Zusammenfassung

**Lernziele**Verteilte Software-Systeme (VSS) sind allgegenwärtig in beruflichem und privatem Alltag; sie führen Konzepte und Technologien aus unterschiedlichen Bereichen der Informatik zusammen. Nach dem Besuch dieses Moduls sind Sie in der Lage, VSS zu analysieren, zu entwerfen und mit Hilfe von Middleware und Frameworks zu realisieren:

* Sie können die charakteristischen Eigenschaften von VSS und die grundlegenden Fragestellungen beim Entwurf von VSS nennen.
* Sie können synchrone Remote Procedure Calls (RPC) und asynchrone Messaging-Kommunikationskanäle konzipieren und implementieren.
* Sie können die Konzepte und Funktionsweisen der wichtigsten Middleware-Dienste zum Bau von VSS erklären und beispielhafte Middleware-Implementierungen vergleichen und bewerten.
* Sie können operationale Infrastrukturen für verteilte Systeme unter Verwendung von Deployment Patterns entwerfen.

**Unterlagen / Bücher (nicht Pflicht)**

* Tanenbaum, M. van Steen, Distributed Systems, 2nd Edition, Pearson Education, 2007 (zweite Hälfte der Vorlesung; auch auf Deutsch verfügbar)
* J. Dunkel, A. Eberhart, S. Fischer, C. Kleiner, A. Koschel, Systemarchitekturen für verteilte Anwendungen, Hanser, 2008 (als E-Book in der Bibloithek vorhanden)
* P. Brown, Implementing SOA – Total Architecture in Practice, Addison Wesley 2008 (ausgewählte Kapitel, im letzten Teil der Vorlesung)
* M. Nygard, Release It!, http://www.infoq.com/articles/nygard-release-it
* MSDN Deployment Patterns, Operational (Topology) Modeling White Paper von IBM und weitere Assets zum Operational Modelling (z.B. IBM Hardware Sizing Tool, Palladio Developer Wiki)
* INCOSE Systems Engineering for Large Infrastructure Projects
* Online Resourcen wie highscalability.com und Udacity Web Application Design Lecture (mit Reddit-Architekturevolution)

**Lerninhalte***Charakteristische Eigenschaften verteilter Software-Systeme:*

* Vom Programm zum System: Verteilungsdimensionen, Kopplungsarten, Designherausforderungen
* VSS-Anwendungsgebiete in der Praxis, z.B. Unternehmensanwendungen, Distributed Control Systems, World-Wide Web (WWW)
* Wichtige Architekturstile und Anwendungstopologien: Client-Server, Hub-and-Spoke, Peer-to-Peer

*Remoting (Netzwerkprogrammierung):*

* Synchrone Kommunikation: TCP/IP Sockets, Remote Procedure Calls, HTTP Web Services
* Asynchrone Kommunikation mit Message-Oriented Middleware (MOM), Enterprise Integration Patterns
* weitere Message Exchange Patterns und Kommunikationsprotokolle im Überblick

*Zentrale Konzepte für den Entwurf verteilter Systeme:*

* Naming
* Synchronization

*Verteilte Algorithmen und Datenstrukturen:*

* Namensauflösung
* Verteilte Hash-Tabellen
* Lamport-Uhr
* Vektor-Uhr

*Anwendung verteilter Systeme. Zum Beispiel:*

* BitTorrent
* Bitcoin
* Distributed batch processing

*Operationale Modelle und Deployment Patterns für das qualitätsgetriebene Infrastrukturdesign:*

* Performance und Skalierbarkeit
* Robustheit und Verfügbarkeit
* Systemmanagement und Auditierbarkeit
* Middleware-Produkte und Frameworks zum Bau von VSS
* Plattform-Auswahlkriterien und Designentscheidungen im VSS-Entwurf

*Ausblick auf spezielle VSS – Event-Driven Architectures, Cloud Computing, Service-Oriented Architectures*

*Fallstudien aus der Industriepraxis*

***Einen Teil des Moduls wird in Englisch gehalten.***

Übersicht des Modules

[Einführung 5](#_Toc476555973)

[Motivation 5](#_Toc476555974)

[Software Programm und Software System 5](#_Toc476555975)

[Definition Distributed System (zu Deutsch: Verteiltes System) 5](#_Toc476555976)

[Centralization and Transparency Types (ISO) 6](#_Toc476555977)

[Design-Herausforderungen und Lösungsansätze 6](#_Toc476555978)

[Vorteile verteilter Software-Systeme (VSS) 6](#_Toc476555979)

[Herausforderungen bei der Entwicklung verteilter Anwendungen 6](#_Toc476555980)

[Middleware als VSS Enabler 7](#_Toc476555981)

[ACID 7](#_Toc476555982)

[Problemstellungen und Lösungsstrategien 7](#_Toc476555983)

[Architekturstile zum Bau von VSS 8](#_Toc476555984)

[Verschiedene Stile 8](#_Toc476555985)

[Client-Server 8](#_Toc476555986)

[Peer-to-Peer VSS Architecture Style 10](#_Toc476555987)

[WWW und Netzwerkprogrammierung in Java (erste Schritte) 10](#_Toc476555988)

[WWW ist meistens eine REST-basierte Architektur 10](#_Toc476555989)

[REST Beispiel 10](#_Toc476555990)

[Media Typen (MIME Types) 11](#_Toc476555991)

[Architektonische und Designtechnische Aspekte für verteilte Software 11](#_Toc476555992)

[Idempotent 11](#_Toc476555993)

[Beispiel zur Wichtig von Time-Outs und Error Handling 12](#_Toc476555994)

[Netzwerkprogrammierung 12](#_Toc476555995)

[Architekturstile im Vergleich 12](#_Toc476555996)

[Inter-Process Kommunikation Varianten 12](#_Toc476555997)

[TCP/IP Sockets 13](#_Toc476555998)

[Sockets im Überblick 13](#_Toc476555999)

[Generic TCP Application 13](#_Toc476556000)

[Socket-Programmierung: Grundkonzepte 13](#_Toc476556001)

[Java Socket API 14](#_Toc476556002)

[Beispiel 14](#_Toc476556003)

[Berkeley Sockets und das Message Passing Interface (MPI) 14](#_Toc476556004)

[IPv4 vs. IPv6 14](#_Toc476556005)

[Vorteile und Nachteile von Java Sockets 14](#_Toc476556006)

[WebSockets 15](#_Toc476556007)

[Das Protokoll 15](#_Toc476556008)

[Client API und Support 15](#_Toc476556009)

[Server Support 15](#_Toc476556010)

[UDP (User Datagram Protocol) 🡺 Vergleich mit TCP 15](#_Toc476556011)

[Struktur eines UDP Programms 16](#_Toc476556012)

[Beispiel - Sender und Empfänger 16](#_Toc476556013)

[UDP vs. TCP 16](#_Toc476556014)

[TCP/IP vs. UDP/IP 17](#_Toc476556015)

[Message Exchange Patterns 17](#_Toc476556016)

[Asynchronous Messaging 18](#_Toc476556017)

[Einführung in Messaging 18](#_Toc476556018)

[Nachrichtenkonzept und die Komponenten 18](#_Toc476556019)

[Message-Queuing Model und RPC’s 18](#_Toc476556020)

[Nachrichten nutzen um Applikationen zu verbinden und Daten zu verteilen 18](#_Toc476556021)

[Message-Oriented-Middleware (MOM) 19](#_Toc476556022)

[Hub-and-Spoke Architecture with CMB 19](#_Toc476556023)

[Java Message Service (JMS) API 19](#_Toc476556024)

[Reliability levels 19](#_Toc476556025)

[JMS Message-Struktur 19](#_Toc476556026)

[Beispiel für einen JMS Message Provider 20](#_Toc476556027)

[Andere APIs oder Provider 20](#_Toc476556028)

[Messaging Middleware Provider 20](#_Toc476556029)

[IBM MQ 20](#_Toc476556030)

[MS MQ 22](#_Toc476556031)

[Active MQ 22](#_Toc476556032)

[MOM API Primitives (Platform-Independet Level) 22](#_Toc476556033)

[Enterprise Integration Patterns 22](#_Toc476556034)

[RabbitMQ (based on AMQP) 22](#_Toc476556035)

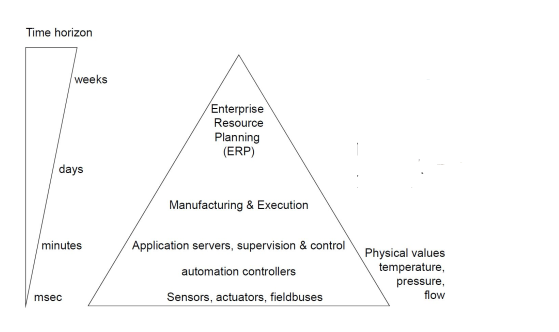
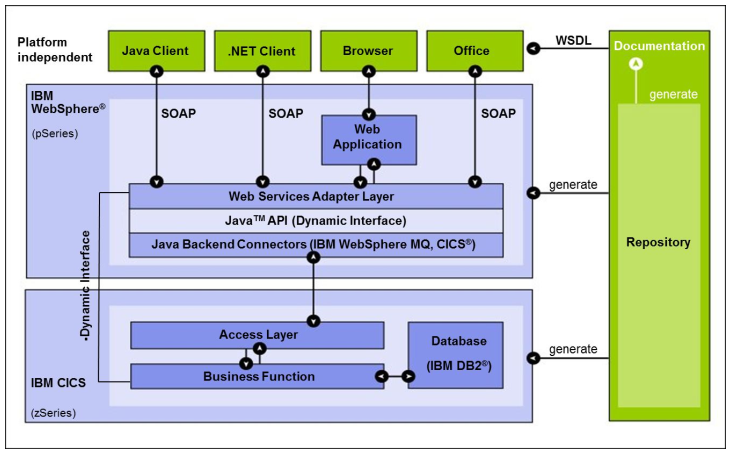
[Patterns 22](#_Toc476556036)

[Vorlesung Woche 4 24](#_Toc476556037)

# Einführung

## Motivation

Verteilte Software-Systeme (VSS) sind allgegenwärtig in beruflichen und privaten Alltag. Sie führen Konzepte und Technologien aus unterschiedlichen Bereichen der Information zusammen. Dazu zählen die Netzwerkprogrammierung (Remoteing), Betriebssysteme und Middleware, User Interfaces, Datenbanken und Algorithmen.

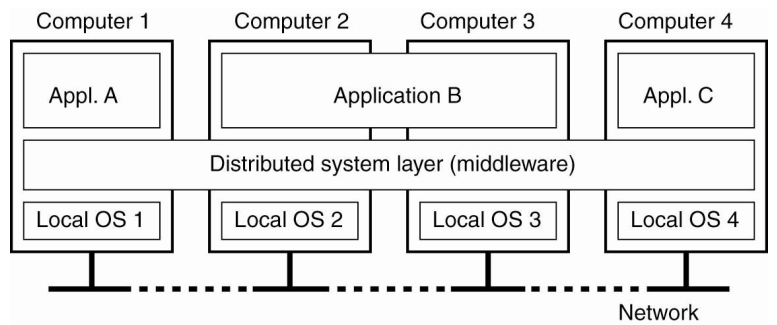
Beispiele dafür sind Unternehmenanwendungen im Core Banking oder auch Prozessautomation und-überwachung aus der Industrie.  


### Software Programm und Software System

|  |  |
| --- | --- |
| **Programm**   * Hat ein Anfang und ein Ende * Wird meist anhand von Parametern ausgeführt. * Kann offline geupdatet werden (Stop, Neustart) * Ist Zeitlich gesteuert, hat einen Callstack | **System (von Programmen/Systemen)**   * Hört vielleicht gar nie auf. * Akzeptiert den Input zu jeder Zeit (Asynchron über externe Schnittstellen) * Meistens wird es dynamisch während der Laufzeit rekonfiguriert. * Ist Event-gesteuert. |

### Definition Distributed System (zu Deutsch: Verteiltes System)

Ein verteiltes System ist eine Sammlung von unabhängigen Computer die gegenüber dem Benutzer als ein ganzes System auftreten.



### Centralization and Transparency Types (ISO)

|  |  |
| --- | --- |
| **Konzept** | **Beispiel** |
| Centralized services | A single server for all users |
| Centralized data | A single on-line telephone book |
| Centralized algorithms | Doing routing based on complete information |

|  |  |
| --- | --- |
| **Transparency** | **Beschreibung** |
| *Access* | Hide differences in data represendation and how a resource is accessed. |
| *Location* | Hide where a resource is located |
| *Migration* | Hide that a resource may move to another location |
| *Relocation* | Hide that a resource may be moved to another location while in use |
| *Replication* | Hide that a resource is replicated |
| *Concurrency* | Hide that a resource may be shared by several competivitve users |
| *Failure* | Hide the failure and recovery of a resource. |

## Design-Herausforderungen und Lösungsansätze

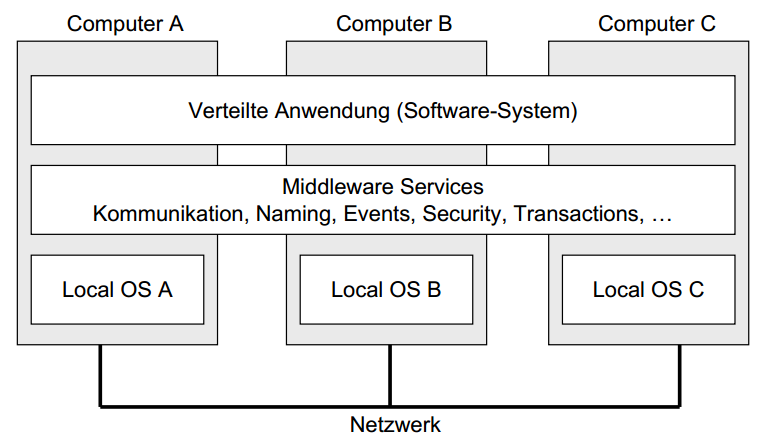
### Vorteile verteilter Software-Systeme (VSS)

VSS bilden die verteilte Realität ab. Die Benutzer arbeiten an verschiedenen Arbeitsplätzen. Die Unternehmensstandorte und Abteilungen sind in der Regel verteilt. VSS führen zu einer Performance Steigerung, wenn mehrere Rechner an einer Aufgabe parallel arbeiten können. Eine Skalierbarkeit ist dadurch gegeben, dass Leistungsengpässe durch zusätzliche Hardware kompensiert werden können. Zudem ist ein System so Fehlertoleranter, da Software auf mehreren Rechnern repliziert werden.

### Herausforderungen bei der Entwicklung verteilter Anwendungen

* Komplexe Kommunikation / Systeme
  + Interprozess-Kommunikation statt lokaler Methodenaufrufe
  + Heterogene Systeme (Rechner, Betriebssysteme, Hardware, Sprachen)
* Performanz Problem
  + Kommunikation über langsame Netzwerke (Latenz, Durchsatz)
* Zuverlässigkeit 🡺 CFIA – Component Failure Impact Analysis
  + Alle Netzzugriffe sind potentiell unsicher
  + Maschinenausfälle
* Transaktionssicherheit
  + Mehrere Benutzer greifen gleichzeitig auf Daten zu (zur Sicherstellung der Konsistenz ist Isolation von Transaktionen erforderlich).

### Middleware als VSS Enabler

Eine Middleware ist eine infrastrukturelle Software zur Kommunikation zwischen Software-Komponenten und Anwendungen auf verschiedenen Computern.

* Verteilungsplattform mit entsprechenden Protokollen
* Höheres Abstraktionsniveau als einfacher Datenaustausch
* Verbirgt Komplexität der zugrundeliegenden Applikation/Infrastruktur.

Gründe für die Einführung einer Middleware sind die Überwindung von der Heterogenität oder die Vereinfachung der Erstellung verteilter Anwendungen.

**Kommunikationsorientierte Middleware**  
Der Schwerpunkt liegt in der Abstraktion (Vereinfachung) von der Netzwerkprogrammierung (TCP/IP, Socket APIs). Beispiele sind Java Remote Method Invocation (RMI) oder Webservices wie SOAP.

**Anwendungsorientierte Middleware**  
Der Schwerpunkt liegt in der weiterreichenden Unterstützung verteilter Anwendungen. Ein komplexer Aufbau führt zu zahlreichen Zusatzdiensten wie Discovery, Sicherheit, Zuverlässigkeit, verteile Transaktionen und Session. Beispiele dafür sind CORBA, J2EE oder verteilte Betriebssysteme.

### ACID

**Atomicity**  
Von einer atomaren Operation spricht man, wenn eine Sequenz von Daten-Operationen entweder ganz oder gar nicht ausgeführt wird (Alles-oder-nichts-Eigenschaft)

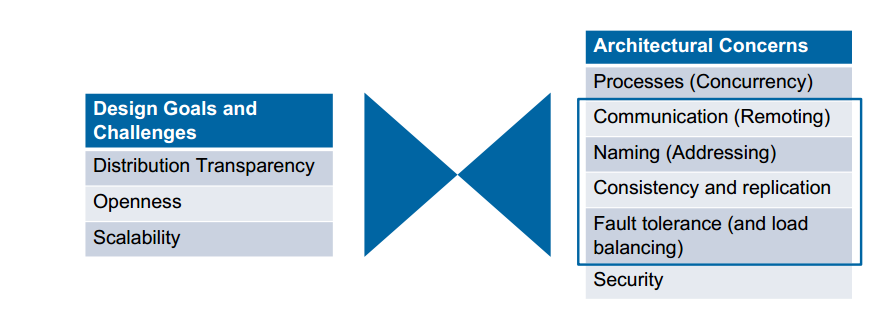
**Consistency**  
Konsistenz heißt, dass eine Sequenz von Daten-Operationen nach Beendigung einen konsistenten Datenzustand hinterlässt, falls die Datenbank davor auch konsistent war.

**Isolation (Abgrenzung)**Durch das Prinzip der Isolation wird verhindert/eingeschränkt, dass sich gleichzeitig in Ausführung befindliche Daten-Operationen gegenseitig beeinflussen.

**Durability**  
Der Begriff Dauerhaftigkeit sagt aus, dass Daten nach dem erfolgreichen Abschluss einer Transaktion garantiert dauerhaft in der Datenbank gespeichert sind. Die dauerhafte Speicherung der Daten muss auch nach einem Systemfehler (Software-Fehler oder Hardware-Ausfall) garantiert sein. Insbesondere darf es nach einem Ausfall des Hauptspeichers nicht zu Datenverlusten kommen. Dauerhaftigkeit kann durch das Schreiben eines Transaktionslogs sichergestellt werden. Ein Transaktionslog erlaubt es, nach einem Systemausfall alle in der Datenbank fehlenden Schreib-Operationen zu reproduzieren

### Problemstellungen und Lösungsstrategien

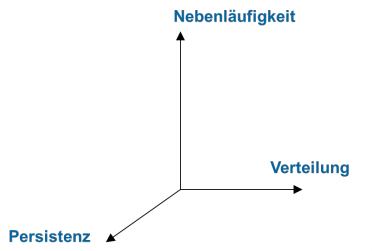
Die VSS-Vorlesung soll eine Mischung aus Wissen und Können vermitteln bezüglicher dieser Problemstellungen und Lösungsstrategien.



## Architekturstile zum Bau von VSS

### Verschiedene Stile

* Client-Server
* Distirbuted objects
* Hub-and-spoke
* Event-Driven Architecture
* Peer-to-Peer (P2P)

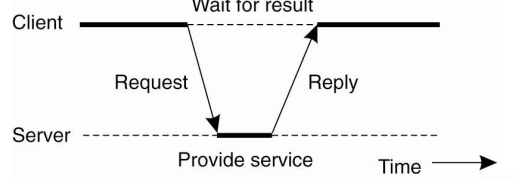
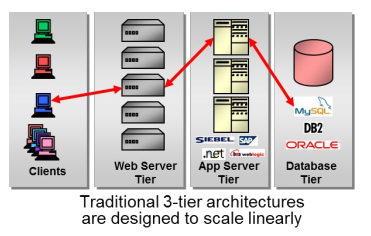
Die unterschiedlichen Architektur-Stile werden durch Integrations-Optionen beeinflusst.

**Transparenz betreffend Nebenläufigkeit**  
Daten oder andere Ressourcen möglicherweise gleichzeitig von konkurrierenden Benutzern bzw. Applikationen verwendet werden.

**Transparenz betreffend Persistenz**  
verbirgt, ob Ressource im lokalen Speicher (Memory) oder auf einer Festplatte ist.

**Transparenz betreffend Verteilung**  
verbirgt, ob die aufgerufene Komponente lokal oder entfernt ist.

### Client-Server

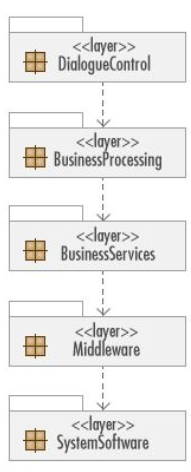
**Tier 1** User PC running browsers (front-end)

**Tier 2** Web server and presentation logic, Application server und business logic, integration

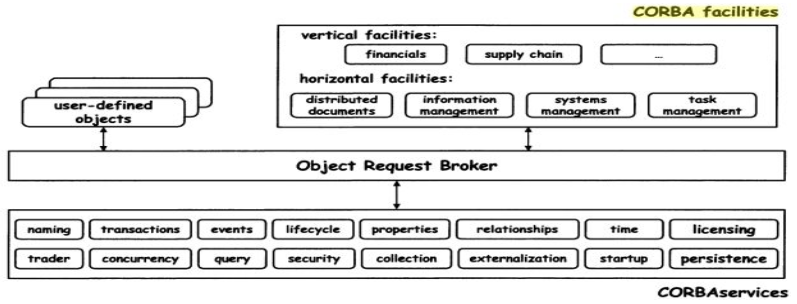
**Tier 3** Database and other backend systems

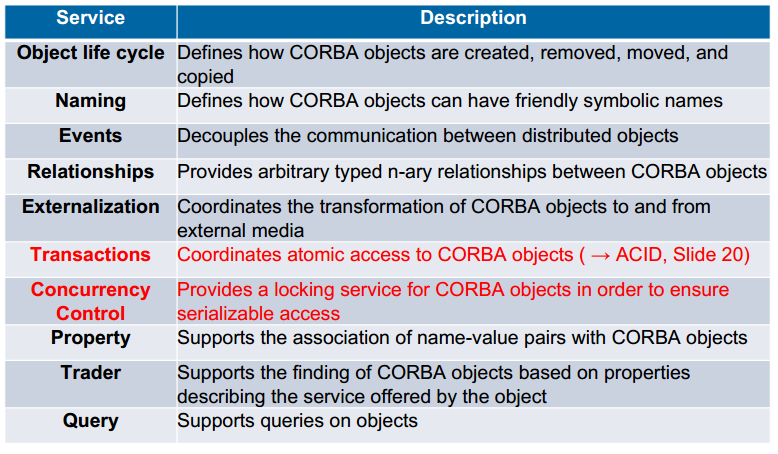
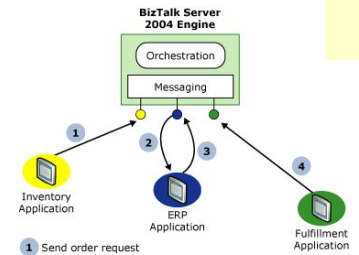
Jedes Tier hat seine eigenen Layer Architektur. Es kann gleiche Teile geben, muss es aber nicht.

**Beispiel einer geschichteten Architektur**



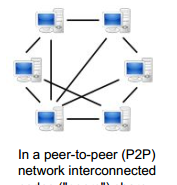
|  |  |
| --- | --- |
| **DialogueControl** | The dialogue control layer handles user-system interactions and use case logic. |
| **BusinessProcessing** | The business processing layer contains application-specific services that handle use case step logic and choreography |
| **BusinessServices** | The business services layer contains more general business components that may be used in several applications. |
| **Middleware** | The middleware layer contains components such as interfaces to databases and platform-independent operating system services. |
| **SystemSoftware** | The system software layer contains components such as operating systems and databases. |

**CORBA (Common Object Request Broker Architecture)**  


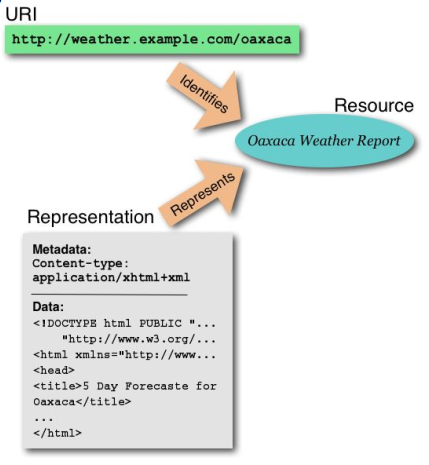
 

### Peer-to-Peer VSS Architecture Style

P2P Systeme implementieren eine verteile Applikationsarchitektur, welche den Workload in Partitionen zwischen den Peers aufteilt. Die Peers sind alle gleichberechtigt. Ein P2P System verbindet dynamisch alle Nodes um Traffic zu routen und somit Tasks auszulagern. Es bietet den Vorteil, dass man skalieren kann ohne die zentralen Ressourcen auszubauen. Es werden die Prozessor- und Netzwerkpower der Client genutzt. Meist sind dort mehr als genug Leistung vorhanden. Dabei gibt es keinen Single Point of Failure. Beispiele sind Skype oder BitTorrent.

## WWW und Netzwerkprogrammierung in Java (erste Schritte)

Das weltweit grösste verteilte Software Systeme, ist das WWW. Es entspricht in etwa dem Client-server architectural style. Ein Naming findet mit Uniform Resource Identifiers (URIs) statt. Die Kommunikation geschieht mit http über TCP/IP. Im Internet gibt es eigentlich keinen Point of Failure, da unglaublich viel Redundanz vorhanden ist.

### WWW ist meistens eine REST-basierte Architektur

REST (Represential State Transfer = Architekturstyle für verteilte Applikationen. REST unterstützt alle vier CRUD Operationen. Es basiert auf einem «stateless» Client-Server Kommunikationsprotokoll. Logisch gesehen ist http simple und einfach dazu da, Aufrufe zwischen zwei Maschinen auszutauschen. Es ist weniger kompliziert als CORBA, RPC oder SOAP.

### REST Beispiel

#### Abfrage in einer Telefonbuchapplikation für Details zu einem Benutzer

**Nutzung von Web Services und SOAP**

Das Resultat könne eine XML-Datei sein, welche als Payload in der SOAP-Antwort eingebettet ist.

**Nutzung von Rest**

Der URL wird mit einem GET Request zum Server gesendet. Als Antwort werden die reinen Daten zurückgeben ohne jeglichen Müll rund herum.

### Media Typen (MIME Types)

Es gibt bereits hunderte von definierten Media Typen. Beispiele sind text/plain oder image/png. Einen bestehenden Typen zu wählen oder einen neuen Typen zu entwerfen ist eine wichtige Designentscheidung bei der Erstellung von http API’s. (Client und Server müssen es verstehen).

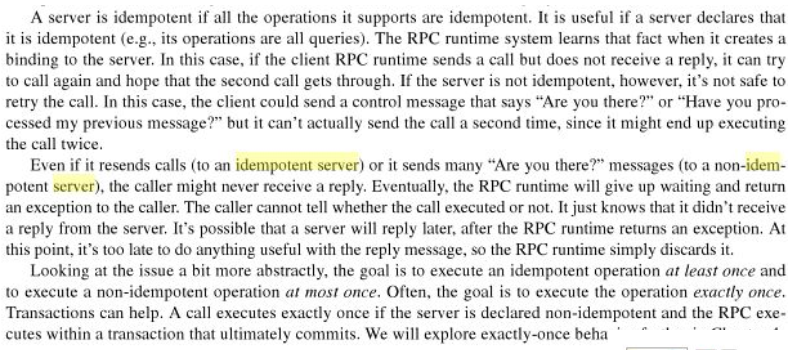
Netzwerkklassen im JDK  


## Architektonische und Designtechnische Aspekte für verteilte Software

* **Kommunikations Topologie**
  + Ein Client, ein Server? Mehrere Clients für einen Server? Dynamisches Setup?
* **Location Autonomy (Transparenz)**
  + Wirkliche Adressen vs. Virtuelle Adressen (Was wenn ein Server verschoben wird?)
* **Invocation and Message Semantics**
  + Bytestream vs. Document vs. Procedure vs. Remote Object
* **Timeout Management**
  + 1ms? 30 Sekunden? Unendlich?
* **Error Handling**
  + Erneute Requests
  + Server idempotent machen?

## Idempotent

Definition gemäss Internet: “*Idempotence is a funky word that often hooks people. Idempotence is sometimes a confusing concept, at least from the academic definition. From a RESTful service standpoint, for an operation (or service call) to be idempotent, clients can make that same call repeatedly while producing the same result. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same (e.g. a resource's state may change between requests). The PUT and DELETE methods are defined to be idempotent. However, there is a caveat on DELETE. The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls, unless the service is configured to "mark" resources for deletion without actually deleting them. However, when the service actually deletes the resource, the next call will not find the resource to delete it and return a 404. However, the state on the server is the same after each DELETE call, but the response is different. GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data. This makes them idempotent as well since multiple, identical requests will behave the same*.”



## Beispiel zur Wichtig von Time-Outs und Error Handling

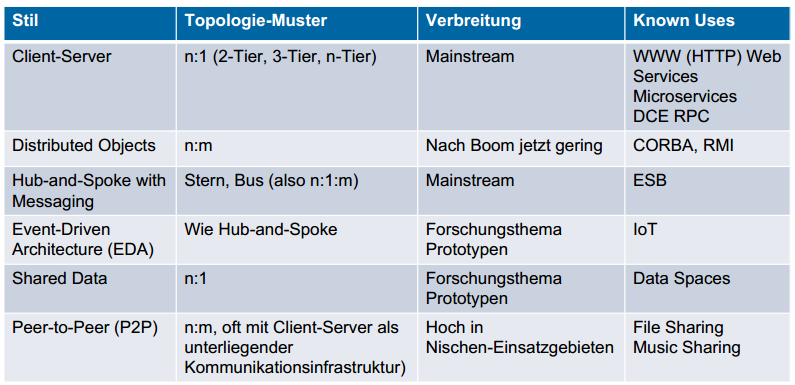
**Eine Exception, welche eine Airline gegrounded hat.**  
Eine JDBC Exception wurde auf einem Applikationsserver nicht gefangen (not caught). Dies führte dazu, dass alle Self Check-In Kioske der Airline offline gingen. Die Kioske waren dann offline für mehrere Stunden. Zur Normalität kam es erst als alle Applikations- und Datenbankserver neugestartet wurden.

**Hauptgründe, wieso es dazu kam**

* Unangemessenes Wiederholungs-/Failover-Management auf den Applikationsservern
* Kein Timeout Management auf den Applikationsservern
* Unvollständiges Monitoring der Applikationsserver (Sie wurden immer noch als Grün angezeigt, obwohl Sie geblockt waren).

# Netzwerkprogrammierung

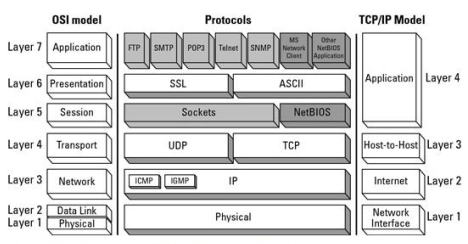
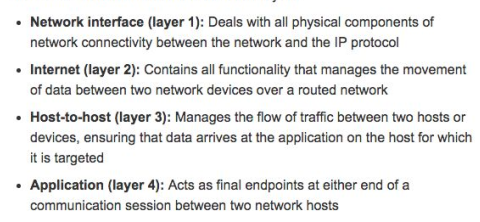
## Architekturstile im Vergleich



## Inter-Process Kommunikation Varianten

|  |  |  |
| --- | --- | --- |
| **Datenaustauch** Sockets (UDP oder TCP über IP), File Transfer, Shared Database | **Nachrichtenaustausch** Message-Oriented Middleware, http GET, POST | **Remote Procedure Call** DCE RPC, Java RMI, CORBA, Web Services, RESTFul HTTP |

## TCP/IP Sockets

Verbindung zwischen dem OSI und dem TCP/IP Model  
 

### Sockets im Überblick

Sockets sind ein Basis-Mechanismus für alle komplexeren Verfahren wie http oder RMI. Es dient zum Austausch von Bytestörmen auf der Programmierebene. Das Sockets ist eine Verbindung zwischen zwei Kommunikations-Endpunkten (IP + Port). Beim Konzept gibt es die zwei Rollen: Client und Server.

**Nachteile von Sockets**  
Es wird ein Konstruieren und Parsen der Byteströme erforderlich (keine Objekte und daher auch keine Typsicherheit). Zudem muss das Message Exchange Pattern (MEP) selbst spezifiziert, implementiert und überprüft werden. Das MEP definiert die Reihenfolge der send/receive Calls durch Client und Server.

### Generic TCP Application

**Server Sockets** Warten auf Request, welche für das Netzwerk kommen

**Client Sockets** Sendet und bekommt Daten, Implementiert mit der java.net.Socket-Klasse.

**Algorithmus für den TCP Client**Finden der IP und der Portnummer des Servers, Erstellen eines TCP Sockets, Verbinden des Sockets mit dem Server (Server muss laufen und auf Request hören), Senden/Empfangen von Daten durch das Socket, Verbindung schliessen.

Algorithmus für den TCP Server  
Ein TCP Server Socket erstellen, Binden des Server Sockets auf die Server IP und Port, Akzeptieren von neuen Verbindungen von den Clients, Senden / Empfangen von Daten über das Server Socket, Schliessen der Verbindung mit dem Client.

### Socket-Programmierung: Grundkonzepte

|  |  |
| --- | --- |
|  |  |

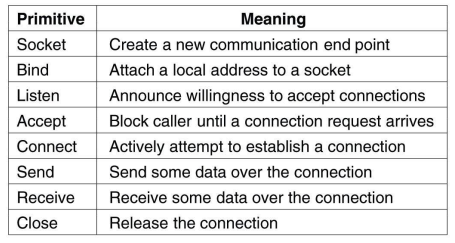
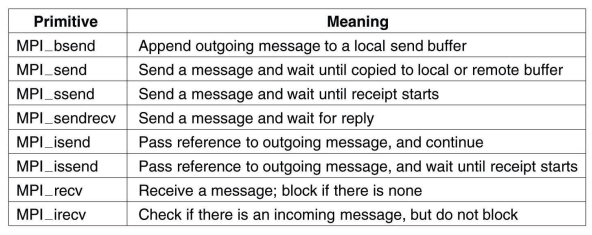
## Java Socket API

### Beispiel

|  |  |
| --- | --- |
| **Socket Server** Ein Socket (engl. Steckdose) ist ein Verbindungsendpunkt, der vom Programm wie eine gewöhnliche Datei beschrieben und gelesen werden kann. Der Serverprozess muss von außen eindeutig angesprochen werden können. Dazu bindet er sich an einen festen Port, den sogenannten “well known port”, über den er erreichbar ist | **Socket Client**  Der Client braucht keinen festen  Port. Er holt sich einen normalen Socket, dem vom System eine freie Nummer zugeteilt wird.  Der Server erfährt die Nummer des Clients aus der Anfrage und kann ihm unter diesem Port  Antworten. Im nächsten Schritt ruft der Client connect() auf, um eine Verbindung mit dem Server aufzunehmen, der in den Parametern beschrieben wird. Sobald die Verbindung da ist,  sendet der Client seine Anfrage per write() oder alternativ send() und wartet per read() oder recv() auf die Antwort des Servers. |

### Berkeley Sockets und das Message Passing Interface (MPI)

Gleiches Protokoll auf allen Platformen und in allen Sprachen. Der API Komfort ist je nach Sprache und Bibliothek unterschiedlich. Dabei müssen die System Resourcen gemanaged werden.

### IPv4 vs. IPv6

Es gibt einige Änderungen für die Socket-Programme (auch Java). Das Socket API und das primitive Interface bleiben das Gleiche. Die Adressierung hat geändert und daher auch die Formate der Antworten. Zudem unterscheidet sich das Quality-of-Service (Performance, Timeout Management).

### Vorteile und Nachteile von Java Sockets

**Vorteile**  
Flexibel, Mächtig, Generiert nur wenig Netzwerk Traffic (wenn es effizient genutzt wird).

**Nachteile**  
Können nur Raw Daten senden, Client und Server müssen einen Mechanismus haben um die Daten zu interpretieren, Zudem müssen beide die State Informationen halten.

## WebSockets

### Das Protokoll

Eine auf TCP basierendes Netzwerkprotokoll, das entworfen wurde, um eine bidirektionale Verbindung zwischen einer Webanwendung und einem WebSocket-Server bzw. einem Webserver herzustellen. Während bei einer reinen HTTP-Verbindung jede Aktion des Servers eine vorhergehende Anfrage des Clients erfordert, reicht es beim WebSocket-Protokoll, wenn der Client die Verbindung öffnet. Der Server kann dann diese offene Verbindung aktiv verwenden und kann neue Informationen an den Client ausliefern, ohne auf eine neue Verbindung des Clients zu warten. Bei WebSockets entfallen die durch den HTTP-Header verursachten zusätzlichen Daten, die bei jeder Anfrage einige Hundert Bytes umfassen können

### Client API und Support

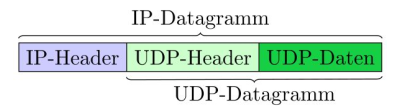
Web Sockets ist eine state-of-the-art Kommunikationtechnologie für Webapplikationen. Es arbeitet über Sockets. Ist sichtbar für ein JavaScript Interface oder kompatible HTML5 Browser. Sobald man eine WebSocket Verbindung hat, ist es möglich Daten mit send() an den Server zu senden.

### Server Support

* C++: libwebsockets
* Erlang: Shirasu.ws
* Java: Jetty (and others)
* Node.JS: ws
* Ruby: em-websocket
* Python: Tornado, pywebsocket
* PHP: Ratchet, phpws

Eine Demo zu Websockets gibt es unter <http://www.websocket.org/echo.html>

## UDP (User Datagram Protocol) 🡺 Vergleich mit TCP

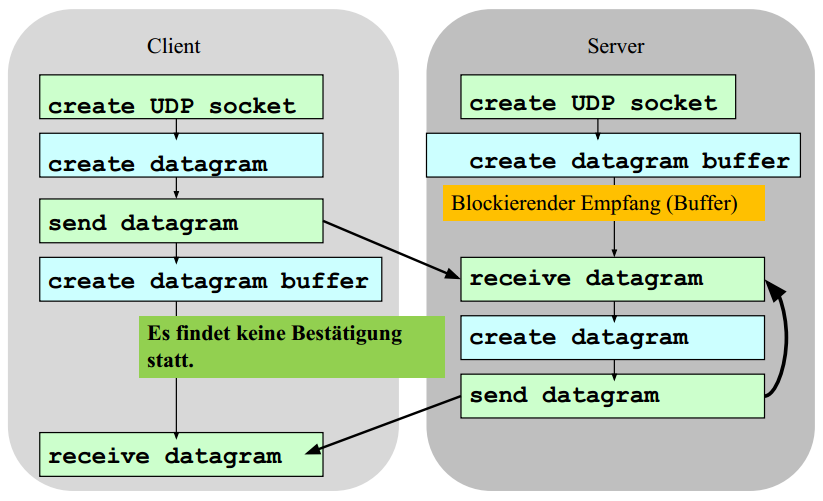


Verbindungsloses Netzwerkprotokoll für den Versand von Datagrammen in IP-basierten Netzen Die Entwicklung von UDP begann 1977, als man für die Übertragung von Sprache ein einfacheres Protokoll benötigte als das bisherige verbindungsorientierte TCP.

* Keine Verzögerungen bei der Sprachübertragung
* Dafür keine Garantie, dass ein Datagram ankommt
* Keine Garantie, dass Datagramme in der gleichen Reihenfolge ankommen, in der sie gesendet wurden
* Keine Garantie, dass ein Datagram nur einmal beim Empfänger eintrifft.

Eine Anwendung, die UDP nutzt, muss daher gegenüber verlorengegangenen und unsortierten Paketen unempfindlich sein oder selbst entsprechende Korrekturmaßnahmen und je nachdem auch Sicherungsmaßnahmen vorsehen.

### Struktur eines UDP Programms



### Beispiel - Sender und Empfänger

|  |  |
| --- | --- |
| **Sender** | **Empfänger** |

### UDP vs. TCP

Da mit UDP vor Übertragungsbeginn nicht erst eine Verbindung aufgebaut werden muss, kann der Datenaustausch schneller beginnen. *Das fällt vor allem bei Anwendungen ins Gewicht, bei denen nur kleine Datenmengen ausgetauscht werden müssen.*

Einfache Frage-Antwort-Protokolle wie DNS (Domain Name System) verwenden UDP, um die Netzwerkbelastung gering zu halten und damit den Datendurchsatz zu erhöhen. *Ein Drei-Wege-Handschlag wie bei TCP (dem Transmission Control Protocol) für den Aufbau der Verbindung würde unnötigen Overhead erzeugen.*

Daneben bietet die ungesicherte Übertragung auch den Vorteil von geringen Übertragungsverzögerungsschwankungen: Geht bei einer TCP-Verbindung ein Paket verloren, wird es automatisch neu angefordert. Das braucht Zeit, die Übertragungsdauer kann daher schwanken, was für Multimediaanwendungen schlecht ist. Bei VoIP z. B. käme es zu plötzlichen Aussetzern, bzw. die Wiedergabepuffer müssten größer angelegt werden. *Bei UDP bringen verlorengegangene Pakete nicht die gesamte Übertragung ins Stocken, sondern vermindern lediglich die Qualität.*

Bei Übertragungsfehlern oder bei Überlast löscht IP Pakete. *Datagramme können daher fehlen. UDP bietet hierfür keine Erkennungs- oder Korrekturmechanismen, wie etwa TCP.*

Im Falle von mehreren möglichen Routen zum Ziel kann IP bei Bedarf neue Wege wählen. Dadurch ist es in seltenen Fällen möglich, dass später gesendete Daten früher gesendete überholen. Außerdem kann ein einmal abgesendetes Datenpaket mehrmals beim Empfänger eintreffen.

### TCP/IP vs. UDP/IP

* Verbindungsorientiert vs. Nicht-verbindungsorientiert
* IP-Level Addressing und Routing via DNS, ARP, etc.
* Unterschiede in den Quality of Server (QoS)
  + Sequencing
  + How to deal with packet loss?
* Broadcast und Multicast möglich

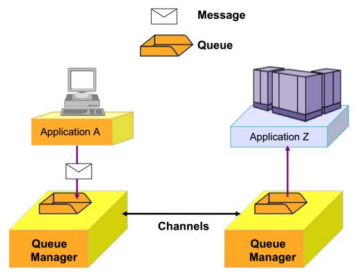
## Message Exchange Patterns

|  |  |
| --- | --- |
| **Basic Messaging Pattern** Applikationen kommunizieren indem sie Nachrichten senden über Message Channels. Die Applikation weiss über das Message Format, Channels, oder andere Details zur Verbindung. | **Blocking Receiver Message Pattern** Synchrone Nachrichtenendpunkte, welche solange blockieren bis eine Nachricht empfangen wird. |
| **Polling (Non-Blocking) Receiver Message Pattern** Polling Consumer, einer der explizit einen Aufruf macht, wenn er eine Nachricht empfangen möchte. | **Service Activator Message Pattern** Design und Implementation eines Service Activator, welcher die Nachrichten zu den Channels verbinden (die durch die Services genutzt warden). Dieser kann one-way oder Two-Way sein. |

# Asynchronous Messaging

## Einführung in Messaging

### Nachrichtenkonzept und die Komponenten

**Message Descriptor**  
Identifiziert die Nachricht und enthält Steuerinformationen (z. B. Nachrichtentyp und Priorität), die der Nachricht durch die sendende Anwendung zugewiesen wurde.

**Message data**  
Enthält die Anwendungsdaten. Die Struktur der Daten wird durch die Anwendungsprogramme definiert, die sie verwenden, und der Warteschlangenmanager ist weitgehend unbekümmert mit seinem Format oder Inhalt.

**Queue Manage**r  
Ist verantwortlich für das Akzeptieren und Zustellen der Nachrichten. Er verwaltet die Queues aller Nachrichten, welche warten. Warteschlangenmanager sind über ein Kommunikationsnetzwerk über logische Kanäle verbunden. Nachrichten fliessen automatisch über diese Kanäle vom Produzenten einer Nachricht an den Verbraucher dieser Nachricht, basierend auf der Konfiguration der Warteschlangenmanager in der Infrastruktur.

### Message-Queuing Model und RPC’s

|  |  |  |
| --- | --- | --- |
| **Loose Coupling / Asynchronous Processing** | **Synchronous RPC** | **Asynchronous RPC** |

### Nachrichten nutzen um Applikationen zu verbinden und Daten zu verteilen

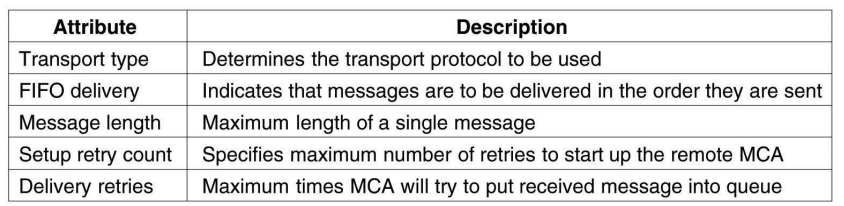
|  |  |
| --- | --- |
|  |  |

Asynchrone Verarbeitung kann über verschiedene Wege genutzt werden. Warteschlange große Mengen an Arbeit, die dann einmal pro Tag durch einen Batch-Prozess eingereicht wird. Die Arbeit kann auch sofort eingereicht und dann verarbeitet werden, wenn die empfangende Bewerbung den Umgang mit einer vorherigen Anfrage beendet hat. Asynchrone Verarbeitung impliziert nicht unbedingt alle Antwortzeiten.

|  |  |
| --- | --- |
| Message-Oriented-Middleware (MOM) | Hub-and-Spoke Architecture with CMB |

## Java Message Service (JMS) API

Point-to-Point Channel über die Queue API, Publish-Subscribe Channel über die Topic API.



### Reliability levels

**Best effort nonpersistent**  
Nachrichten werden verworfen, wenn eine Messaging-Engine stoppt oder fehlschlägt. Nachrichten können auch verworfen werden, wenn eine Verbindung, die verwendet wird, um sie zu senden, nicht verfügbar ist oder als Ergebnis von eingeschränkten Systemressourcen.

**Express nonpersistent**Nachrichten werden verworfen, wenn eine Messaging-Engine stoppt oder fehlschlägt. Nachrichten können auch verworfen werden, wenn eine Verbindung, die verwendet wird, um sie zu senden, nicht verfügbar ist.

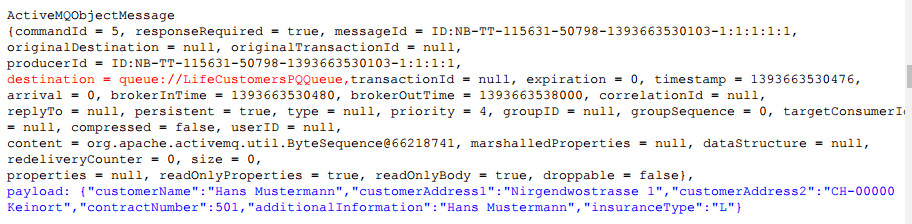
**Reliable nonpersistent**  
Nachrichten werden verworfen, wenn eine Messaging-Engine stoppt oder fehlschlägt.

**Reliable persistent**  
Nachrichten können verworfen werden, wenn eine Messaging-Engine ausfällt.

**Assured persistent**  
Nachrichten werden nicht verworfen.

### JMS Message-Struktur

Eine JMS-Message besteht aus a) Header, b) Properties und c) Body. Ein Header wird immer gesendet. Er enthält Informationen für das Routing und die Identifikation. Ein MOM-Provider normiert seine Messsage Header. Properties sind optional – sie enthalten zum Beispiel Informationen, mit deren Hilfe ein Coustomer Nachrichten filtern oder weiterrouten kann (Erweiterung des Standard-Headers). Der Body ist auch zwingend erforderlich – er enthält die auszutauschenden Nutzdaten (Text, Objekte, Binärdaten).



### Beispiel für einen JMS Message Provider

|  |  |
| --- | --- |
| **Queue Sender** | **Queue Receiver** |

### Andere APIs oder Provider

* Provider, welche die Java Messaging Service (JMS) Spezifikation beinhalten
  + Apache ActiveMQ
  + Any Java Enterprise Edition (JEE) Application Server: IBM, JBoss, Oracle
* RabbitMQ (eigene API, nicht kompatibel zu JMS)
  + Open Source, Implementiert das AMQP Protokoll.
* Cloud Provider z.B. Amazon Simple Queueing Server (SQS)
* Apache Camel oder Sprint Integration
* ZeroMQ
  + Alternativer Weg zum Queue basierten Nachrichten.

## Messaging Middleware Provider

### IBM MQ

Ermöglicht es, Anwendungen asynchron zu kommunizieren, d.h. Nachrichten können zu unterschiedlichen Zeiten gesendet, empfangen und verarbeitet werden. Kann jede Art von Daten als Nachrichten transportieren. Arbeitet mit einer breiten Palette von Computing-Plattformen, Anwendungen, Web-Services und Kommunikationsprotokolle.

#### Two-Phase Commit (2PC)

**Erste Phase**  
Der Koordinator sendet ein prepare an alle Teilnehmer. Die Teilnehmer verarbeiten die Transaktion bis zu dem Punkt, wo die Transaktion entweder mit commit oder rollback abgeschlossen wird. Dabei schreiben sie Einträge in ihr undo log und in ihr redo log. Die Teilnehmer antworten mit ready, wenn die Transaktion erfolgreich war - oder sie antworten mit failed, wenn die Transaktion fehlgeschlagen ist.

**Zweite Phase**  
Wenn der Koordinator von allen Teilnehmern eine ready Meldung bekommen hat: Der Koordinator sendet commit an alle Teilnehmer. Die Teilnehmer schließen die Transaktion mit commit ab und geben alle Locks und Ressourcen frei. Die Teilnehmer senden ein acknowledgment zurück. Der Koordinator beendet die Transaktion, wenn er von allen Teilnehmern die Bestätigung erhalten hat.

Wenn zumindest einer der Teilnehmer ein failed schickt: Der Koordinator sendet abort an alle Teilnehmer. Die Teilnehmer schließen die Transaktion mit rollback ab (mittels des undo logs) und geben alle Locks und Ressourcen frei. Die Teilnehmer senden ein acknowledgment zurück. Der Koordinator beendet die Transaktion ebenso mit rollback, wenn er von allen Teilnehmern die Bestätigung erhalten hat.

#### Persistente und nicht persistente Nachrichten

Meldungen, die kritische Geschäftsdaten enthalten, wie z. B. Zahlungseingang für eine Bestellung, sollten zuverlässig aufrechterhalten werden und dürfen im Falle eines Fehlers nicht verloren gehen. Einige Nachrichten können nur Abfragedaten enthalten, wobei der Verlust der Daten nicht entscheidend ist, da die Abfrage wiederholt werden kann. In diesem Fall gilt die Leistung als wichtiger als die Datenintegrität.

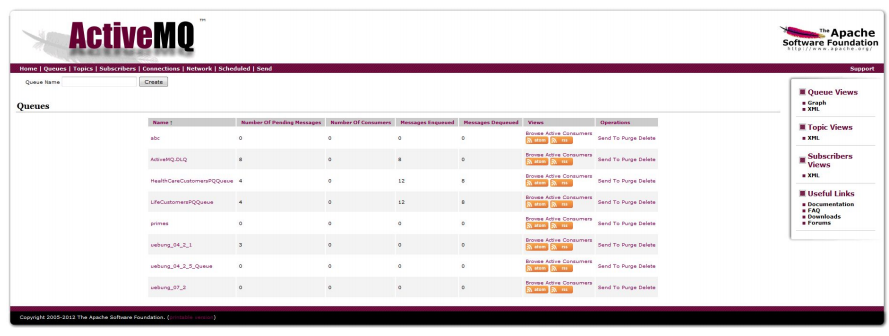
**Persistente Nachrichten**  
IBM MQ verliert keine anhaltende Nachricht durch Netzwerkfehler, Lieferfehler oder Neustart des Warteschlangenmanagers. Jeder Warteschlangenmanager hält ein fehlertolerantes Wiederherstellungsprotokoll aller Aktionen, die bei persistenten Nachrichten ausgeführt werden. Dieses Protokoll wird manchmal als Zeitschrift bezeichnet

**Nicht-persistierende Nachrichten**  
IBM MQ optimiert die Aktionen, die auf nicht persistenten Nachrichten für die Leistung durchgeführt werden. Die nicht-persistente Nachrichtenspeicherung basiert im Systemspeicher, so dass es möglich ist, dass sie in Situationen wie Netzwerkfehlern, Betriebssystemfehlern, Hardwarefehlern, Warteschlangenmanager-Neustart und internem Softwarefehler verloren gehen können.

### MS MQ

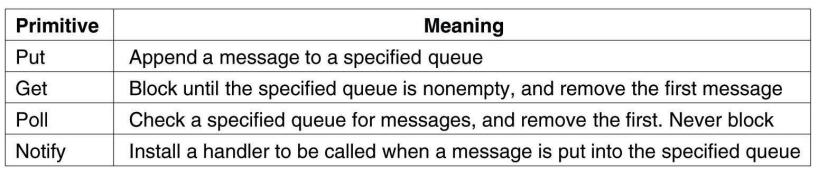
Message Queuing (MSMQ) -Technologie ermöglicht es Anwendungen, die zu unterschiedlichen Zeiten laufen, um über heterogene Netzwerke und Systeme zu kommunizieren, die vorübergehend offline sein können. Anwendungen senden Nachrichten an Warteschlangen und lesen Nachrichten aus Warteschlangen. Die folgende Abbildung zeigt, wie eine Warteschlange Meldungen enthalten kann, die von mehreren Sendeanwendungen erzeugt und von mehreren empfangenden Anwendungen gelesen werden. Es ist Bestandteil von Windows aber nicht installiert/aktiviert im Default Setup.

### Active MQ



### MOM API Primitives (Platform-Independet Level)

Write (Put), Consuming Read/Receive (Get) – Synch and asynchronous (event-or callback-based), Non-Consuming Read/Receive (Browse) – Like top() in stack.



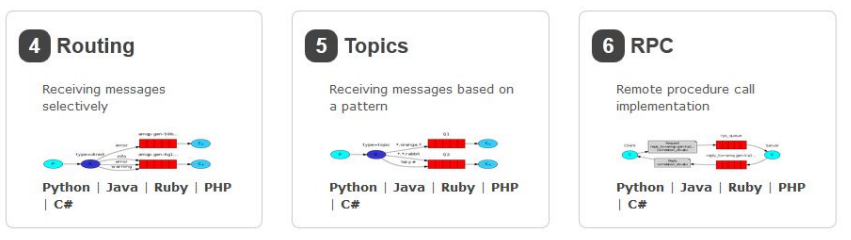
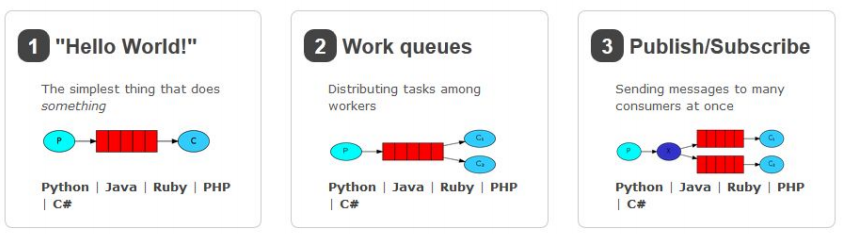
## Enterprise Integration Patterns

Siehe Vorlesungfolien.

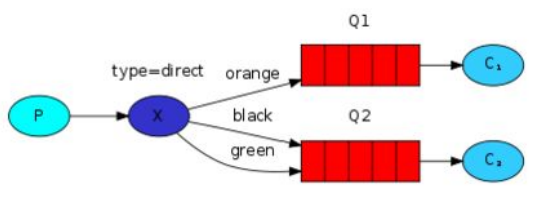
## RabbitMQ (based on AMQP)

### Patterns

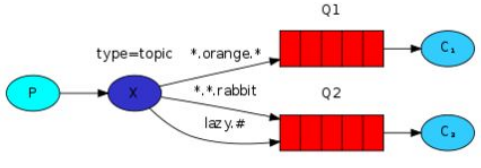
Basierend auf dem AMQP (Advanced Message Queueuing Protocol)



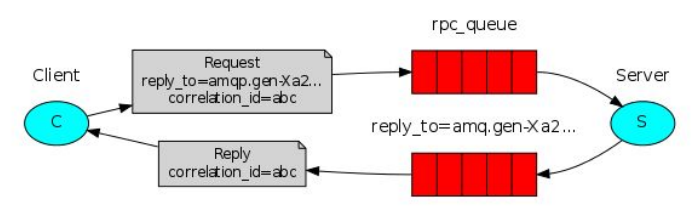
#### Routing

Beispiel von einem direkten Austausch mit zwei gebundenen Warteschlangen. Die erste Warteschlange ist mit dem Key «Orange» gebunden, die zweite Warteschlange hat zwei Bindings von schwarz und green. Die Zustellung erfolgt entsprechend den Bindings. Alle anderen Nachrichten werden verworfen.

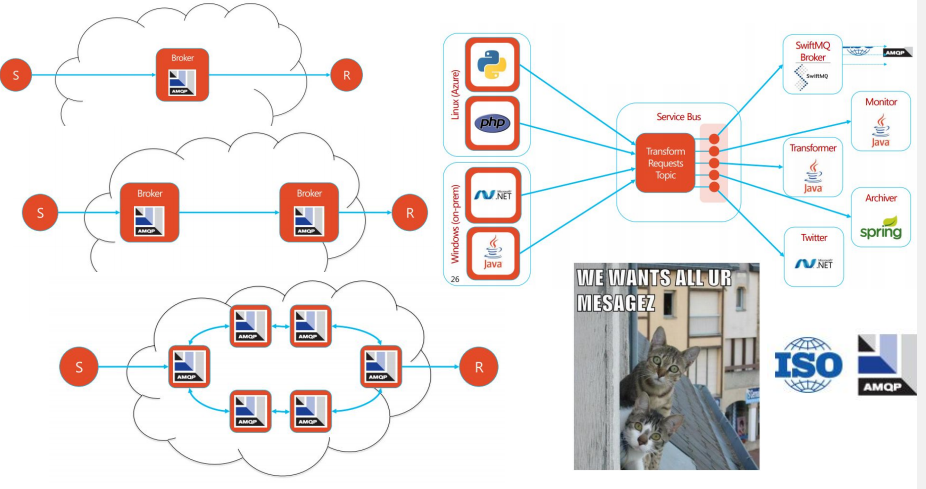
#### Topics

In this example, messages describe animals. Messages have a routing key that consists of three words .. Q1 is bound "\*.orange.\*" and Q2 with "\*.\*.rabbit" and "lazy.#". Q1 is interested in all the orange animals. Q2 wants to hear everything about rabbits, and everything about lazy animals. Messages with "quick.orange.rabbit" will be delivered to both queues. Messages "lazy.orange.elephant" also will go to both of them. "quick.orange.fox" will only go to the first queue, and "lazy.brown.fox" only to the second. "lazy.pink.rabbit" will be delivered to the second queue only once, even though it matches two bindings. "quick.brown.fox" doesn't match any binding so it will be discarded. 

#### RPC

Wenn der Client startet, erstellt er eine anonyme, exklusive Callback-Warteschlange. Für eine RPC-Anforderung sendet der Client eine Nachricht mit zwei Eigenschaften: reply\_to, die auf die Callback-Warteschlange und correlation\_id gesetzt ist, die für jede Anfrage auf einen eindeutigen Wert gesetzt wird. Die Anforderung wird an eine rpc\_queue-Warteschlange gesendet. Der RPC-Arbeiter (aka: server) wartet auf Anfragen auf diese Warteschlange. Wenn eine Anforderung erscheint, führt sie den Job aus und sendet eine Nachricht mit dem Ergebnis an den Client zurück, wobei die Warteschlange aus dem Feld reply\_to verwendet wird. Der Client wartet auf Daten in der Rückrufwarteschlange. Wenn eine Meldung erscheint, prüft sie die Eigenschaft cororrect\_id. Wenn es dem Wert aus der Anfrage entspricht, gibt er die Antwort an die Anwendung zurück.

#### AMQP Adanved Message Queueing Protocol



# Vorlesung Woche 4