wie Skalierbarkeit, Sicherheit, Persistenz oder Zuverlässigkeit. Bereitstellung einer funktionsfähigen Infrastruktur mit standardisierten Schnittstellen ~> gestattet

- beschleunigten Entwicklungsprozess (time-to-market),
- Erstellung portabler und wiederverwendbarer Software (re-usable).

Unterstützung (Architektur, Laufzeit, Clientzugriff) in Java-Technologie durch

- Enterprise JavaBeans Technologie (EJB), integriert in
- Java (2) Platform Enterprise Edition (J2EE / Java EE).

Vergleichbare Technologien mit Komponentenmodellen:

- Microsoft: Plattform .NET mit Komponentenmodell COM / DCOM,
- CORBA 3.0 mit Komponentenmodell CCM (CORBA Component Model),.

EJB‘s bilden Kernstück der Spezifikation J2EE / Java EE. Sie definieren, wie serverseitige Komponenten geschrieben werden und legen Verbindlichkeiten zwischen den Komponente und dem verwaltenden Applikationsserver fest.

Komplexe Middleware-Aufgaben, wie Ressourcenverwaltung, Sicherheit, Transaktions-Management usw. werden dem Applikationsserver übertragen. Dadurch kann sich der Komponentenentwickler auf die Implementation der Geschäftslogik konzentrieren

~> höhere Effektivität, kürzere Entwicklungszeit (time-to-market),

~> Kapselung existierender Komponenten, deren Integration wird durch wohldefinierte Migrationspfade unterstützt.

EJB: stellt umfassendes Konzept zur Realisierung eines serverseitigen Komponentenmodells dar: Es eignet sich hervorragend in Zusammenhang mit anderen J2EE-Technologien zum praktischen Einsatz (Beispiel: MEMOS / Universität Leipzig).

Ergänzende Technologien: CORBA, Web-Services (mit SOAP, XML, WSDL, UDDI), Web-Technologien (PHP, AJAX: Asynchronous JavaScript and XML, HTML5, …) z.B. Web 2.0, SOA (Service Oriented Architecture), Cloud Computing.

Bean-Typen

EJB-Standard: Teil des J2EE-Standards, definiert das Komponentenmodell am Anwendungsserver. Standard umfasst Art und Struktur der Beans, die zur Entwicklung durchzuführenden Aufgaben und Rollen, Aufgaben des EJB-Containers.
EJB unterstützt als Programmiermodell den entfernten Methodenaufruf, synchrone und asynchrone Kommunikation. Sprache: Java.
4 verschiedene Bean-Typen:

- Stateless Session Beans: zustandslos; zur Verwaltung von Client-Sitzungen für die Dauer eines Methodenaufrufs.
- Stateful Session Beans: zustandsbehaftet; zur Verwaltung von Client-Sitzungen über mehrere Methodenaufrufe hinweg.
- Entity Beans: stellen fachliche Entitäten einer Anwendung dar. Persistierung der Attribute.
- Message Driven Beans: zustandslos; Bearbeitung asynchron empfangener Nachrichten.

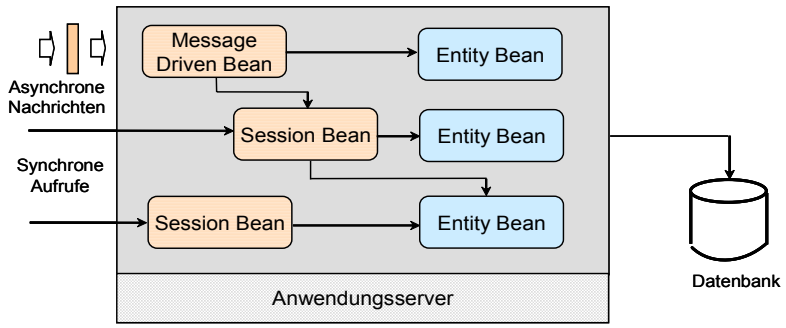


Abbildung 12.3: Bean-Typen im Container

Die verschiedenen EJB-Komponenten werden in Anwendung in Kombination eingesetzt:

- Session Beans und Message Driven Beans bilden Schnittstelle zur Umgebung und empfangen synchrone bzw. asynchrone Aufrufe.
- Entity Beans bilden den Kern der Anwendung.

Aufbau einer Bean

Bean („Bohne“) setzt sich aus verschiedenen Klassen und Schnittstellen zusammen. Teile der Klassen sind vom Entwickler zu erstellen, andere generiert. Genereller Aufbau einer Bean nahezu identisch für alle Bean-Typen.
Struktur einer Bean wird durch folgende Klassen und Schnittstellen gebildet:

- (Local) Home Interface als Verwaltungsschnittstelle für Bean Instanzen,
- Remote / Local Interface als fachliche Schnittstelle der Bean,
  Angaben Local und Remote kennzeichen, wie auf jeweilige Schnittstelle zugegriffen wird. Local Interface für lokale, Remote Interface für entfernte Methodenaufrufe.
- verschiedene Klassen (wie Stubs, Skeleton) sowie technische Klassen zur Einbettung der Beans in den Container,
- Deployment Descriptor für deklarative Beschreibung der Schnittstelle zum Container.

Erstellung Schnittstellen und Bean-Implementierung durch Entwickler. Alle weiteren Klassen sowie Deployment Descriptor (durch Entwickler anpassbar) werden generiert.

Deployment Descriptor

Deployment Descriptor definiert die Schnittstelle zum Container. Festlegung zu

- unter welchem Namen die Bean im Namensdienst veröffentlicht wird,
- Bean-Typ und Namen der ausführbaren Dateien.

Zusätzlich werden Dienste angegeben, die Bean-Instanzen vom Container erwarten, z.B. Transaktionsverhalten der Beans sowie Abbildung der Bean-Attribute auf Felder der DB. Außerdem Ressourcen festgelegt, die Beans nutzen wollen, z.B. andere Beans oder DB-Verbindung. Angaben in XML definiert und zur Laufzeit vom Container ausgelesen. Vorteil: Änderungen (z.B. andere DB) durch Anpassung des Deployment Descriptors realisierbar.
Standardmethoden werden durch eine Bean-Klasse implementiert und dienen Container zur Verwaltung der Bean-Instanzen. Aufruf durch Container zu definierten Zeitpunkten, z.B. bei Zustandswechsel. Zusätzliche Aktivitäten kann Entwickler durch Methoden implementieren.

Schrittweise Entwicklung von Geschäftsprozessen mit EJB

- Modularisierung in Form wiederverwendbarer EJB-Komponenten;
- Standardisierung eines Containers, um EJB-Komponenten in einem verteilten mehrschichtigen Serversystem betreiben zu können.

Anschließend werden die verschiedene EJB-Komponenten zu komplexen Anwendungen kombiniert und auf einem verteilten System implementiert.
Im EJB-Standard wird Arbeitsteilung durch die **6 Parteien des Rollenkonzepts** erreicht:

Enterprise Bean Provider: Der Bean Provider entwickelt die Enterprise Beans (liefern Funktionalität). Ergebnis: eine ejb-jar Datei, die mehrere Beans enthalten kann. Er ist verantwortlich für die Java-Klassen, die die Geschäftsobjekte implementieren, die Definition von Home- und Remote-Interface und den Deployment Descriptor, der die strukturellen Informationen über die Beans enthält. Bean Provider ist in der Regel ein Anwendungsentwickler, der kein Experte der Systemprogrammierung sein muss und oft auch die Rolle des Application Assemblers übernimmt.

Application Assembler: Er kombiniert mehrere Beans zu einer komplexeren Einheiten und entwickelt damit einen Workflow. Dazu gehört auch die Entwicklung weiterer Beans, Nutzerschnittstellen und von Client-Code zum Zugriff auf Bean-Funktionalitäten.

EJB Deployer: Er übernimmt die vom Bean Provider oder Application Assembler bereitgestellten ejb-jar Dateien und gibt sie an eine spezifische Arbeitsumgebung weiter, die auch den EJB Server und Container enthält, nimmt ggf. Anpassungen vor und löst externe Abhängigkeiten auf.

EJB Container Provider: Er stellt die für Deploymentprozess notwendigen Werkzeuge und den EJB Container zur Verfügung, welcher die Runtime-Umgebung für weitergegebene Enterprise Beans darstellt.

EJB Server Provider: Er wird in der aktuellen EJB-Architektur vom selben Anbieter wie der Container Provider repräsentiert und bietet auf seinem Server eine Laufzeit-Umgebung für einen oder mehrere Container an. Es ist daher nicht notwendig, Schnittstellen zwischen EJB Server und EJB Container festzulegen.

System Administrator: Er ist für die Konfiguration der Computer- und Netzwerkinfrastruktur, die den EJB Server enthält, verantwortlich und überwacht die Funktion der Enterprise Applications zur Laufzeit. Umzusetzung der Aufgaben schreibt die EJB-Architektur nicht vor. EJB Server und EJB Container Provider bieten i.d.R. dazu Überwachungs- und Verwaltungswerkzeuge an.

13 Abbildungsverzeichnis

14 Literatur

Beer, W.; Birngruber, D.; Mössenböck, H.; Wöß, A.: Die .NET-Technologie. Grundlagen und Anwendungsprogrammierung. dpunkt, 2003

Borghoff, U.; Schlichter, J.: Rechnergestützte Gruppenarbeit. Springer, 1995

Couloris, G.; Dollimore, J.; Kindberg, T.: Verteilte Systeme. Konzepte und Design. Pearson Studium / Addison Wesley, München, 2002

Hammerschall, U.: Verteilte Systeme und Anwendungen. Pearson Studium, München, 2005

Masak, D.: SOA ? - Serviceorientierung in Business und Software. Springer, 2007

Müller, G.; Eymann, T.; Kreutzer, M.: Telematik- und Kommunikationssysteme in der vernetzten Wirtschaft. Oldenbourg Verlag München/Wien, 2003

Orfali, R.; Harkey, D.: Client/Server Programming with Java and CORBA. Wiley, 1998

Popien, C.; Schürmann, G.; Weiß, K.-H.: Verteilte Verarbeitung in Offenen Systemen. Das ODP-Referenzmodell. Teubner, Stuttgart, 1996

Schäfer, G.: Netzsicherheit. dpunkt-Verlag, Heidelberg, 2003

Sloman, M.; Kramer, J.: Verteilte Systeme und Rechnernetze. Hanser, München, 1988

Siegel, J.: CORBA Fundamentals and Programming. Wiley, New York, 1996

Stevens, D.L.: Netzwerkprogrammierung. Prentice-Hall, München, 1994

Tanenbaum, A.S.: Moderne Betriebssysteme. Hanser, München, 1998

Tanenbaum, A.S.; Steen, M.v.: Verteilte Systeme. Pearson Studium, München, 2003

Weber, M.: Verteilte Systeme. Spektrum Akademischer Verlag, Heidelberg, 1998