Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering
InIT Institut für angewandte
Informationstechnologie

Bachelor of Science (BSc) in Informatik

Modul Software-Entwicklung 1 (SWEN1)

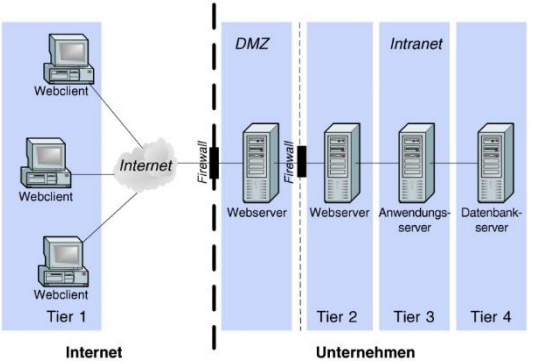
V1 – Verteilte Systeme

SWEN1/PM3 Team:
R. Ferri (feit), D. Liebhart (lieh), K. Bleisch (bles), G. Wyder (wydg)

Ausgabe: HS24

Um was geht es?

- Was sind verteilte Systeme?
- Wie ist der prinzipielle Aufbau eines Client-Server-Systems?
- Welche Phänomene und Probleme ergeben sich bei verteilten Systemen?
- Welche Aspekte sind zu berücksichtigen beim Design und der Implementierung eines Client-Server-Systems?
- Was sind gängige Technologien (Middleware) zur Entwicklung von verteilten Systemen?



Lernziele VT 01 – Verteilte Systeme

Sie sind in der Lage,

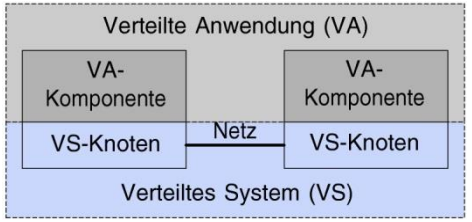
- zu erläutern, was ein verteiltes System ist und warum verteilte Systeme eingesetzt werden,
- die fundamentalen Konzepte eines verteilten Systems wie Architekturstil, Kommunikationsverfahren, Fehlertoleranz und Fehlersemantik zu erläutern,
- wichtige Design- und Implementierungsaspekte von Client-Server-Systemen zu diskutieren,
- für einen Entwurf eines verteilten Systems gängige Architektur und Design Patterns zu benutzen,
- gängige Technologien (Middleware) zur Entwicklung von verteilten betrieblichen Informationssystemen und Internet-basierten Systemen einzuordnen.

Agenda

1. Einführung in verteilte Systeme
2. Design- und Implementierungskonzepte von Client-Server-Systemen
3. Middleware für verteilte Systeme
4. Wrap-up und Ausblick

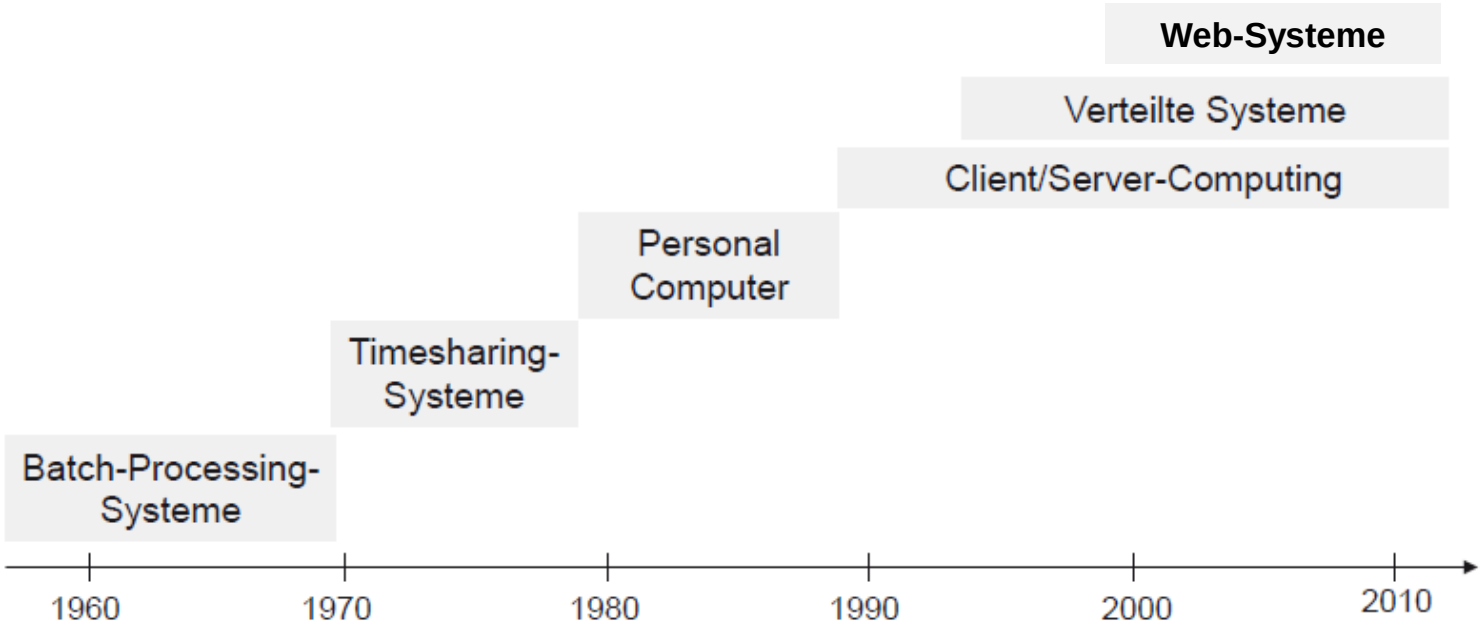
Was ist ein verteiltes System?

- Verteiltes System
  - Basiert auf einer Menge voneinander unabhängiger Rechnersysteme (Knoten) und Softwarebausteinen (Komponenten).
  - Erscheinen dem Benutzer wie ein einzelnes, kohärentes System bzw. Anwendungssystem.



- Verteilte Anwendung
  - Anwendung, die auf einem verteilten System läuft.
  - Jeder Softwarebaustein kann auf einem eigenen Rechner liegen.
  - Es können aber auch mehrere Softwarebausteine auf dem gleichen Rechner installiert sein.

Historische Entwicklung

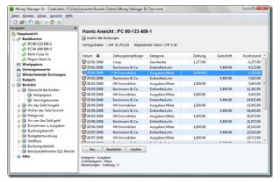


Die folgenden Faktoren haben die Entwicklung wesentlich beeinflusst:

- Leistungsexplosion in der Mikroprozessortechnik,
- schnelle lokale Netzwerke (LAN)
- Verbindung mehrerer physischer Netze zu einem einheitlichen Kommunikationssystem (WAN) und das Anbieten eines Universaldienstes für heterogene Netzwerke, dem Internet

Typische verteilte Systeme heute sind…

- Informationssysteme
- Mobile Systeme
- Eingebettete Systeme
- Cloudbasierte Systeme
- Hochleistungsrechnersysteme

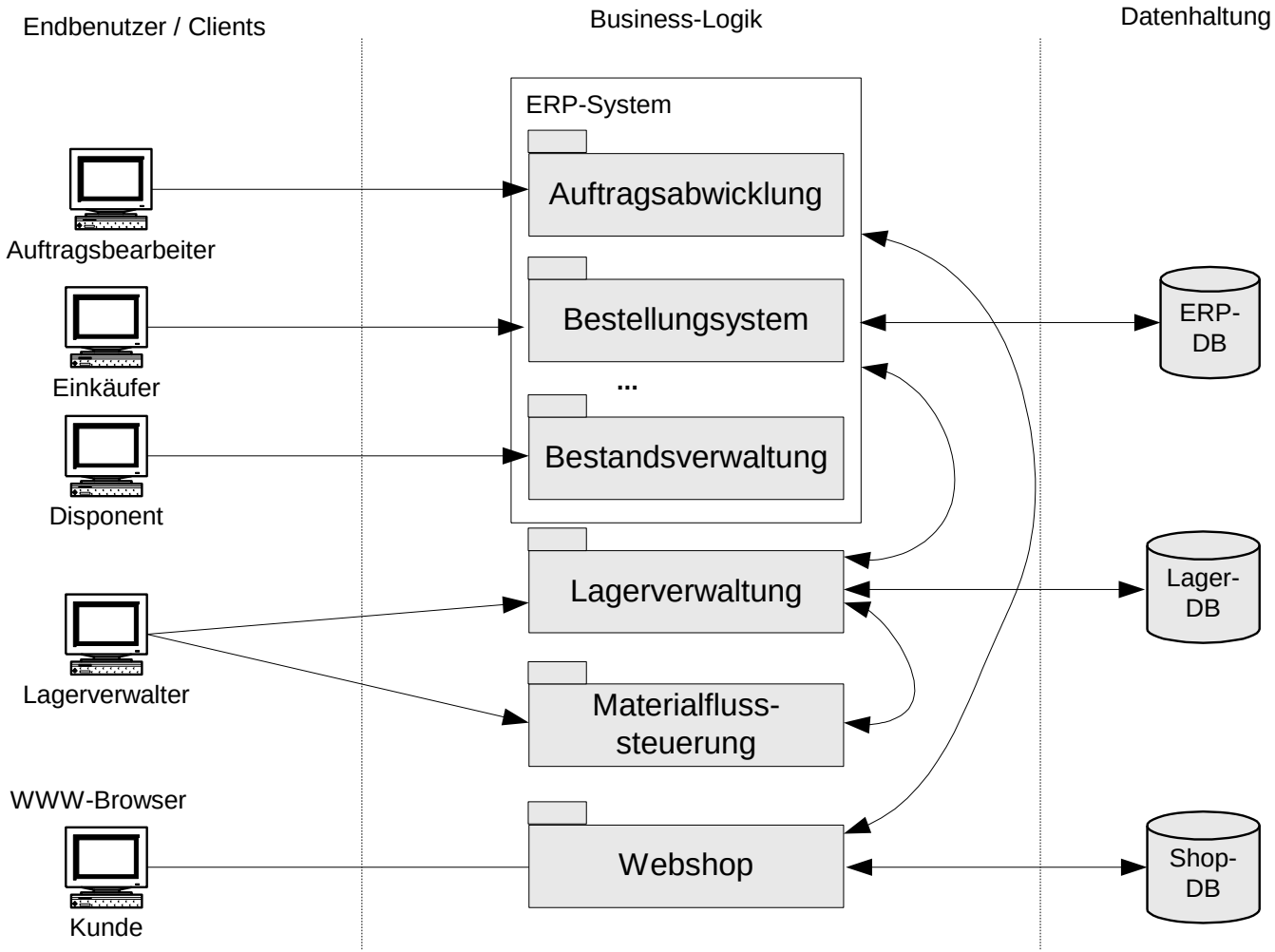





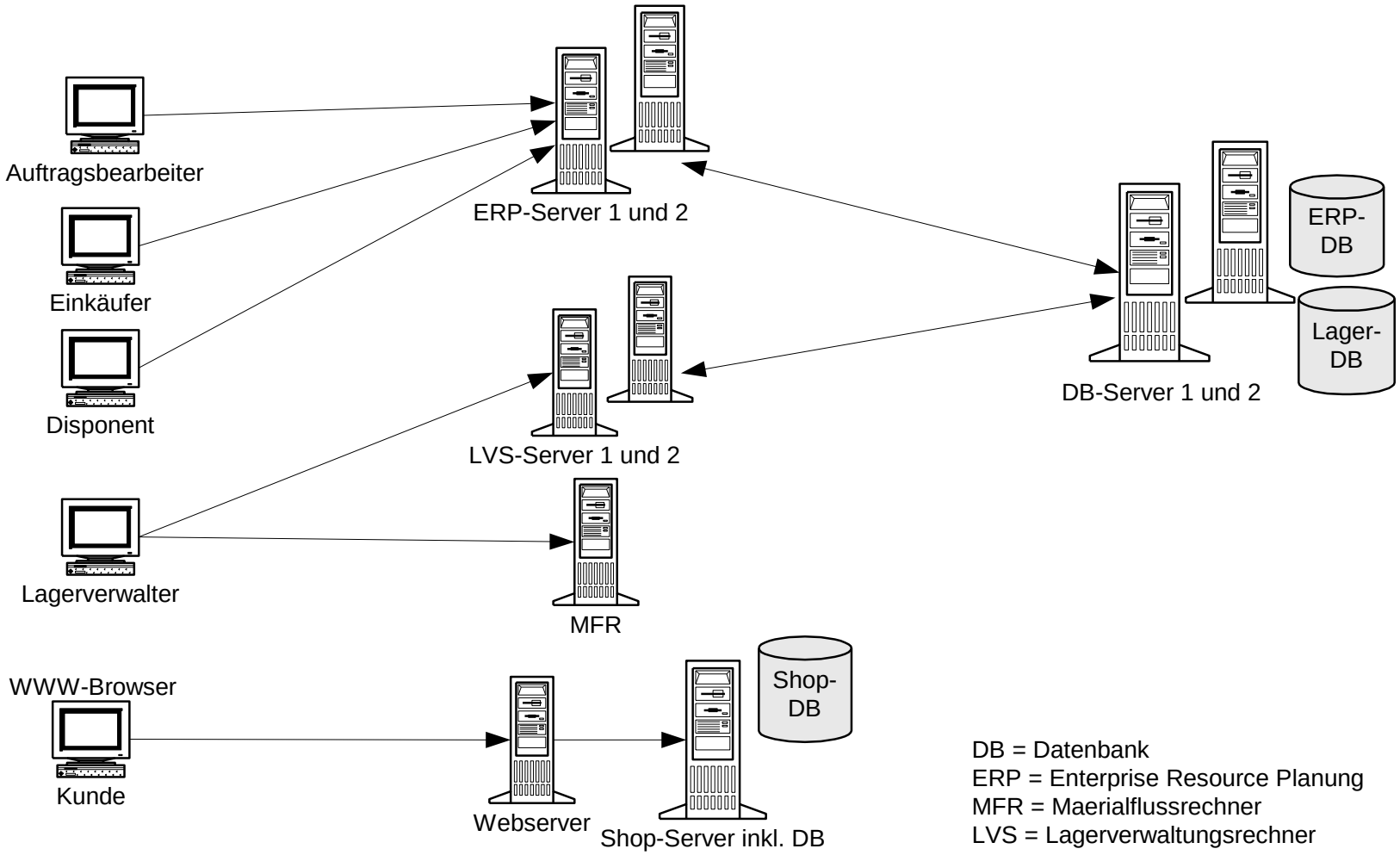
Was sind verteilte Informationssysteme?

- Verteilte Informationssysteme sind verteilte Systeme mit besonderen Merkmalen.
- Typische Merkmale:
  - Oft sehr gross
  - Sehr datenorientiert: Datenbanken im Zentrum der Anwendung
  - Extrem interaktiv: GUI, aber auch Batch
  - Sehr nebenläufig: Grosse Anzahl an parallel arbeitenden Benutzern
  - Oft hohe Konsistenzanforderungen

Klassische verteilte Informationssysteme im E-Commerce (logische Sicht)



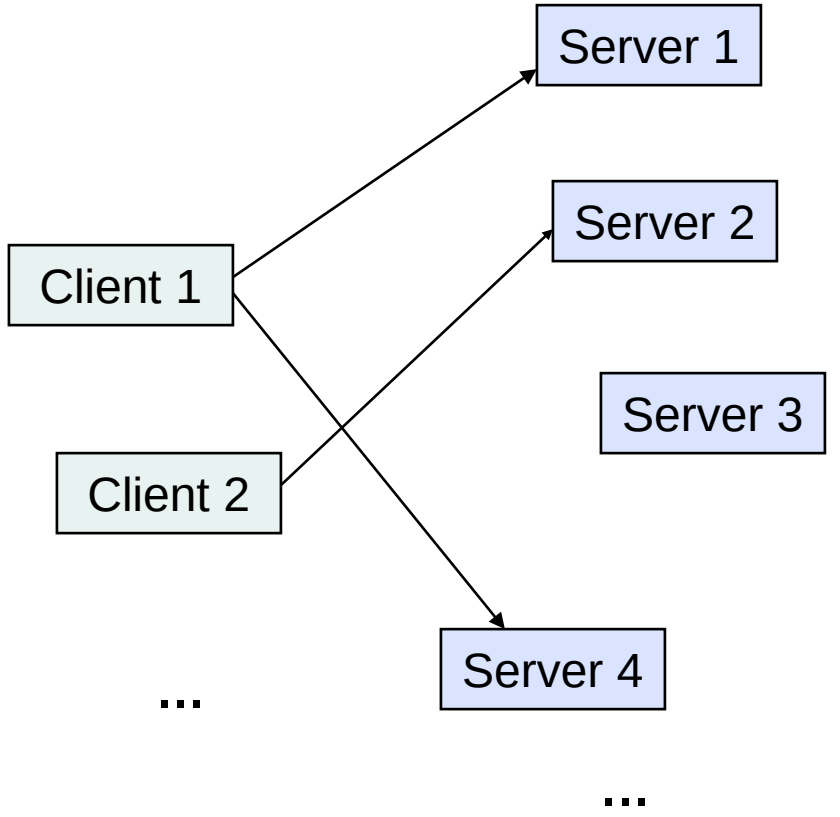
Klassische verteilte Informationssysteme im E-Commerce (physische Sicht)



DB = Datenbank
ERP = Enterprise Resource Planung
MFR = Maerialflussrechner
LVS = Lagerverwaltungsrechner

Warum setzt man auf verteilte Systeme?

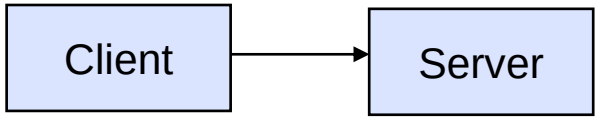
- Vorteile:
  - Gemeinsamer Ressourcenzugriff
  - Lastverteilung
  - Ausfallsicherheit, Verfügbarkeit
  - Skalierbarkeit
  - Flexibilität
  - Verteilungstransparenz (Ort, Fehler, Persistenz, ...)
- Nachteile:
  - Komplexität durch Verteilung, Netzinfrastruktur
  - Sicherheitsrisiken



Architekturmodelle verteilter Systeme

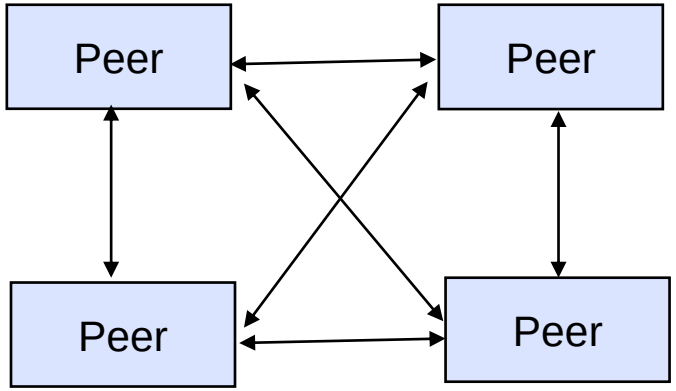
- Ein Architekturmodell beschreibt die Rollen der Komponenten innerhalb einer verteilten Anwendung sowie die Beziehungen zwischen ihnen.
- Heute finden vor allem folgende Architekturmodelle ihren Einsatz:

Client/Server



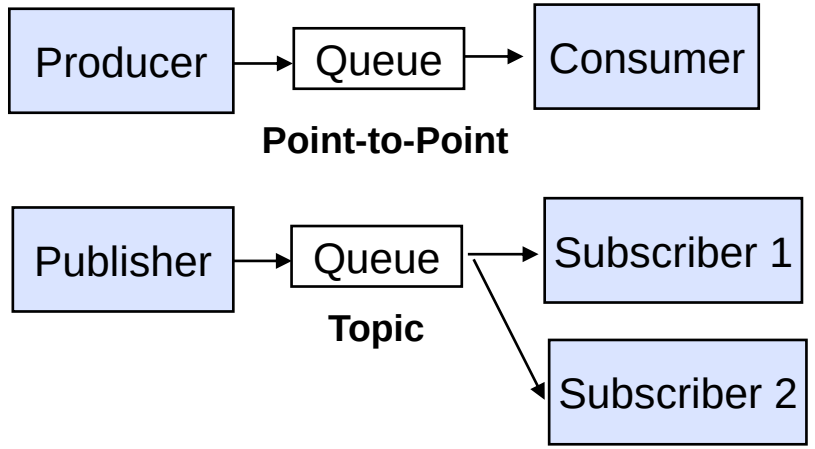
Kurzlebiger Client-Prozess, der mit einem langlebigen Server-Prozess kommuniziert (z.B. Web-Applikation)

Peer-to-Peer



Gleichberechtigte Peer-Prozesse, die nur bei Bedarf Informationen austauschen (z.B. Blockchain)

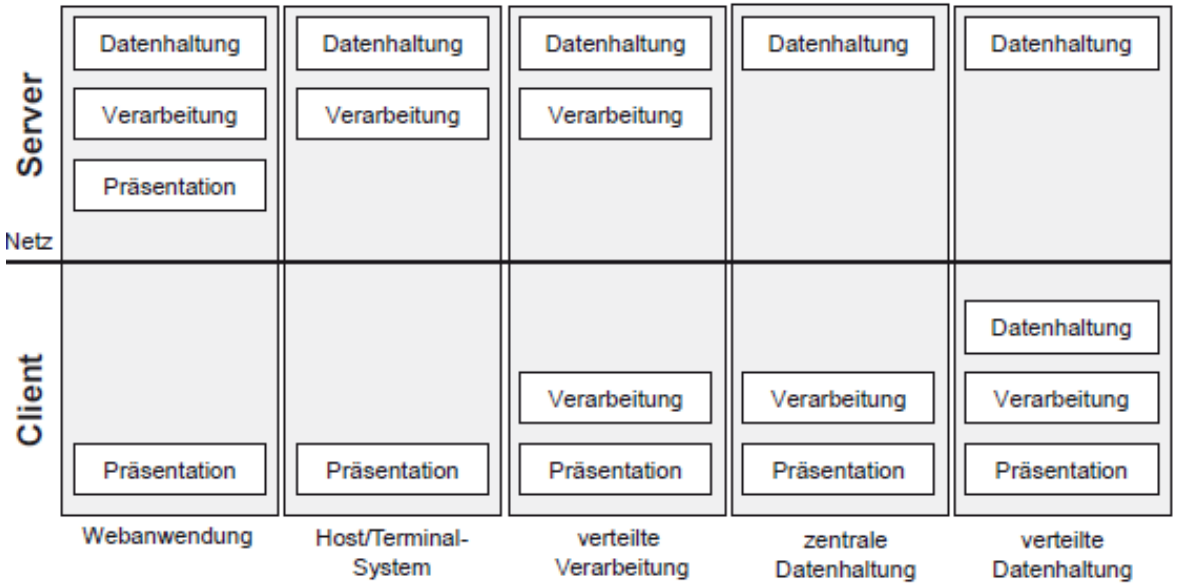
Event Systems



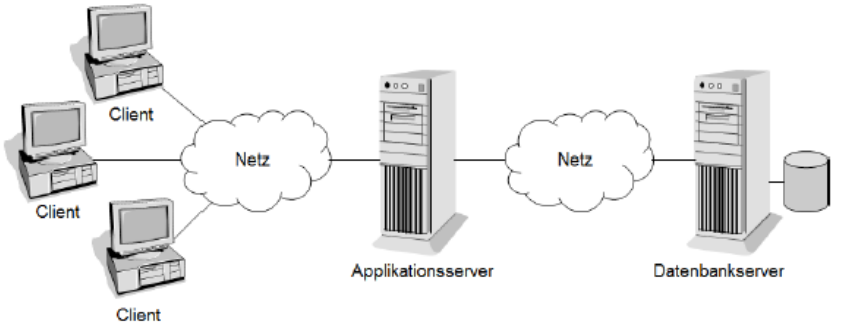
Event-Sources-Prozesse und Event-Sinks-Prozesse, die asynchron Informationen austauschen (z.B. E-Mail)

Mehrstufige Architekturen (Multi-Tier-Architekturen)

- Multi-Tier-Architekturen sind eine Ergänzung zum Client-Server-Architekturmodell und beschreiben Modelle zur Verteilung einer Anwendung auf den Rechnern (engl. tiers) des verteilten Systems.
- Für die Arbeitsteilung zwischen Client und Server existieren verschiedene Alternativen, je nachdem, wo die Schichten (Layer) Präsentation, Verarbeitung (Domänenlogik) und Datenhaltung angesiedelt sind.



Beispiel: 3-Tier-Architektur



Denkpause

Aufgabe 10.1 (5’)

Diskutieren Sie in Murmelgruppen die acht Irrtürmer bzw. falschen Annahmen der verteilten Datenverarbeitung und was heute immer noch fatale, falsche Annahmen sind (engl. Fallacies of Distributed Computing, Deutsch/Gosling, 1994/97).

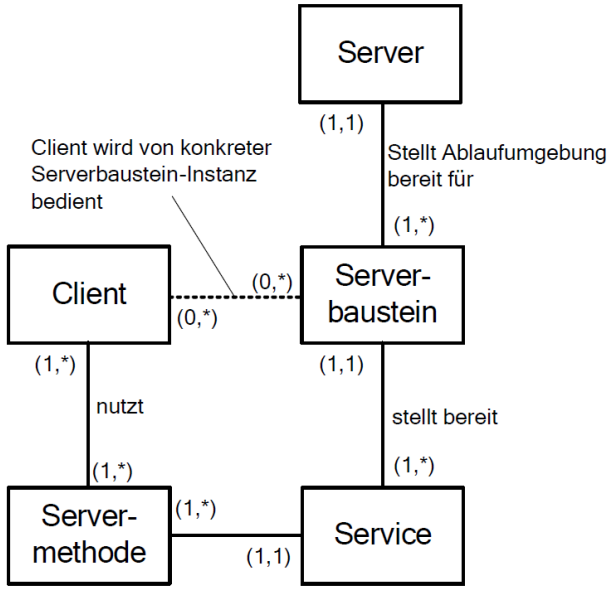
- Das Netzwerk ist ausfallsicher.
- Die Latenzzeit ist gleich Null.
- Der Datendurchsatz ist unendlich.
- Das Netzwerk ist sicher.
- Die Netzwerktopologie wird sich nicht ändern.
- Es gibt immer nur einen Netzwerkadministrator.
- Die Kosten des Datentransports können mit Null angesetzt werden.
- Das Netzwerk ist homogen.

Agenda

1. Einführung in verteilte Systeme
2. **Design- und Implementierungskonzepte von Client-Server-Systemen**
3. Middleware für verteilte Systeme
4. Wrap-up und Ausblick

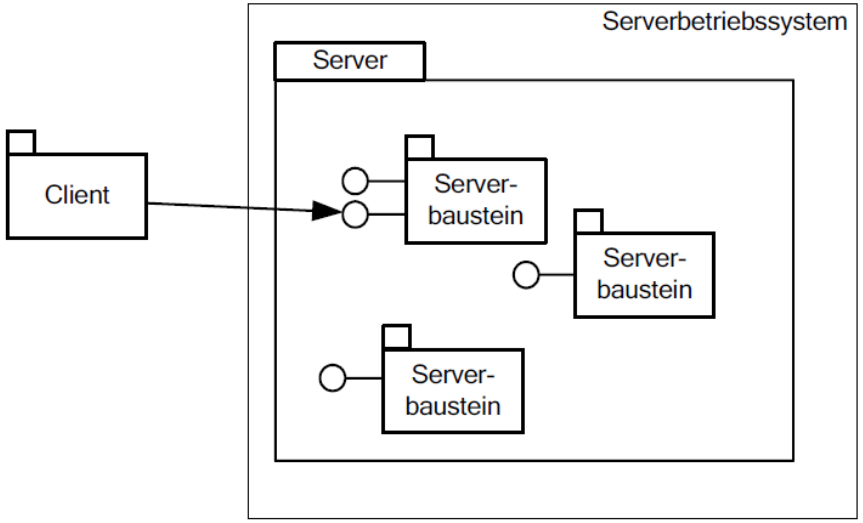
Terminologie bzw. Metamodell für die Diskussion von Client-Server-Systemen

- Ein Server stellt eine Ablaufumgebung für einen oder mehrere Serverbausteine bereit.
- Ein Applikationsserver ist auch ein Server, auf dem Serverbausteine ausgeführt werden, aber im engeren Sinne noch verschiedene Dienste den Serverbausteinen anbietet (z.B. Authentifizierung, Transaktionen etc.).
- Ein Serverbaustein ist ein Objekt, Modul oder Komponente (je nach verwendetem Programmiermodell), das zum Ablaufzeitpunkt instanziiert und bei Bedarf einem Client für die Abarbeitung einer Anforderung (eines Requests) zugeordnet wird.
- Ein Service oder Dienst wird durch einen Serverbaustein bereitgestellt und enthält eine oder mehrere Servermethoden oder Serverprozeduren.
- Eine Servermethode oder eine Serverprozedur ist Bestandteil eines Services, den ein Client durch das Senden eines entsprechenden Requests nutzen kann.



Server, Serverbausteine und Servermethoden

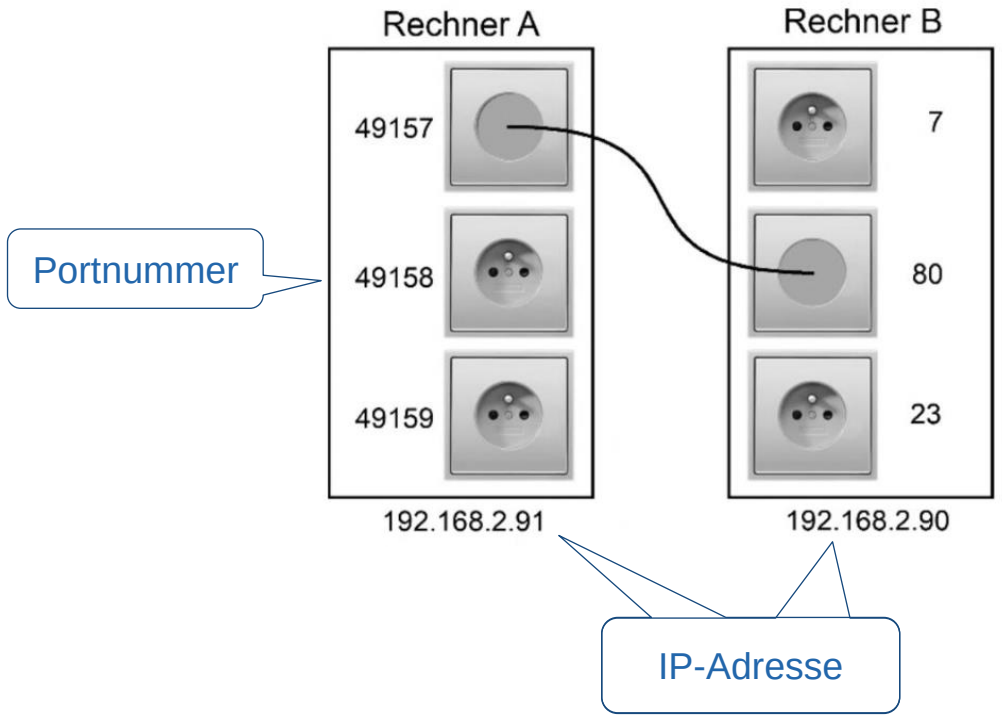
- Ein Server empfängt die Client-Requests und leitet diese zur Verarbeitung an den entsprechenden Serverbaustein.
- Ein Server ist seinerseits in eine Ablaufumgebung innerhalb eines Betriebssystems eingebettet.



- Man kann auch noch zwischen einem Server bzw. Serverbaustein und seiner konkreten Instanzierung unterscheiden.

Kommunikation zwischen Client und Server (1/2)

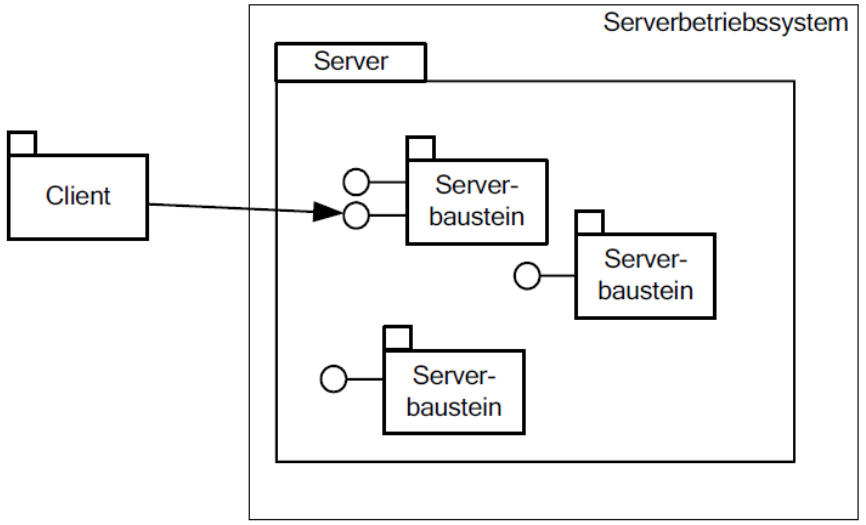
- Jeder Service ist über seine URL aufrufbar:
  - protokoll://<server>:<port>/<pfad_des_service>
- Kommunikation zwischen Client und Server
  - Über TCP oder UDP
  - Socket
    - Programmierschnittstelle zu Kommunikationskanal
    - IP-Socket-Adresse: IP-Adresse + Portnummer



Kommunikation zwischen Client und Server (2/2)

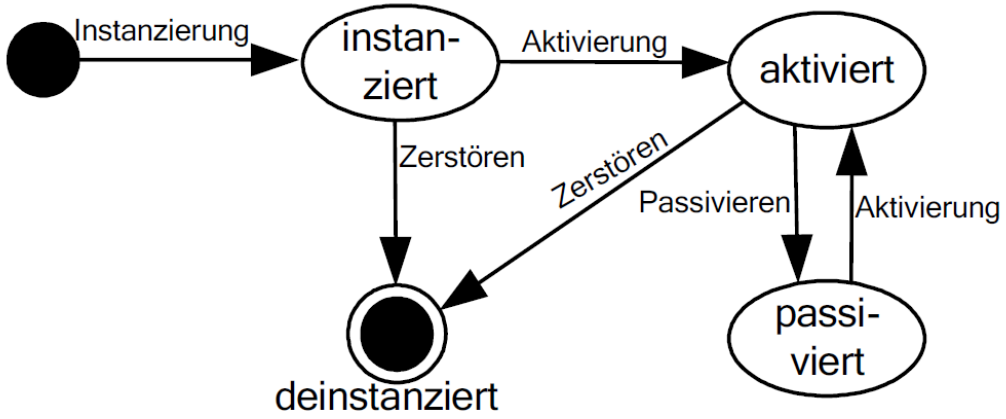
- Client sendet Request an Server
- Ein Server empfängt den Client-Request und leitet diesen zur Verarbeitung an den entsprechenden Service (des betreffenden Serverbausteins) weiter.
- Service bearbeitet Request und schickt Antwort (Response) zurück an den Client.

- Ein Server ist seinerseits in eine Ablaufumgebung (z.B. VM) innerhalb des Rechnerbetriebssystems eingebettet.
- Server und Serverbaustein müssen vor der Verwendung instanziiert werden.



Lebenszyklus von Serverbausteinen

- Ein Serverbaustein wird zur Laufzeit von einem Server instanziiert und durchläuft, je nach Bausteintyp und Implementierung verschiedene Zustände.
- Zustandsdiagramm für eine Serverbaustein-Instanz:

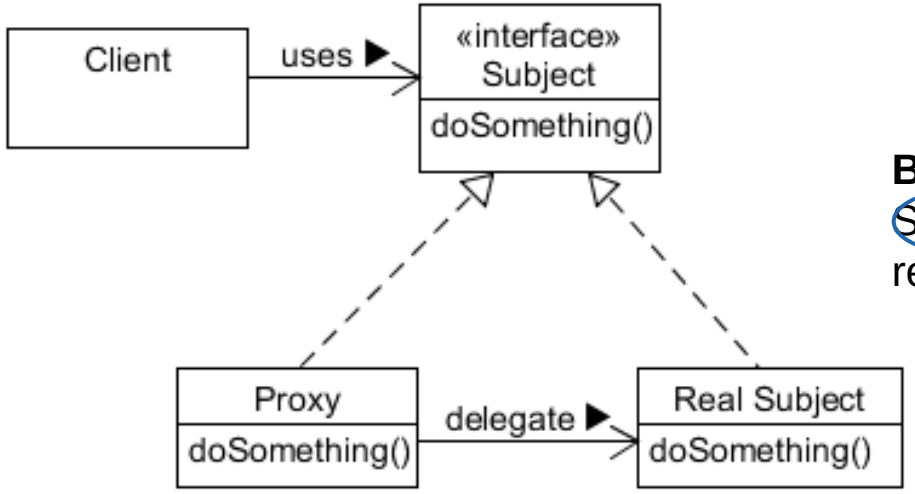


- Die Anzahl und Benennung der Zustände sind je nach Client-Server-Implementierung (Middleware) verschieden.

Verwendetes Design Pattern für den Zugriff auf Services in Serverbausteinen

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

- Grundlegendes Design Pattern für den Zugriff auf Serverbausteine ist das Remote Proxy.



Interface

Proxy:
Instanziierung in der Regel durch Dependency Injection (DI)

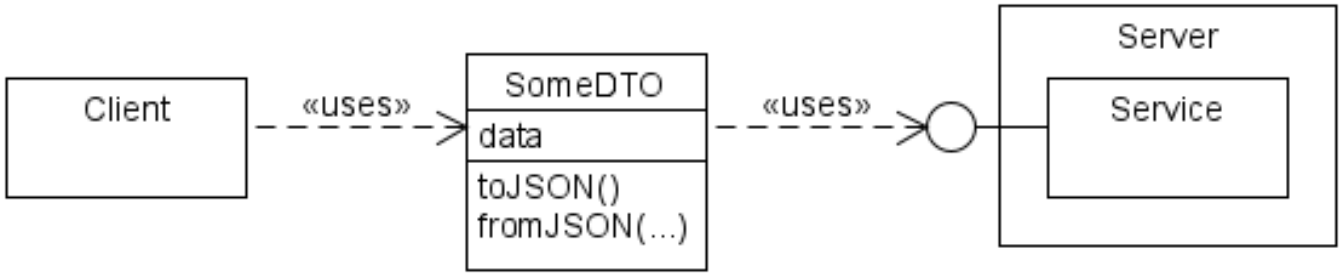
Beispiel:

```
Service service = new ServiceImpl();
result = service.doSomething();
```

Anmerkung: *In einer Client-Server-Implementierung heisst der (client- und serverseitige) Proxy auch Stub (von englisch stub, Stubben, Stummel, Stumpf). Ein server-seitig generierter Stub wird dabei Skeleton (engl. Skelett, Gerippe, Gerüst) genannt.*

Verwendetes Design Pattern für den Datenaustausch zwischen Client und Server

- Grundlegendes Design Pattern ist das Data Transfer Object (DTO). [3]



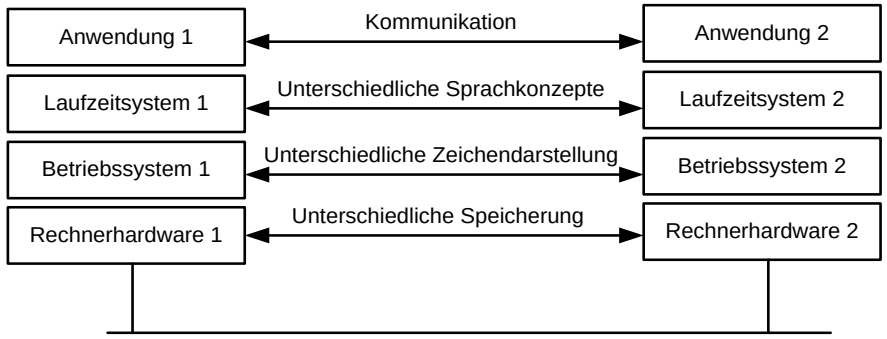
- Es bündelt mehrere Daten in einem Objekt, sodass sie durch einen einzigen Programmaufruf übertragen werden können.
- Der Zweck ist, mehrere zeitintensive Remotezugriffe durch einen einzigen zu ersetzen.
- Ein DTO ist in der Regel «immutable», d.h. enthält nur getter-Methoden.

Ausgewählte Implementierungsaspekte von Client-Server-Systemen

- Wir betrachten im Weiteren einige ausgewählte Aspekte:
  - Heterogenität
  - Serverarchitektur
  - Nebenläufigkeit im Server (Parallelität)
  - Serverseitige Service- bzw. Dienstschnittstellen
  - Fehlersituationen, Fehlerklassierung
  - Parameterübergabe zwischen Client und Server
  - Marshalling/Unmarshalling
  - Kommunikation
  - Zustandsverwaltung
  - Garbage Collection
  - Lastverteilung, Verfügbarkeit, Skalierbarkeit

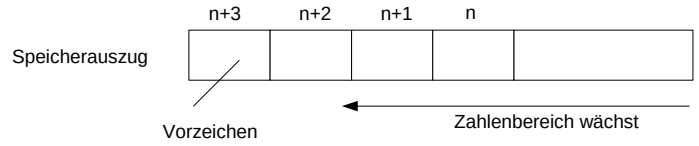
Heterogenität

- Mehrere Ebenen der Heterogenität
- Standardformate notwendig!

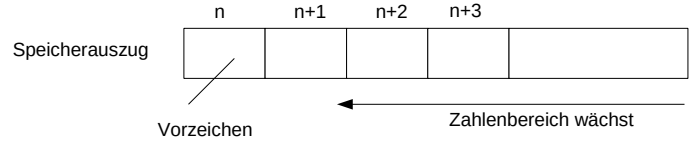


- **Rechnerhardware und Betriebssysteme**
  - Unterschiede bei der Speicherung der Daten
    - «Little Endian» versus «Big Endian»
  - Unterschiedliche Zeichensätze
    - ASCII - EBCDIC - Unicode

Darstellung: "little endian"

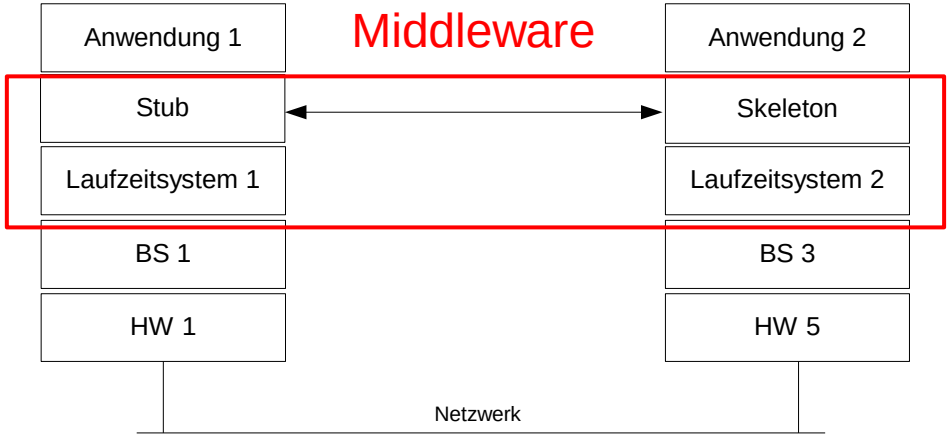


Darstellung: "big endian"



Überlegungen zur Überwindung von Heterogenität

- Was wir brauchen!
  - Einheitliche Transportsyntax (ASN.1, XDR, HTML, XML, JSON …) → Schicht 6 (ISO/OSI-Modell)
  - Middleware-Technologien bieten meist ähnliche Ansätze
  - Marshalling (Serialisierung) und Unmarshalling (Deserialisierung) der Nachrichten über generierten Code (Stubs und Skeletons)

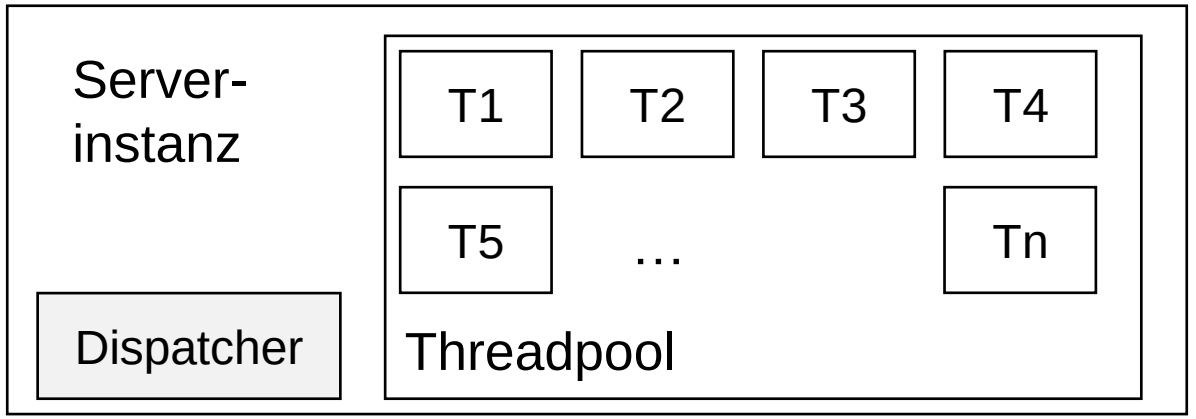


Nebenläufigkeit (Parallelität)

- Iterative (sequentielle) oder parallele Serverbausteine
- Threadpooling, Multithreading für die Bedienung mehrerer Clients gleichzeitig
- Ein Dispatcher ist ein Softwarebaustein im Server, der alle Requests der Clients entgegennimmt und sie auf Threads verteilt
- Einfaches sequentielles Programmiermodell für die Programmierer-Sicht
- Im JDK gibt es verschiedene Klassen für Thread-Pooling (s. java.util.concurrent)

Innenleben eines Servers

*Allg.: **Pooling** von Ressourcen = Vorbereiten zur schnelleren Nutzung*

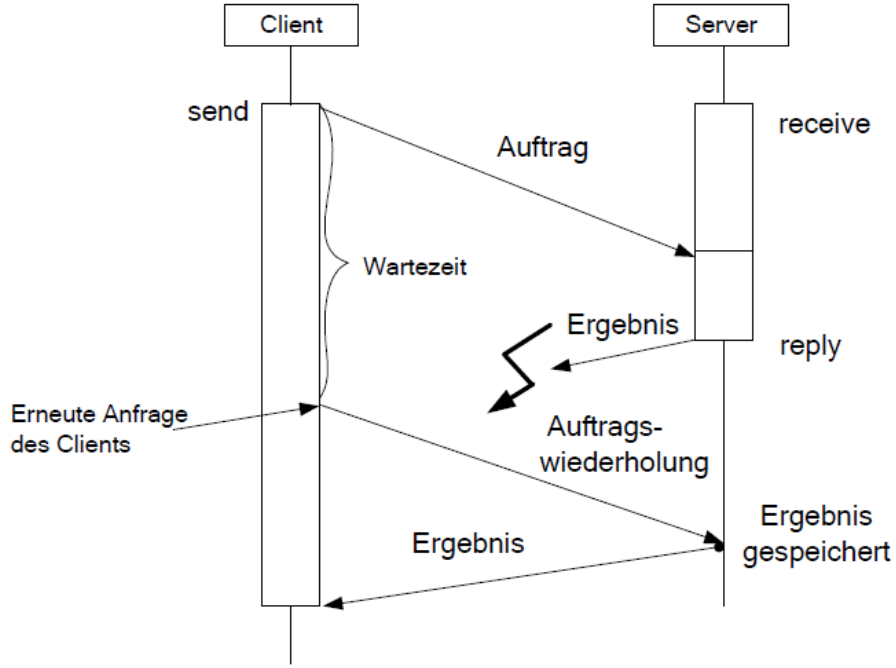


Dienst- bzw. Serviceschnittstellen

- Wie wird die Schnittstelle (Parameter- und Rückgabewertetypen) eines Serverbausteins beschrieben?
  - Neutrale Schnittstellenbeschreibungssprache oder eingebettet in Hostsprache (sprachabhängig)
  - Exception-Behandlung nicht immer gleich
- Diskussionsfrage:
  - Wie gut muss ein Server, der einen Service bereitstellt, prüfen, ob die empfangenen Parameter korrekt sind?

Fehlersituationen

- Es kann u.a. passieren, dass
  - ein Auftrag (engl. request) verloren geht,
  - das Ergebnis (engl. reply) des Servers verloren geht,
  - der Server während der Ausführung des Auftrags abstürzt,
  - der Server für die Bearbeitung des Auftrags zu lange braucht oder
  - der Client vor Ankunft des Ergebnisses abstürzt.



Parameterübergabe

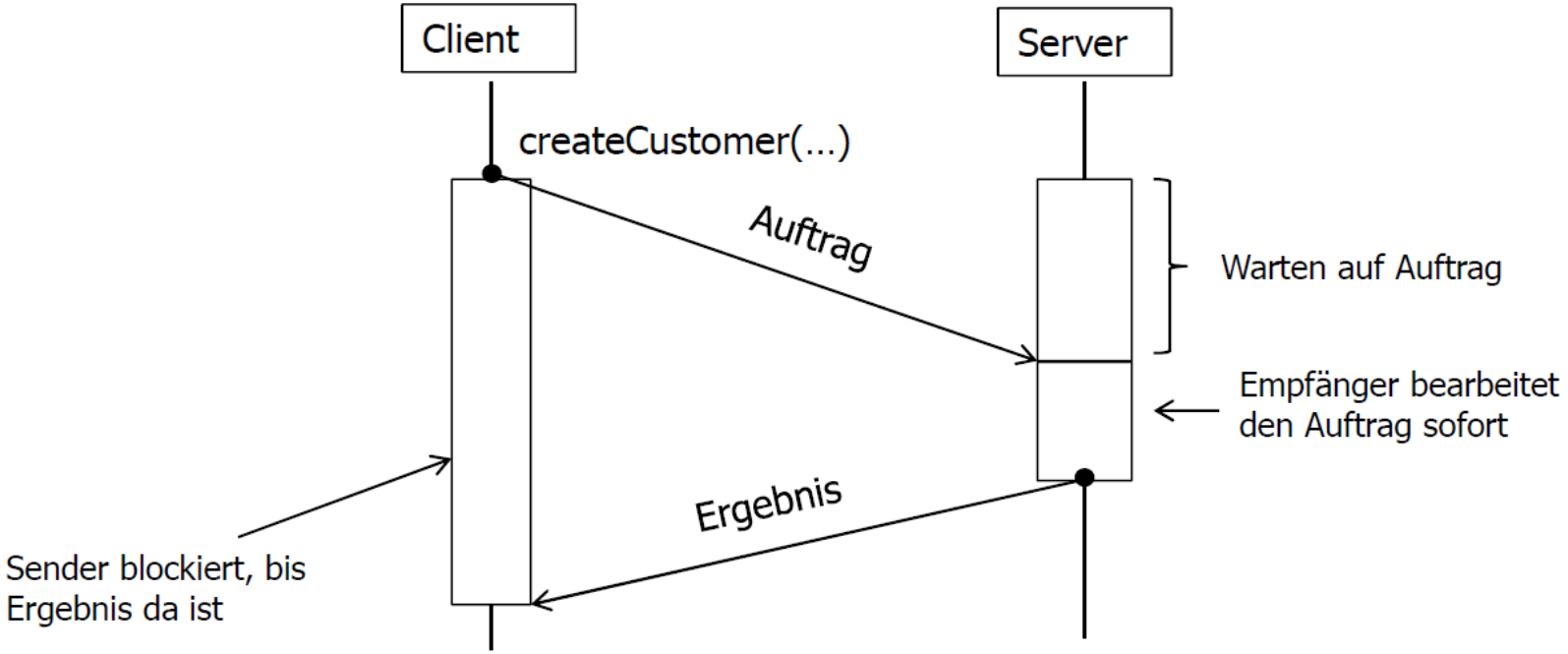
- Methodenaufruf und Parameterübergabe
  - ist lokal in demselben Prozess einfacher als bei entferntem (remote) Aufruf.
  - Entfernte Methodenaufrufe müssen für die Datenübertragung zwischen Rechnerknoten serialisiert (Marshalling) und deserialisiert (Unmarshalling) werden.
- Varianten für den entfernten Aufruf:
  - **Call-by-value:** Wert wird übergeben
    - Synonym: Call-by-copy
  - **Call-by-reference**: Verweis auf Variable wird übergeben
  - **Call-by-copy/copy-back**: Aufrufer arbeitet mit Kopie
    - Synonym: Call-by-restore = Call-by-value-result

Marshalling/Unmarshalling

- Marshalling/Unmarshalling ist das Umwandeln (Serialisierung/Deserialisierung) von strukturierten oder elementaren Daten für die Übermittlung an andere Prozesse.
- **Tag-basierte** Transfersyntax
  - Siehe ASN.1 mit BER (Basic Encoding Rules)
  - TLV-Kodierung (Type, Length, Value)
- **Tag-freie** Transfersyntax
  - Siehe Sun ONC XDR, CORBA CDR
  - Beschreibung der Daten aufgrund der Stellung in der Nachricht
  - Aufbau der Datenstrukturen ist dem Sender und dem Empfänger bekannt
- Meist automatische Erzeugung von Marshalling- und Unmarshalling-Routinen durch Compiler/Präcompiler
- Heute werden oft auch sprachunabhängige Notationen verwendet:
  - XML (Markup-Sprache), Tag-basiert
  - JSON (JavaScript Object Notation), Tag-basiert, sprachunabhängig?

Kommunikationsmodelle: Synchrone Kommunikation

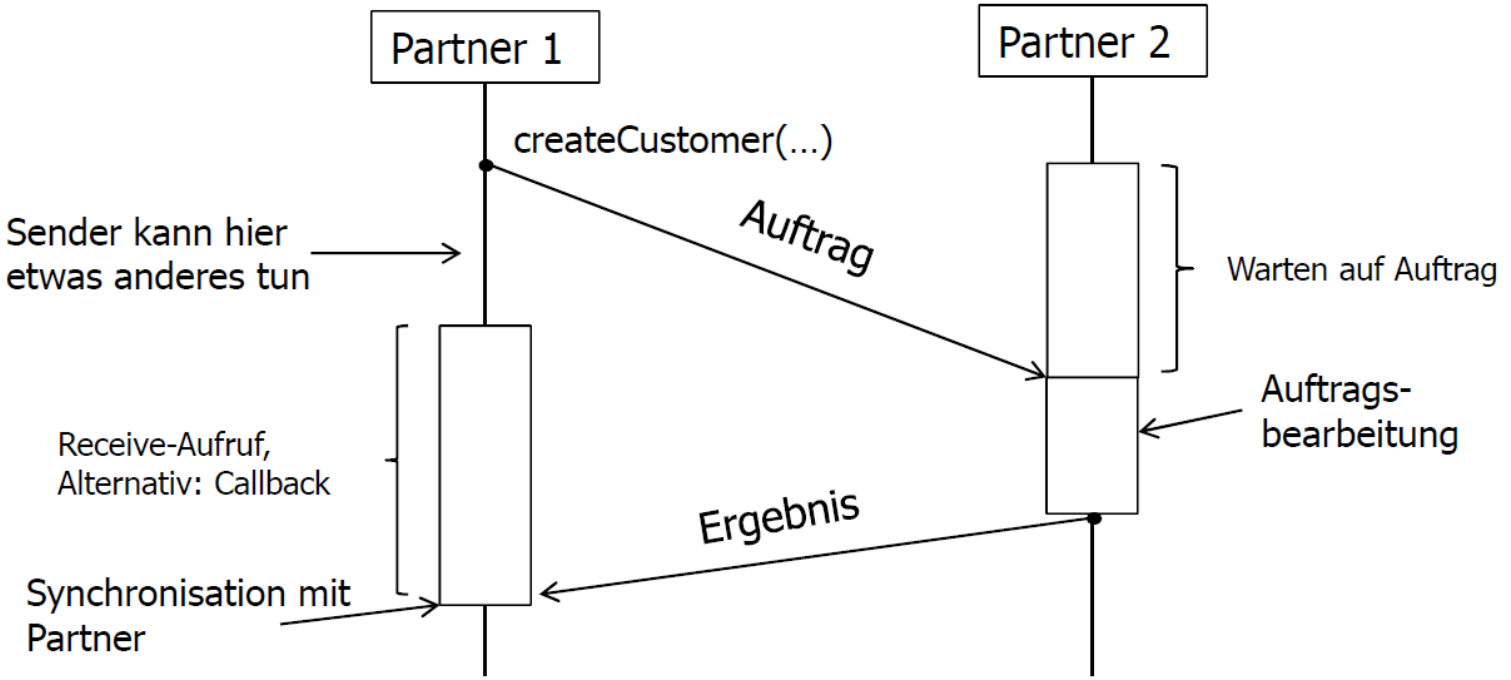
- Synchroner entfernter Dienstaufruf → blockierend
- Der Sender wartet, bis eine Methode send mit einem Ergebnis zurückkehrt



***Synchronisation** = **Synchronisierung (griech:** sýn = zusammen, chrónos = Zeit)**:** Aufeinander-Abstimmen von Vorgängen (zeitlich). Engere Bedeutung je nach Wissensgebiet: siehe Film, Informatik,…*

Kommunikationsmodelle: Asynchrone Kommunikation

- Asynchroner entfernter Serviceaufruf → Nicht blockierend, der Sender kann weiter machen



In der Datenkommunikation: asynchron = *Senden und Empfangen von Daten zeitlich versetzt und ohne Blockieren des Prozesses*

Ausgewählte Implementierungsaspekte von Client-Server-Systemen

- Wir betrachten im Weiteren einige ausgewählte Aspekte:
  - Heterogenität
  - Serverarchitektur
  - Nebenläufigkeit im Server (Parallelität)
  - Serverseitige Service- bzw. Dienstschnittstellen
  - Fehlersituationen, Fehlerklassierung
  - Parameterübergabe zwischen Client und Server
  - Marshalling/Unmarshalling
  - **Kommunikation**
  - Zustandsverwaltung
  - Garbage Collection
  - Lastverteilung, Verfügbarkeit, Skalierbarkeit

Kommunikation

- Namensauflösung und Adressierung auf der Anwendungsebene (entferntes Objekt oder Prozedur)
  - Naming- und Directory-Services notwendig
- Binding-Vorgang: Aufbau eines Verbindungskontextes zwischen Client und Server
  - Statisch zur Übersetzungszeit
  - Dynamisch zur Laufzeit
- Kommunikationsprotokoll für die Client-Server-Kommunikation
  - Nachrichtentypen (meist Request-Response-Protokolle)
  - Unterstützte Fehlersemantik
  - Unterstützung für verteiltes Garbage Collection

Zustandsverwaltung

- Server können zustandsinvariante und zustandsändernde Dienste bzw. Services anbieten
  - Zustandsändernde Dienste führen bei der Bearbeitung zu einer Änderung von Daten (z.B. in Datenbanken)
  - Zustandsinvariante Dienste verändern nichts
- Weiterer Aspekt: Server muss sich das Wissen über die Zustandsänderung über einen Aufruf hinweg merken
  - stateful und stateless Server
  - Stateless Server verwalten den aktuellen Zustand der Kommunikationsbeziehung zwischen Client und Server nicht
  - Wenn möglich: stateless!
- Zustandslose Kommunikationsprotokolle im Web: HTTP und REST für Webservices

Garbage Collection (GC)

- Verteiltes Reference-Counting
  - Server verwaltet eine Liste aller Clients (Proxies), die entfernte Referenzen nutzen
  - Server verwaltet Referenzzähler für alle benutzten Objekte
  - Client sendet spezielle Nachrichten an den Server, wenn Referenz benutzt bzw. gelöscht wird
- Leases
  - Referenz wird nur eine begrenzte Zeit für den Client freigegeben
  - Nach definierter Zeit löscht der Server die Referenz, wenn sich der Client nicht meldet
  - Ein Client kann sich somit problemlos beenden
- Zusammenarbeit mit lokalen GC-Mechanismen
  - Heap-Bereinigung

Lastverteilung, Hochverfügbarkeit, Skalierbarkeit

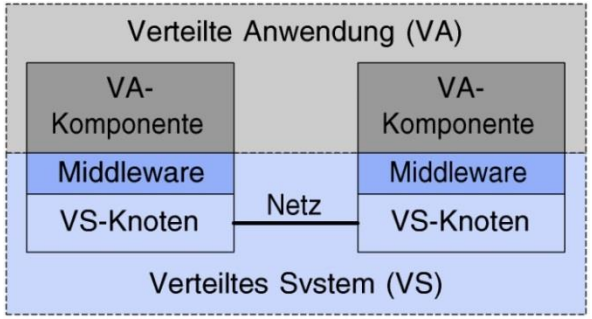
- Load Balancing (Lastverteilung)
  - Lastverteiler verteilen die Last auf mehrere Serverinstanzen
  - Dispatching z.B. über DNS-basiertes Request-Routing
- Hochverfügbarkeit
  - Server-Cluster, Beispiel: JBoss Cluster, Oracle Real Application Cluster
  - Failover
  - Session-Replikation
- Skalierbarkeit
  - Horizontal: Steigerung der Leistung durch Hinzunahme von Rechnern
  - Vertikal: Steigerung der Leistung durch Hinzufügen von Ressourcen zu einem Rechner (CPU, Speicher, …)

Agenda

1. Einführung in verteilte Systeme
2. Design- und Implementierungskonzepte von Client/Server-Systemen
3. **Middleware für verteilte Systeme**
4. Wrap-up und Ausblick

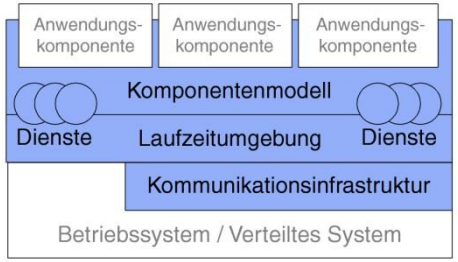
Middleware

- Middleware ist eine Softwareschicht, die den Anwendungen standardisierte, höhere Kommunikations- und sonstige Dienste über ein Application Programming Interface (API) bereitstellt und damit die transparente Kommunikation von Komponenten verteilter Systeme unterstützt.



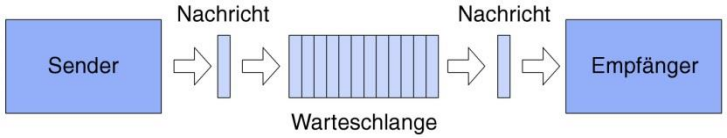
Middleware-Kategorien

- **Anwendungsorientierte Middleware**
  Java Enterprise Edition (EE) neu Jakarta EE
  Spring-Framework
  .NET Enterprise Services



- **Kommunikationsorientierte Middleware**
  Remote Procedure Call (RPC), Remote Method Invocation (RMI), REST, WebSocket …

- **Nachrichtenorientierte Middleware**
  Message Oriented Middleware (MOM),
  Java Messaging Service (JMS), MQTT …



Eine Middleware-Plattform vereinigt die verschiedenen Kategorien zu einer vollständigen verteilten Plattform für verteilte Anwendungen auf allen Tiers (Java EE, .NET)

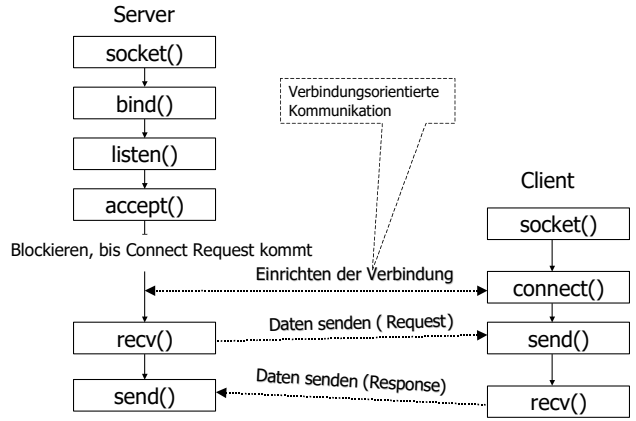
Implementierungskonzepte: Konkrete Ansätze für das Client-Server-Modell

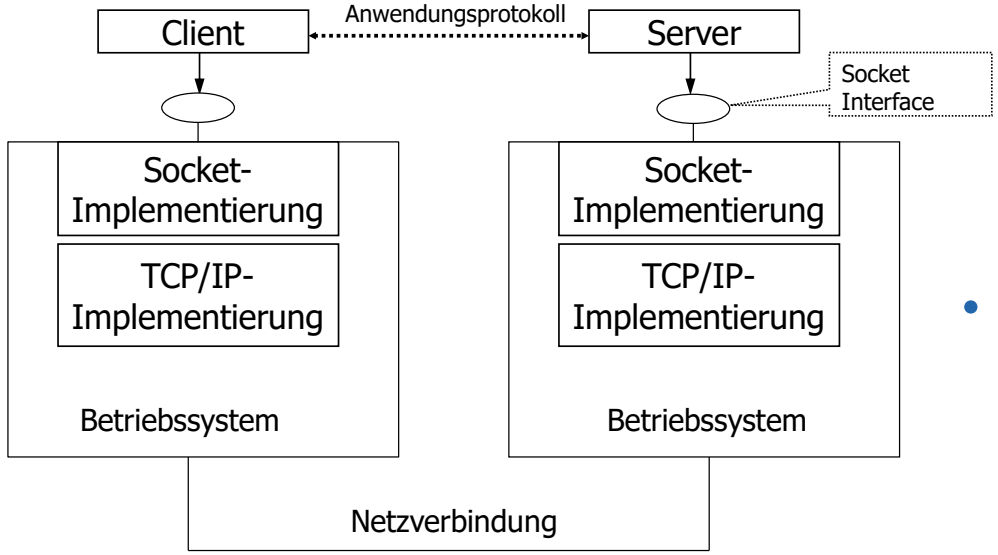
- **Remote Procedure Call (RPC)**
  - z.B. Sun ONC RPC, DCE RPC
- **Verteilte Objekte**
  - z.B. CORBA, Java RMI, .NET Remoting
- **Verteilte Services**
  - z.B. Webservices, SOAP, RESTful

Historische Entwicklung

Recap: Nachrichten-basierte Kommunikation mit dem Socket-API

- Socket (von engl. Sockel, Steckverbindung oder Steckdose) ist eine plattformunabhängige, standardisierte Netzwerk-Schnittstelle (API).
  - Verbindungsorientiert: **TCP** (Transmission Control Protocol)
  - Verbindungslos: **UDP** (User Datagram Protocol)

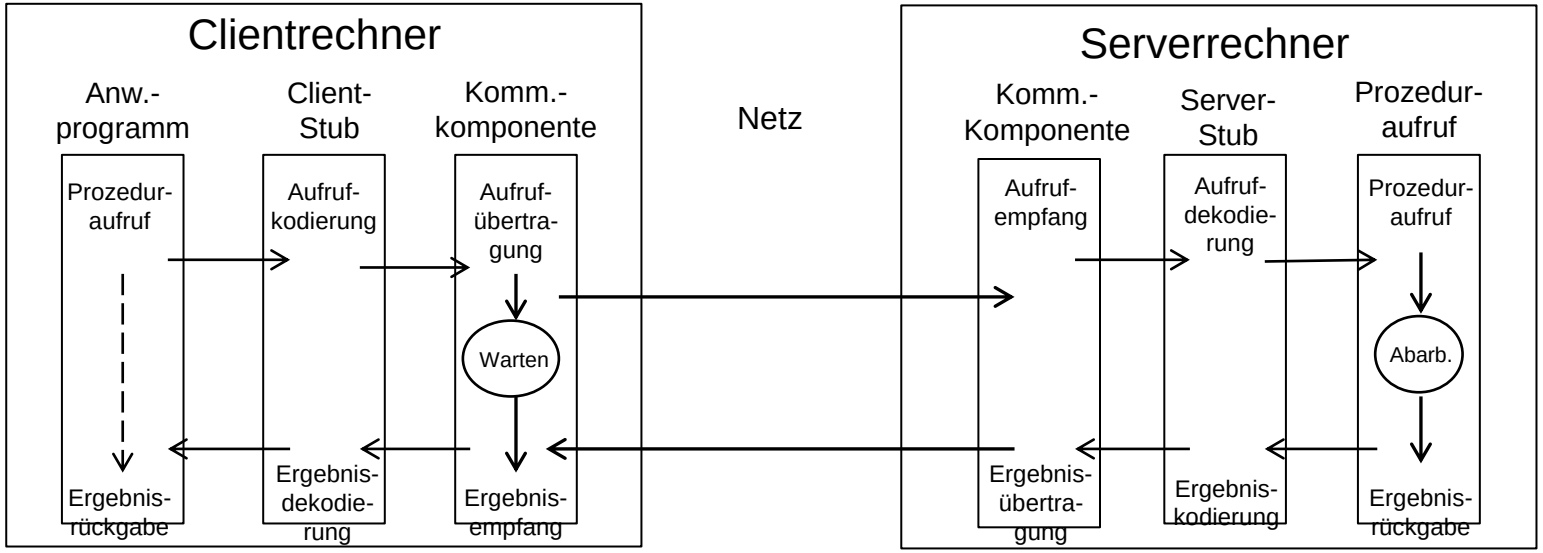




- Typisches Kommunikations-modell ist das Client-Server-Modell.

Verteilte Prozeduraufrufe (RPC)

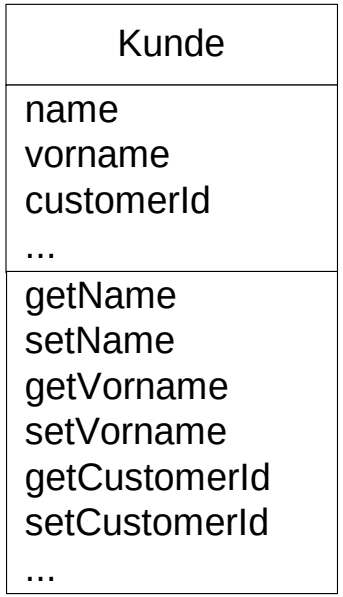
- RPC (Remote Procedure Call) ist eine Möglichkeit, Client/Server-Aufrufe zu implementieren.
- Der Grundgedanke von RPC wurde erstmals 1976 von James E. White im RFC 707 publiziert.
- Der genaue Aufbau von RPC ist in RFC 1057 und RFC 5531 beschrieben.



- Implementierungen: (Sun) ONC RPC, DCE RPC, Google gRPC, …

Objekt-basierte Kommunikation
Ausgangsidee: Lokales Objektmodell

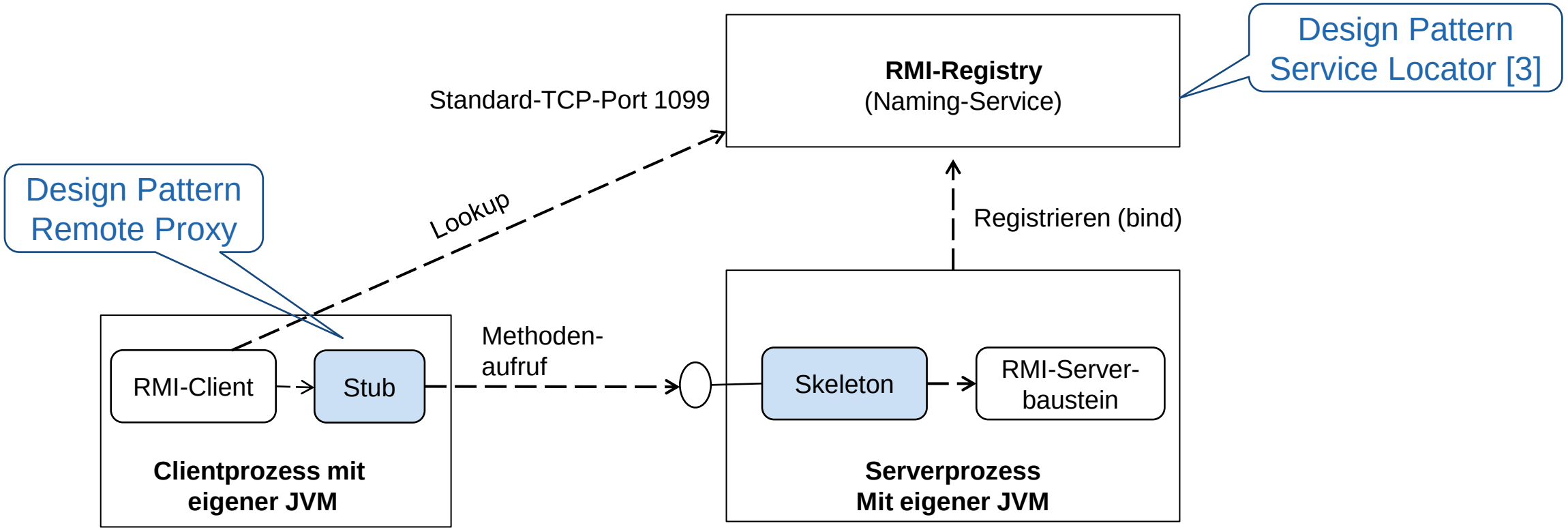
Zürcher Hochschule
für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie




```
class Kunde {
        private String name;
        private String vorname;
        public int customerId;
        …
        public Kunde(String name, String vorname, int id)
        {   /* Konstruktorcode */ }
        public String getName() {…}
        public String getVorname() {…}
        …
}
public static void main() {
        Kunde customer = new Kunde();
        String name = Customer.getName();
        int id = Customer.customerId;
        …
}
```

- Erweiterung des Client/Server-Modells um OO-Eigenschaften
- Entfernte («Remote») Objekte sind von Clients aus nutzbar
- Anwendung der Design Patterns Remote Facade, Remote Proxy und Data Transfer Object (DTO) [3]

Fallstudie Java RMI: Architektur und Überblick (1/2)

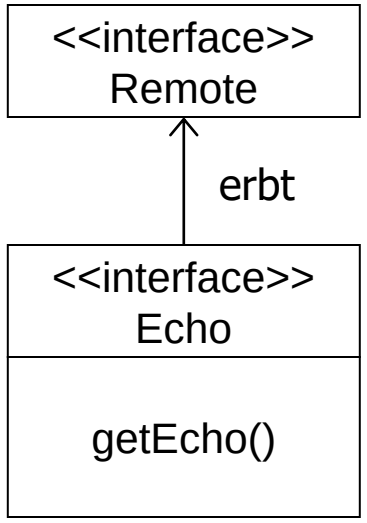


Fallstudie Java RMI: Architektur und Überblick (2/2)

- Vorteil von Java RMI (Remote Method Invocation) ist die einheitliche Architektur auf allen Systemen
  - Auf jedem Rechner ist eine Java Virtual Machine (JVM)
  - Die Datenrepräsentation ist immer gleich, keine Unterschiede in der lokalen Präsentation der Daten (Speicherung von Integern, ...)
- Nachteil: Alle Partner müssen in Java programmiert sein!
- Die Adressierung von Serverobjekten erfolgt über URLs
  - Allgemein: rmi://<server>:<port>/<object>
- Ein Nameservice auf dem Serverrechner nimmt URL-Anfragen entgegen und löst diese auf
  - Protokolltyp rmi://
- Die Kommunikation zwischen Client und Server erfolgt über ein proprietäres Protokoll dem Java Remote Method Protocol (JRMP) und setzt auf TCP/IP auf

Fallstudie Java RMI - Dienstschnittstellen: Schnittstelle (Service) definieren (1/2)

- Ein Remote-Interface wird vom Interface Remote abgeleitet
- Jede Methode wirft bei Fehlern eine Exception (RemoteException)
- Die Implementierung des Interface erfolgt wie gehabt

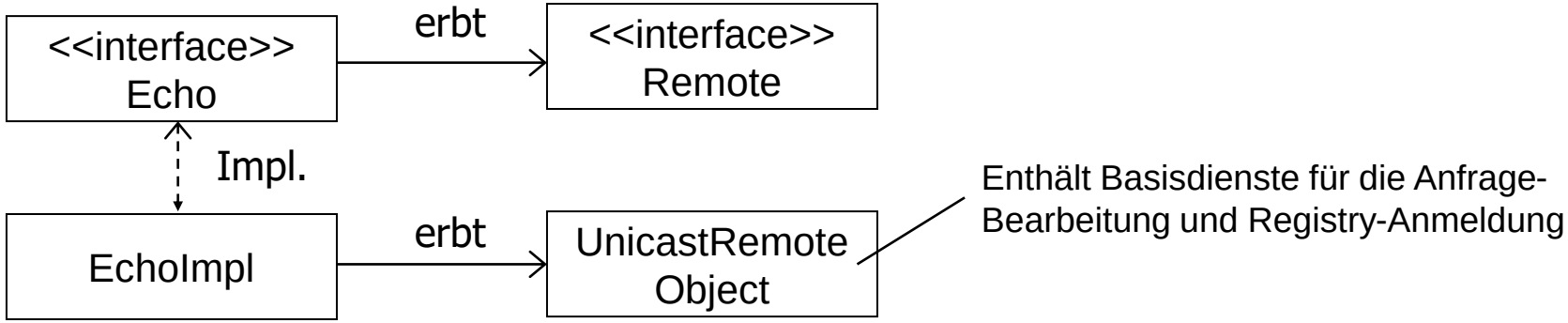



```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Echo extends Remote {
    String getEcho(String s) throws RemoteException;
}
```

Fallstudie Java RMI - Dienstschnittstellen: Schnittstelle (Service) definieren (2/2)

- Die Implementierungsklasse eines Remote-Interface wird abgeleitet von der Klasse UnicastRemoteObject




```java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class EchoImpl extends UnicastRemoteObject implements Echo {
    public EchoImpl() throws RemoteException {
    }

    public String getEcho(String s) {
        return s;
    }
}
```

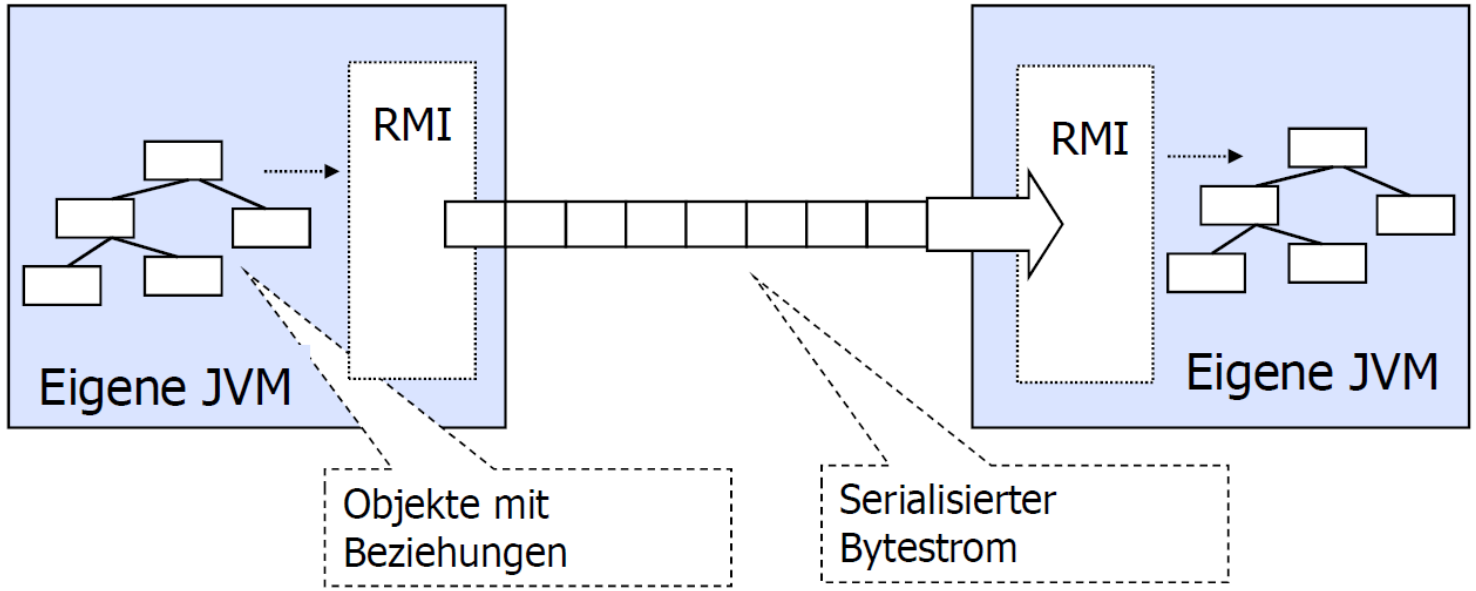
Fallstudie Java RMI – Marshalling/Unmarshalling: Objektserialisierung

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

- Objekte werden zur Übertragung von RMI serialisiert (Marshalling) → Parameter müssen serialisierbar sein (Interface Serializable implementieren!)
- Empfänger deserialisiert Objekte (Unmarshalling)

Regeln

Typ	lokale Methode	entfernte Methode
einfacher Typ	by value	by value
Objekt	by reference	by value (Serialisierung)
entferntes Objekt	by reference	by remote reference (Stub-Objekt)



Fallstudie Java RMI - Adressierung und Kommunikation: RMI-Server

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

```java
import java.rmi.Naming;
import java.rmi.Remote;

public class EchoServer {
    public static void main(String args[]) throws Exception {
        Remote remote = new EchoImpl();
        Naming.rebind("echo", remote);
        System.out.println("EchoServer started ...");
    }
}
```

- Alternativ muss die implementierende Klasse nicht unbedingt von UnicastRemoteObject abgeleitet werden. Bsp.: `UnicastRemoteObject.exportObject(remote, 50000);`
- Mit dem zweiten Parameter kann explizit das Port spezifiziert werden.

Fallstudie Java RMI - Adressierung und Kommunikation: RMI-Client

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

```java
import java.rmi.Naming;

public class EchoClient {
    public static void main(String args[]) throws Exception {
        String host = args[0];

        Echo remote = (Echo) Naming.lookup("//" + host + "/echo");
        String received = remote.getEcho("This is a test.");
        System.out.println(received);
    }
}
```

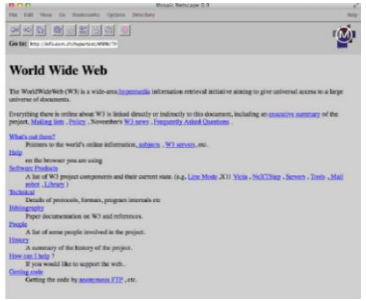
- host steht hier für den Hostnamen des Rechners, wo die RMI-Registry läuft.
- Falls kein port spezifiziert wird, ist der Default 1099.

Fallstudie Java RMI – Weitere Aspekte

- RMI bietet Threadpooling (paralleler Server). Aber ein Server ist nicht automatisch thread-safe!
- Server können verschieden aktiviert werden (sofort oder bei Bedarf).
- Ein (serialisierbares) Objekt kann auch dann vom Client zum Server transportiert werden, wenn der Server den Bytecode der zugehörigen Klasse noch nicht lokal zur Verfügung hat (mobile Agenten).
- RMI bietet eine Distributed Garbage Collection (nicht mehr referenzierte Objekte werden eingesammelt und gelöscht).
- Für die sichere Datenübertragung bietet RMI Verschlüsselung mittels SSL bzw. TLS (ServerSocket-Factory-Klassen).

Web-basierte Kommunikation

- Das Web bzw. WWW basiert auf drei (Kern-)Standards:
  - HTTP als Kommunikationsprotokoll, mit dem der Browser Informationen vom Webserver anfordern kann.
  - HTML als Auszeichnungssprache (engl. markup language), die festlegt, wie die Information gegliedert ist und wie die Dokumente verknüpft sind (Hyperlinks).
  - URIs als eindeutige Identifizierung einer Ressource, die in Hyperlinks verwendet wird (z.B. http://www.google.com).
- Folgende Standards kamen später dazu:
  - Cascading Style Sheets (CSS) legen das Aussehen der Elemente einer Webseite fest.
  - Hypertext Transfer Protocol Secure (HTTPS) ist eine Weiterentwicklung von HTTP, bei der Datentransfer komplett verschlüsselt wird.
  - Document Object Model (DOM) als Programmierschnittstelle für externe Programme oder Skriptsprachen von Browsern.
  - Skript- oder Makrosprache von Webbrowsern JavaScript (Standardisiert durch ECMA).



Das Web 1992

Das HTTP-Protokoll

- HTTP ist ein Protokoll der Anwendungsschicht im TCP/IP-Schichtenmodell.
- Es Regelt insbesondere, wie ein Webbrowser mit einem Webserver im World Wide Web (WWW) kommuniziert.
- HTTP setzt auf der Transportschicht TCP auf.
- Damit ein Webbrowser eine Webseite im Web abrufen kann, muss er sie zunächst adressieren (mittels URI bzw. URL).
- HTTP Versionen:
  - HTTP/1.0 (seit 1996, im RFC 1945 der IETF spezifiziert)
  - HTTP/1.1 (seit 1999, im RFC 723X der IETF spezifiziert)
  - HTTP/2 (seit 2015, im RFC 7540 der IETF spezifiziert)
- Die neue Version HTTP/2 soll die Übertragung durch Zusammenfassung mehrerer Anfragen, Datenkompression und binäre Übertragung beschleunigen und optimieren.

Denkpause

Aufgabe 10.2 (5’)

Diskutieren Sie in Murmelgruppen folgende Frage:

- Wie ist das Client-Server-Architekturmodell in Webapplikationen implementiert (Kommunikation, Parameterübergabe, Zustandsverwaltung, Fehlertoleranz etc.)?

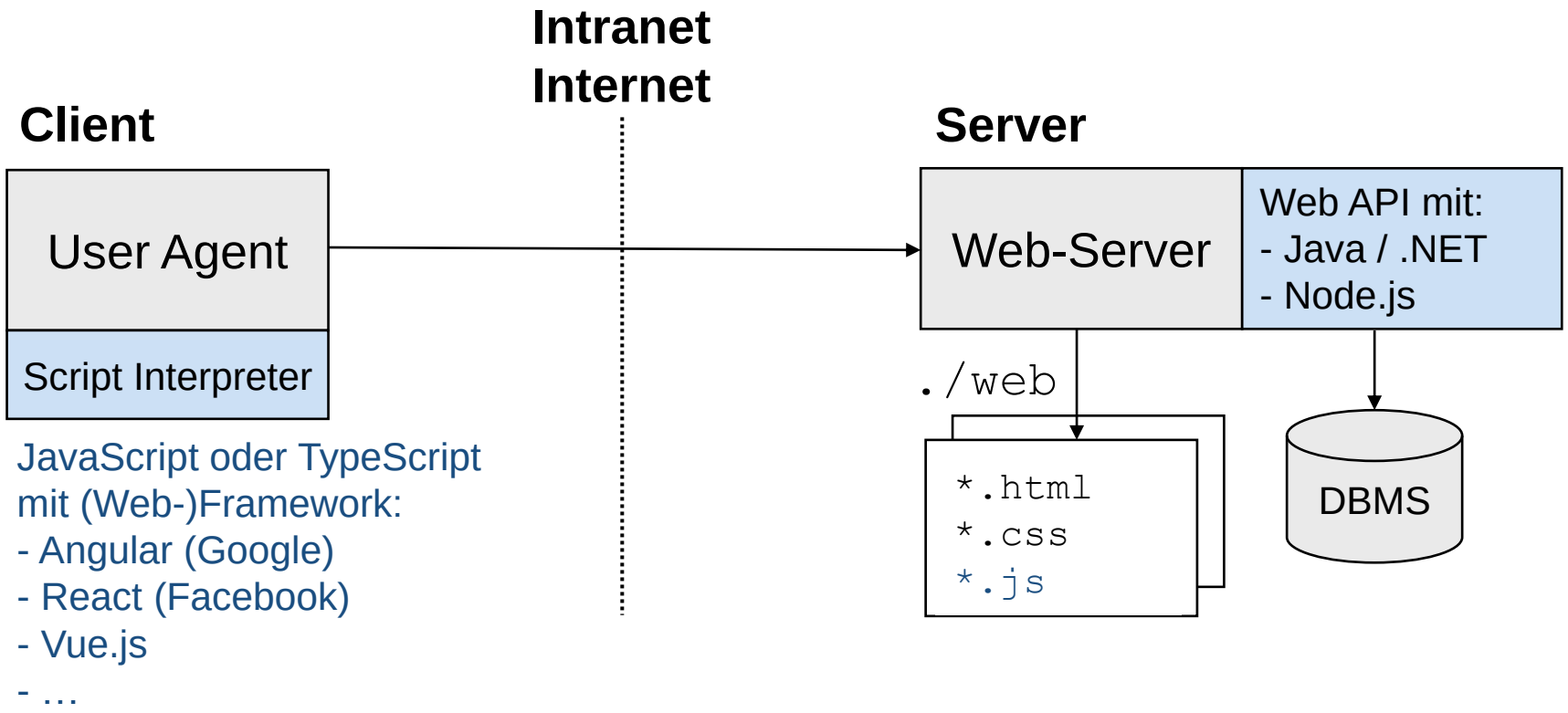
Denkpause

Aufgabe 10.2 – Musterlösung

Folgende Punkte sind hervorzuheben:

- Klassisches Client-Server-Architekturmodell
- HTTP implementiert RPC mit fixen Methoden (GET, POST etc.)
- Namensauflösung über eine eindeutige URL (DNS etc.)
- Parameterübergabe nur «by value» mit verhandelbarem Format (MIME-Typ)
- HTTP ist zustandslos, Zustand im Webserver oder im Client (Browser)
- Fehlertoleranz gegeben durch TCP/IP: At most once
- Netzwerkinfrastruktur des Internets (TCP/IP) bietet Hochverfügbarkeit, Skalierung, Caching etc.

Moderne Single-Page-Webanwendungen



Eine Single-Page-Webanwendung (engl. single-page application, kurz SPA) besteht nur aus einem einzigen HTML-Dokument. Inhalte werden dynamisch nachgeladen (über ein Web API mit z.B. REST).

Asynchronous JavaScript and XML (Ajax)

- Ein Konzept der asynchronen Kommunikation zwischen einem Browser und dem Webserver.
- Dieses ermöglicht es, HTTP-Anfragen durchzuführen, während eine HTML-Seite angezeigt wird, und die Seite zu verändern, ohne sie komplett neu zu laden.
- Eine Ajax-Anwendung basiert auf folgenden Web-Techniken:
  - HTML (oder XHTML)
  - Document Object Model (DOM) zur Repräsentation der Daten oder Inhalte
  - JavaScript zur Manipulation des Document Object Models und zur dynamischen Darstellung der Inhalte
  - dem XMLHttpRequest-Objekt, Bestandteil vieler Browser, um Daten auf asynchroner Basis mit dem Webserver austauschen zu können
  - Mit der window.fetch-Methode und der neuen Fetch API gibt es eine bequemere Alternative zu XMLHttpRequest.

Fallstudie WebSockets (1/2)

- Das WebSocket-Protokoll ist ein auf TCP basierendes Netzwerkprotokoll, das eine Ergänzung zu HTTP darstellt, um eine bidirektionale Verbindung zwischen einer Webapplikation und einem einem Webserver, der auch WebSockets unterstützt, herzustellen.
- Der Client startet wie bei HTTP eine Anfrage, aber die zugrundeliegende TCP/IP-Verbindung bleibt nach der Übertragung der Client-Daten bestehen (Handshake-Mechanismus mit zwei Phasen).
- Ein ständiger Verbindungsauf- und -abbau entfällt. Zudem entfallen die HTTP-Header-Informationen bei jeder Anfrage und Antwort.
- Geeignet für viele modere Webanwendungen, die einen Server-Push benötigen wie z.B. Anzeige von Börsenkursen, Verkehrsinformationen, Online-Spiele.
- Aufbau der URI: `ws://server[:port][/resource]`
- Spezifiziert seit 2011 durch IETF im RFC 6455
- Java API for WebSockets: https://www.oracle.com/technical-resources/articles/java/jsr356.html

Fallstudie WebSockets (2/2)

Beispiel: WebSocket Echo Service mit Java

```
<script>
    var uri = "ws://" + window.location.host + "/ws/echo";
    var socket;

    function $(id) {
        return document.getElementById(id);
    }
    function sendMessage() {
        socket.send($("userInput").value);
    }
    window.onload = function () {
        if (! "WebSocket" in window) {
            $("status").innerHTML = "The browser does not support WebSockets.";
            return;
        }
        socket = new WebSocket(uri);
        socket.onopen = function () {
            $("status").innerHTML = "onOpen";
        };
        socket.onclose = function () {
            $("status").innerHTML = "onClose";
        };
        socket.onerror = function (error) {
            $("status").innerHTML = error;
        };
        socket.onmessage = function (message) {
            $("data").innerHTML = message.data;
        };
    }
    window.onunload = function () {
        socket.close();
    }
</script>
```

URL: ws://<host>/ws/echo

Event-Handler onOpen

Event-Handler onClose

Event-Handler onMessage

```
@ServerEndpoint("/echo")
public class Echo {
    private AtomicInteger counter = new AtomicInteger();
    private final Logger logger = Logger.getLogger(Logger.GLOB
                                                   GER_NA
    @OnOpen
    public void onOpen(Session session) {
        logger.fine(session.getId() + ": onOpen");
        try {
            session.getBasicRemote().sendText("connected");
        } catch (IOException e) {
        }
    }
    @OnClose
    public void onClose(Session session) {
        logger.fine(session.getId() + ": onClose");
    }
    @OnError
    public void onError(Session session, Throwable error) {
        logger.fine(session.getId() + ": " + error.getMessage());
    }
    @OnMessage
    public void onMessage(Session session, String message) {
        try {
            logger.fine(session.getId() + ": " + message);
            session.getBasicRemote().sendText(counter.incrementAndGet() + ". " + message);
        } catch (IOException e) {
            try {
                session.close();
            } catch (IOException e1) {
            }
        }
    }
}
```

- Serverseitig und im Browser können verschiedene Event-Handler registriert werden, um via WebSockets zu kommunizieren.

Webservices

- Ein Webservice (auch Web API) stellt eine Schnittstelle für die Maschine-zu-Maschine- oder Anwendungs-Kommunikation über Rechnernetze wie das Internet zur Verfügung.
- Dabei werden Daten ausgetauscht und auf entfernten Computern (Servern) Funktionen bzw. Methoden aufgerufen.
- Jeder Webservice besitzt einen Uniform Resource Identifier (URI), über den er eindeutig identifizierbar ist, sowie je nach Implementierung eine Schnittstellenbeschreibung in maschinenlesbarer Form definiert, wie mit dem Webservice zu interagieren ist.
- Die Kommunikation kann über Protokolle aus dem Internetkontext wie HTTP oder HTTPS erfolgen; über diese Protokolle wiederum kann beispielsweise XML oder JSON übertragen werden.

Fallstudie Webservices mit REST

- Representational State Transfer (REST) ist ein Design Pattern für Webservices (REST-basierte oder RESTful Web APIs):
  - Client/Server-Modell: Kein Client-Kontext im Server, Server ist also stateless
  - Ressourcen (Objekte):
    - URL/URI-basierte Adressierung der Ressourcen
  - REST nutzt als Transportprotokoll HTTP/HTTPS
    - RESTful HTTP
  - Beliebige Darstellungsmöglichkeiten für Ressourcen:
    - HTML, JSON, XML, …
  - REST-konforme Dienste: GET, POST, PUT, DELETE, ...
  - HATEOAS (Hypermedia as the Engine of Application State):
    - Ressourcen sind – wie im Web üblich – über Links vernetzt
    - Entwurfsprinzip für REST-Architekturen
  - Basis für REST ist eine Dissertation von Roy Fielding im Jahr 2000

REST-basierte Webservices: URI-Design (1/2)

- URIs identifizieren Ressourcen
- Ressourcen sind «Dinge», Nomen, keine Aktionen oder Verben
- Spezifikation einer REST-Schnittstelle (API) mit URI-Templates:
  `http[s]://server:port/basePath/resourcePath`
  - Beispiel 1: http://example.com/api/customers/{id}
  - Beispiel 2: http://example.com/api/customers/{id}/orders/{id}
- Korrektes Ressourcendesign ergibt in der Regel automatisch passende Namen
- Query-Parameter als Filter bzw. genauere Spezifikation der gewünschten Ressource(n)
  - Beispiel: http://example.com/api/customers/{id}?state=active
- IDs sollten möglichst dauerhaft konstant bleiben (fachlicher vs. technischer Schlüssel)

REST-basierte Webservices: URI-Design (2/2)

Beispiel: Ressourcen eines Bestell-Service

Ressource	URI-Template	HTTP-Methode
Liste aller Bestellungen	/orders	GET
Bestellung erstellen	/orders	POST
Bestimmte Bestellung	/orders/{id}	GET
Bestellung aktualisieren	/orders/{id}	PUT
Bestellung löschen	/orders/{id}	DELETE
Stornierte Bestellungen	/orders?state=cancelled	GET
…		

Anmerkung:

- GET, HEAD und OPTION werden als sichere Methoden bezeichnet, da sie den Zustand der Ressource nicht verändern.
- Alle Methoden ausser POST müssen idempotent sein!

REST-basierte Webservices in Java (1/2)

Beispiel: JAX-RS Hello World Service

```
4  package demo.hello;
5
6  import javax.ws.rs.GET;
10
11 @Path("/hello")
12 public class HelloResource {
13     public HelloResource() {
14     }
15     @GET
16     @Produces(MediaType.TEXT_PLAIN)
17     public String getPlainMessage() {
18         return "Hello World!";
19     }
20     @GET
21     @Produces(MediaType.TEXT_HTML)
22     public String getHtmlMessage() {
23         return "<html><body><h1>Hello World!<h1></body></html>";
24     }
25     @GET
26     @Produces(MediaType.APPLICATION_XML)
27     public String getXmlMessage() {
28         return "<?xml version=\"1.0\"?><hello>Hello World!</hello>";
29     }
30     @GET
31     @Produces(MediaType.APPLICATION_JSON)
32     public String getJsonMessage() {
33         return "{\"message\":\"Hello World!\"}";
34     }
```

Ressourcenpfad für den Zugriff

HTTP-Methode zur Ausführung

HTTP-Content-Type der Antwort

```
4  package demo.hello;
5
6  import java.util.HashSet;
7  import java.util.Set;
8  import javax.ws.rs.core.Application;
9
10 public class MyApplication extends Application {
11   private Set<Object> singletons = new HashSet<Object>();
12   private Set<Class<?>> classes = new HashSet<Class<?>>();
13
14   public MyApplication() {
15     singletons.add(new HelloResource());
16 //    classes.add(HelloResource.class);
17   }
18
19   @Override
20   public Set<Class<?>> getClasses() {
21     return classes;
22   }
23
24   @Override
25   public Set<Object> getSingletons() {
26     return singletons;
27   }
28 }
```

JAX-RS-Basis-Klasse für die Registrierung der Ressourcen im Server

Liste der JAX-RS-Ressourcen; für jede Anfrage wird eine neue Instanz erzeugt

Liste bereits erstellter Instanzen; Instanz wird für jede Anfrage beibehalten

REST-basierte Webservices in Java (2/2)

Beispiel: Deployment und Bereitstellung des Service

```
4  import java.net.URI;
10
11 public class Server {
12   public static void main(String[] args) throws Exception {
13     final String BASE_URL = "http://localhost:50000/api";
14
15     URI endpoint = new URI(BASE_URL);
16     ResourceConfig rc = ResourceConfig
17         .forApplicationClass(MyApplication.class);
18     JdkHttpServerFactory.createHttpServer(endpoint, rc);
19     System.out.println("Server started at: " + BASE_URL);
20     System.out.println("Press Ctrl + C to stop the server.");
21   }
22 }
```

Service-Aufruf:

```
curl -i -H "Accept: text/html"
http://<host>:50000/api/hello
```

Anmerkung:

- Zur Bereitstellung von REST-Services wird hier zur Demonstration der JDK HTTP Server mit der Erweiterung durch den Jersey-Container jersey-container-jdkhttp-2.x.jar verwendet.
- Mit jedem HTTP-Server, der die Servlet-Technologie unterstützt, kann ein JAX-RS Service veröffentlicht werden (z.B. Apache Tomcat v9.x oder höher).

Entwicklungsplattformen und API

- Die Implementierung REST-basierter Web Services wird von zahlreichen Frameworks unterstützt.
- Jede Programmierumgebung, jedes Betriebssystem, jede Entwicklungsumgebung unterstützt HTTP, URIs und viele Standardformate wie HTML, XML oder JSON!
- Im Rahmen des Java Community Process wurde das Java API for RESTful Web Services (JAX-RS) entwickelt.
- Das Open-Source-Projekt Jersey stellt die Referenzimplementierung für JAX-RS bereit.
- Dokumentation und ein User Guide sind verfügbar unter: https://jersey.github.io/
- Für das Testing von RESTful Webservices gibt es verschieden Tools wie Postman (https://www.postman.com) oder das beliebte Kommandozeilen-Programm cURL (http://curl.haxx.se/).

Agenda

1. Einführung in verteilte Systeme
2. Design- und Implementierungskonzepte von Client-Server-Systemen
3. Middleware für verteilte Systeme
4. **Wrap-up und Ausblick**

Wrap-up

- Ein verteiltes System setzt sich aus einzelnen voneinander unabhängigen Bausteinen (Komponenten) zusammen, die dem Benutzer wie ein einzelnes kohärentes System erscheinen.
- Verteilte Systeme sind komplizierter als nicht verteilte Systeme und es müssen verschiedene praktische Probleme gelöst werden (Heterogenität, Fehlersituationen, Deployment etc.).
- Gängige Architekturstile verteilter Systeme sind Client-Server, Peer-to-Peer und Event Systems.
- Wichtige Design- und Implementierungsaspekte von Client-Server-Systemen sind Request-Handling (Threading), Design der serverseitigen Serviceschnittstellen, unterstützte Fehlersemantik, Parameter-Übergabe (Call-by-value, Call-by-reference), Kommunikationsstil (synchron, asynchron), Zustandsverwaltung und Garbage Collection.
- Grundlegende Architektur und Design Patterns für verteilte Systeme sind: Remote Proxy, Service Locator, Data Transfer Object und Remote Facade.
- Java RMI (Remote Method Invocation) ist die objektorientierte Umsetzung des RPCs (Remote Procedure Call) in Java und realisiert einen transparenten, entfernten Methodenaufruf.
- Web-basierte Applikationen verwenden das zustandslose Protokoll HTTP(S), Ajax und RESTful Webservices für die Kommunikation zwischen Browser und Webserver.

Ausblick

- In der nächsten Lerneinheit werden wir:
  - das Thema GUI-Architekturen vertiefen.

Quellenverzeichnis

[1] Mandl, P.: Masterkurs Verteilte betriebliche Informationssysteme, Springer-Vieweg, 2008

[2] Schill, A.; Springer, T.: Verteilte Systeme, 2. Auflage, Springer-Vieweg, 2012

[3] Fowler, M.: Patterns of Enterprise Application Architecture, Addison Wesley, 1. Auflage, 2002

[4] Martin, R. C.: Clean Architecture: A Craftsman's Guide to Software Structure and Design, mitp Professional, 2018

[5] Abts D.: Masterkurs Client/Server Programmierung mit Java, 5. Auflage, Springer-Vieweg, 2019