Verteilte Systeme

SS 2013

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 6. Mai 2013

Verteilte Systeme

SS 2013

2 Middleware

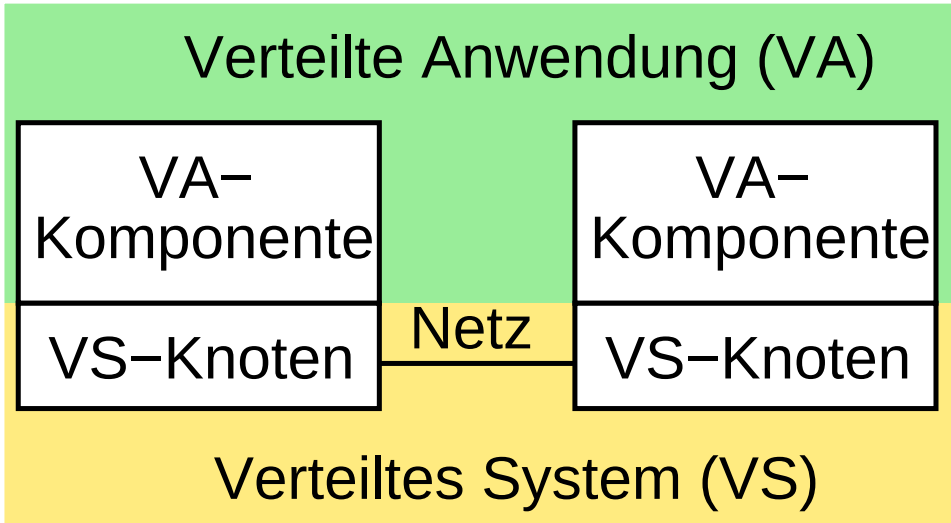
Inhalt

- Kommunikation in verteilten Systemen
- Kommunikationsorientierte Moddleware
- Anwendungsorientierte Middleware

Literatur

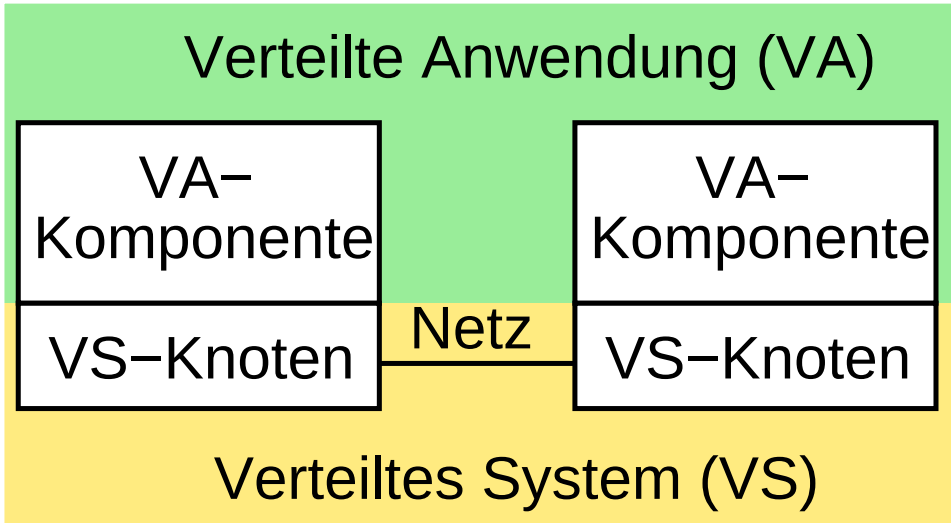
- Hammerschall: Kap. 2, 6
- Tanenbaum, van Steen: Kap. 2
- Colouris, Dollimore, Kindberg: Kap 4.4

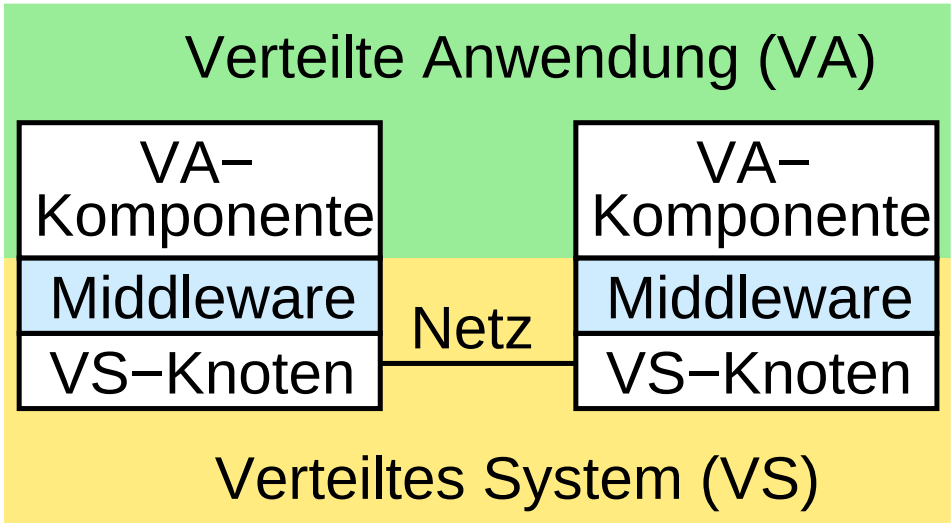
2 Middleware ...



- VA nutzt VS für Kommunikation zwischen ihren Komponenten
- VSe bieten i.a. nur einfache Kommunikationsdienste an
  - direkte Nutzung: **Netzwerkprogrammierung**
- **Middleware** bietet intelligentere Schnittstellen
  - verbirgt Details der Netzwerkprogrammierung

2 Middleware ...





- VA nutzt VS für Kommunikation zwischen ihren Komponenten
- VSe bieten i.a. nur einfache Kommunikationsdienste an
  - direkte Nutzung: **Netzwerkprogrammierung**
- **Middleware** bietet intelligentere Schnittstellen
  - verbirgt Details der Netzwerkprogrammierung

2 Middleware ...

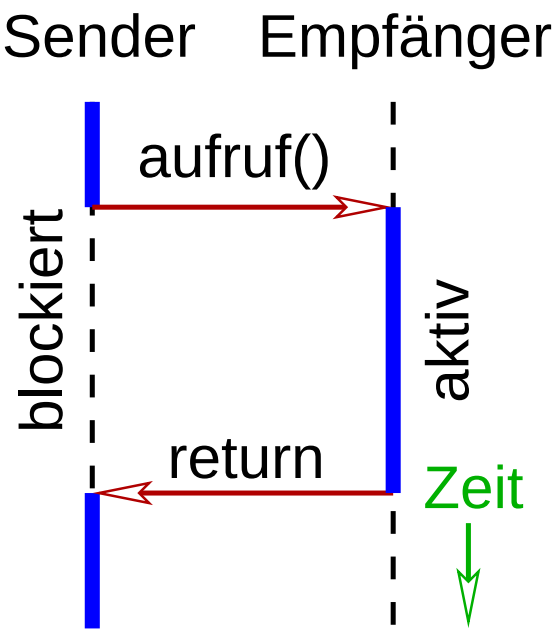
- Middleware ist Schnittstelle zwischen verteilter Anwendung und verteiltem System
- Ziel: Verbergen der Verteilungsaspekte vor der Anwendung
  - Transparenz (☞ **1.3**)
- Middleware kann auch Zusatzdienste für Anwendungen bieten
  - starke Unterschiede bei existierender Middleware
- Unterscheidung:
  - **kommunikationsorientierte Middleware** (☞ **2.2**)
    - (nur) Abstraktion von der Netzwerkprogrammierung
  - **anwendungsorientierte Middleware** (☞ **2.3**)
    - neben Kommunikation steht Unterstützung verteilter Anwendungen im Mittelpunkt

2.1 Kommunikation in verteilten Systemen

- Basis: **Interprozeßkommunikation** (**IPC**)
  - Austausch von Nachrichten zwischen Prozessen (☞ **BS_I: 3.2**)
    - auf demselben oder auf verschiedenen Knoten
    - z.B. über Ports, Mailboxen, Ströme, ...
- Zur Verteilung: Netzwerkprotokolle (☞ **RN_I**)
  - relevante Themen u.a.: Adressierung, Zuverlässigkeit, Reihenfolgeerhaltung, *Timeouts*, Bestätigungen, *Marshalling*
- Schnittstelle zur Netzwerkprogrammierung: Sockets (☞ **RN_II**)
  - Datagramme (UDP) und Ströme (TCP)

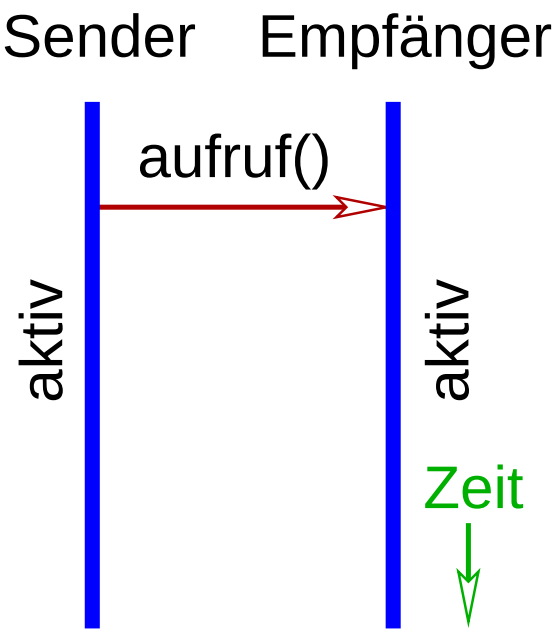
Synchrone Kommunikation

- Sender und Empfänger blockieren beim Aufruf der Sende- bzw. Empfangs-Operation
  - Empfänger wartet auf Aufruf
  - Sender wartet auf Ergebnis des Aufrufs
- Enge Kopplung zwischen Sender und Empfänger
  - Vorteil: einfach zu verstehendes Modell
  - Nachteil: hohe Abhängigkeit, insbes. im Fehlerfall
- Voraussetzungen:
  - zuverlässige und schnelle Netzwerk-Verbindung
  - Empfängerprozeß ist verfügbar

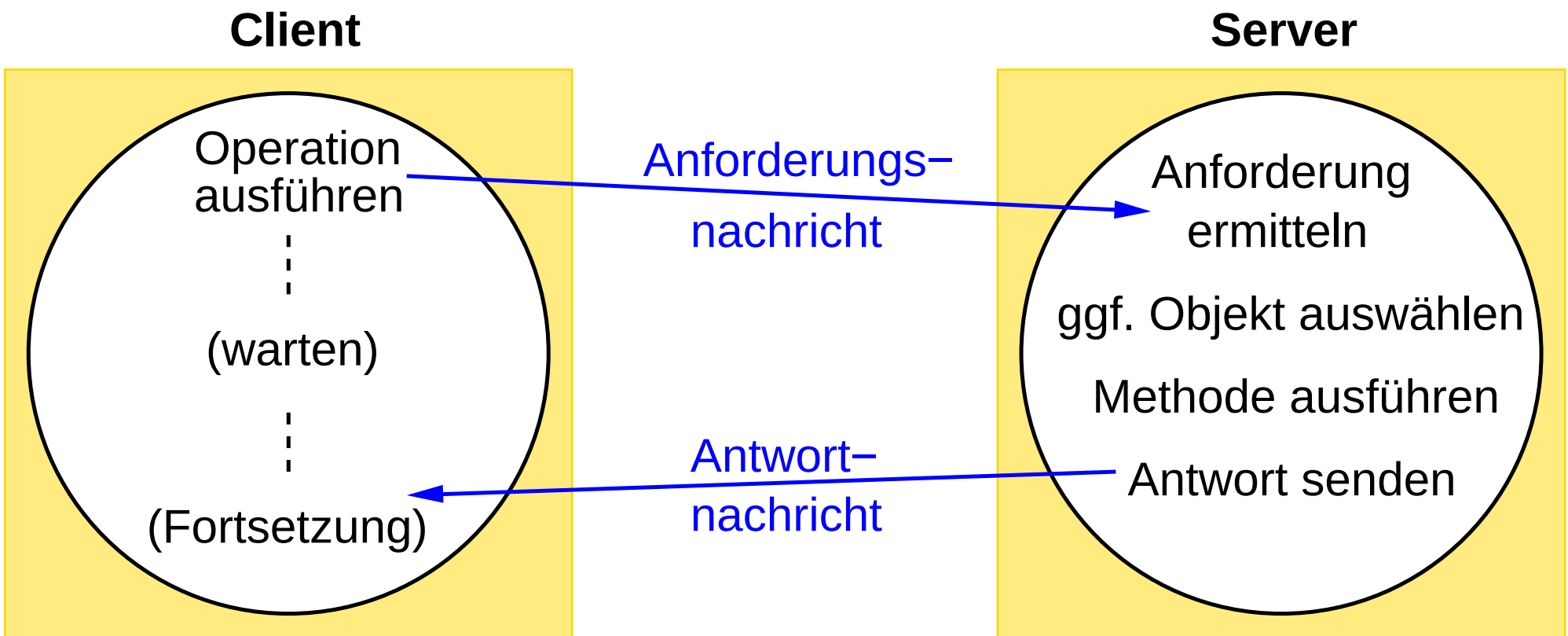


Asynchrone Kommunikation

- Sender wird nicht blockiert, kann nach dem Senden der Nachricht sofort weiterarbeiten
- Eingehende Nachrichten werden beim Empfänger gepuffert
- Antworten sind optional
  - Empfänger kann Antwort asynchron an Sender schicken
- Komplexere Implementierung / Verwendung als synchrone Kommunikation, aber meist effizienter
- Nur lose Kopplung zwischen den Prozessen
  - Empfänger muß nicht empfangsbereit sein
  - geringere Fehlerabhängigkeit

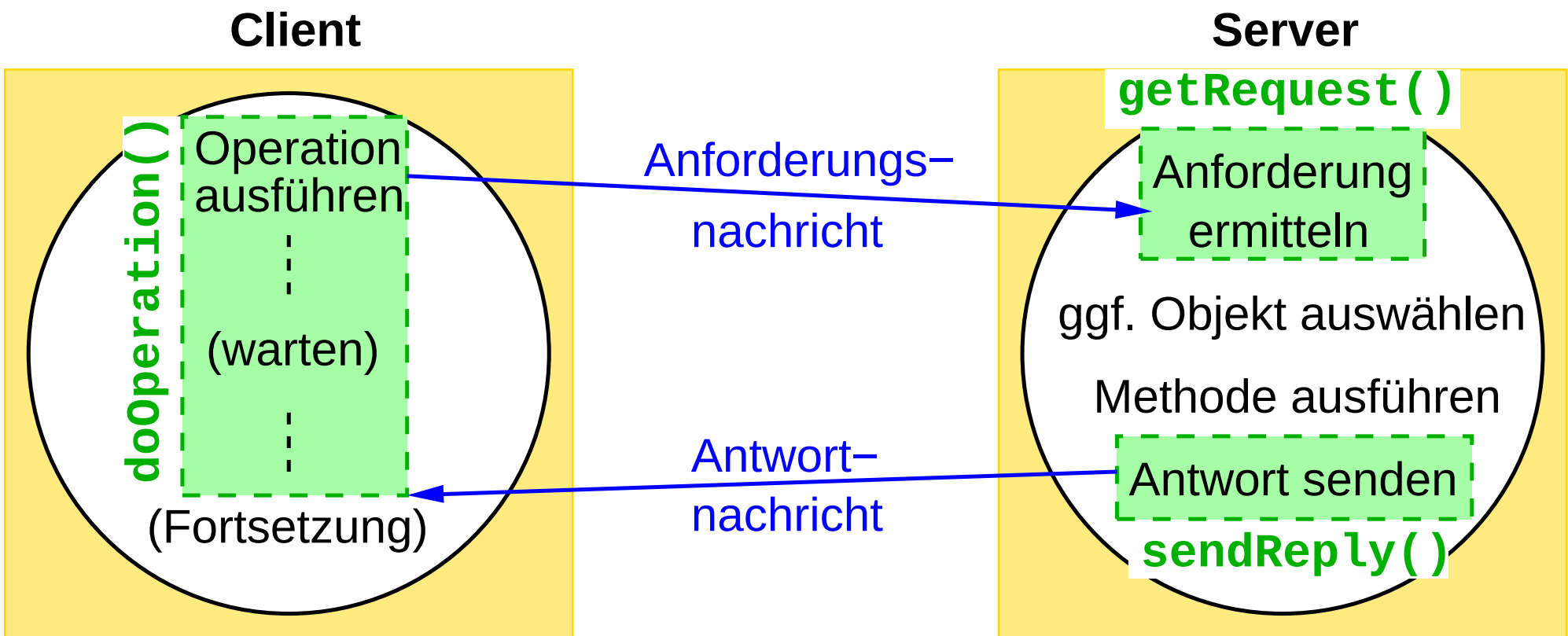


Client/Server-Kommunikation



- Meist synchron: Client blockiert, bis Antwort eintrifft
- Varianten: asynchron (nicht blockierend), *one way* (ohne Antwort)

Client/Server-Kommunikation



- Meist synchron: Client blockiert, bis Antwort eintrifft
- Varianten: asynchron (nicht blockierend), *one way* (ohne Antwort)

Client/Server-Kommunikation: Anfrage-/Antwort-Protokoll

- Typische Operationen:
  - `doOperation()` – sende Anfrage und warte auf Ergebnis
  - `getRequest()` – warte auf Anfrage
  - `sendReply()` – sende Ergebnis
- Typische Nachrichtenstruktur:

messageType	Anfrage / Antwort ?
requestID	eindeutige ID der Anfrage (i.a. int)
objectReference	Referenz auf entfernes Objekt (ggf.)
methodID	aufzurufende Methode (int / String)
arguments	Argumente (i.a. als Byte−Array)

  - Request-ID + Sender-ID ergeben eindeutige Nachrichten-ID
    - z.B. um Antwort einer Anfrage zuzuordnen

Client/Server-Kommunikation: Fehlerbehandlung

- Anfrage- und Antwort-Nachrichten können ggf. verloren gehen
- Client setzt *Timeout* bei Anfrage
  - nach Ablauf wird Anfrage i.a. erneut gesendet
  - nach einigen Wiederholungen: Abbruch mit Ausnahme
- Server verwirft doppelte Anfragen, falls Anfrage bereits / noch bearbeitet wird
- Bei verlorenen Antwort-Nachrichten:
  - idempotente Operationen können erneut ausgeführt werden
  - sonst: Ergebnisse der Operationen in *History* speichern
    - bei wiederholter Anfrage: nur Ergebnis neu senden
    - Löschen von *History*-Einträgen, wenn nächste Anfrage eintrifft; ggf. auch Bestätigungen für Ergebnisse

Verteilte Systeme

SS 2013

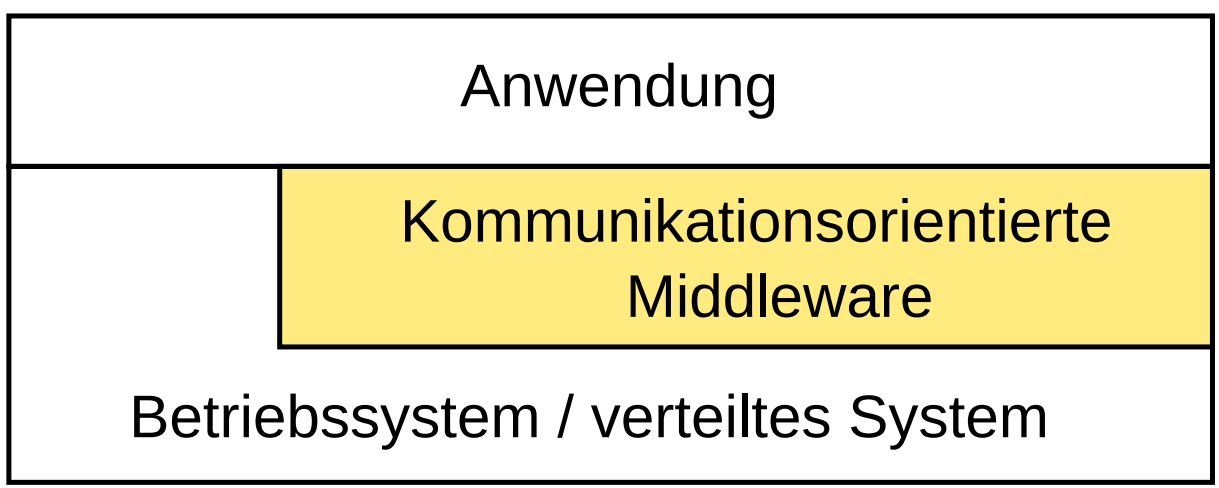
22.04.2013

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 6. Mai 2013

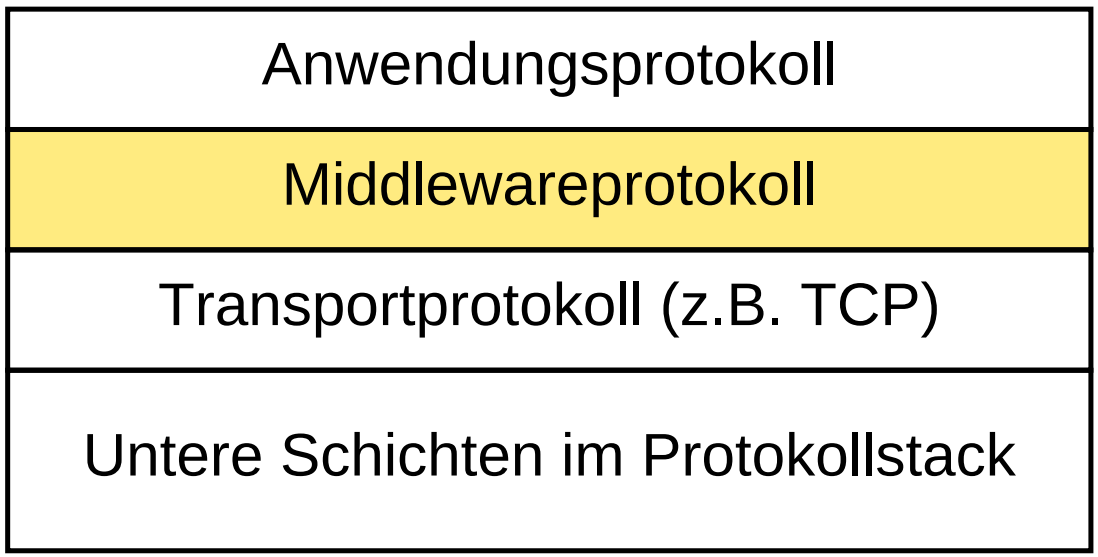
2.2 Kommunikationsorientierte Middleware

- Fokus: Bereitstellung einer Kommunikationsinfrastruktur für verteilte Anwendungen
- Aufgaben:
  - Kommunikation
  - Behandlung der Heterogenität
  - Fehlerbehandlung



Kommunikation

- Bereitstellung eines Middleware-Protokolls
- Lokalisierung und Identifikation der Kommunikationspartner
- Integration mit Prozeß- und Threadverwaltung



Heterogenität

- Problem bei der Datenübertragung:
  - Heterogenität in verteilten Systemen
- Heterogene Hardware und Betriebssysteme
  - unterschiedliche Byte-Reihenfolge
    - *Little Endian* / *Big Endian*
  - unterschiedliche Zeichencodierung
    - z.B. ASCII / Unicode / UTF-8 / EBCDIC (IBM Mainframes)
- Heterogene Programmiersprachen
  - unterschiedliche Darstellung von einfachen und komplexen Datentypen im Hauptspeicher

Heterogenität: Lösungen (☞ RN_I)

- Verwendung übergeordneter, einheitlicher Datenformate
  - allen Kommunikationspartnern und Middleware bekannt
  - plattformspezifische Formate für eine Middleware (z.B. CDR bei CORBA) oder externe Formate, z.B. XML
- Heterogenität von Hardware und Betriebssystem
  - wird transparent für die Anwendungen von der Middleware behandelt
- Heterogenität der Programmiersprachen
  - Anwendungen müssen Daten in übergeordnetes Format und zurück konvertieren (***Marshalling*** / ***Unmarshalling***)
    - notwendiger Code wird i.d.R. automatisch generiert
      - *Client-Stub / Server-Skeleton*

Fehlerbehandlung

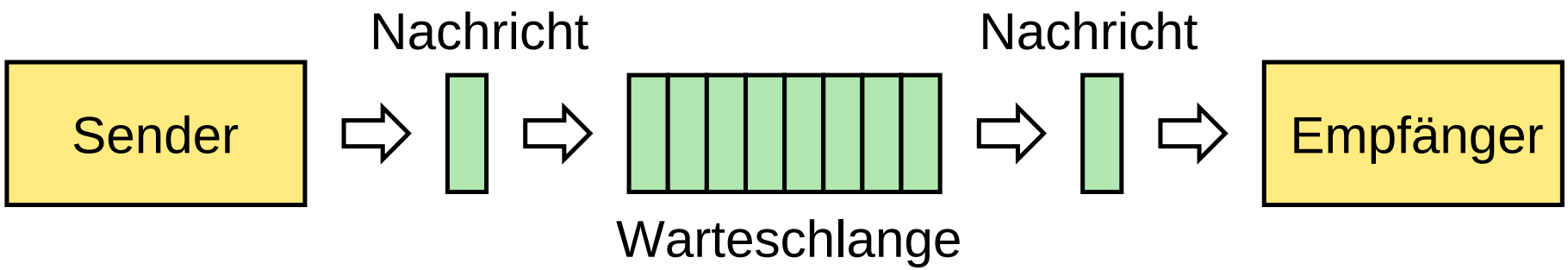
- Mögliche Fehler durch Verteilung
  - Fehlerhafte Übertragung (inkl. Verlust von Nachrichten)
    - Behandlung durch Protokolle des verteiltes Systems:
      - Prüfsummen, CRC
      - Neuübertragung von Paketen (z.B. bei TCP)
  - Ausfall von Komponenten (Netz, Hardware, Software)
    - Behandlung durch Middleware oder Anwendung:
      - Akzeptanz des Fehlers
      - Neusenden von Nachrichten
      - Replikation von Komponenten (Fehlervermeidung)
      - kontrolliertes Beenden der Anwendung

2.2.2 Programmiermodelle

- Programmiermodell legt zwei Konzepte fest:
  - Kommunikationsmodell
    - synchron vs. asynchron
  - Programmierparadigma
    - objektorientiert vs. prozedural
- Drei verbreitete Programmiermodelle bei Middleware:
  - nachrichtenorientiertes Modell (asynchron / beliebig)
  - entfernte Prozeduraufrufe (synchron / prozedural)
  - entfernte Methodenaufrufe (synchron / objektorientiert)

Nachrichtenorientiertes Modell

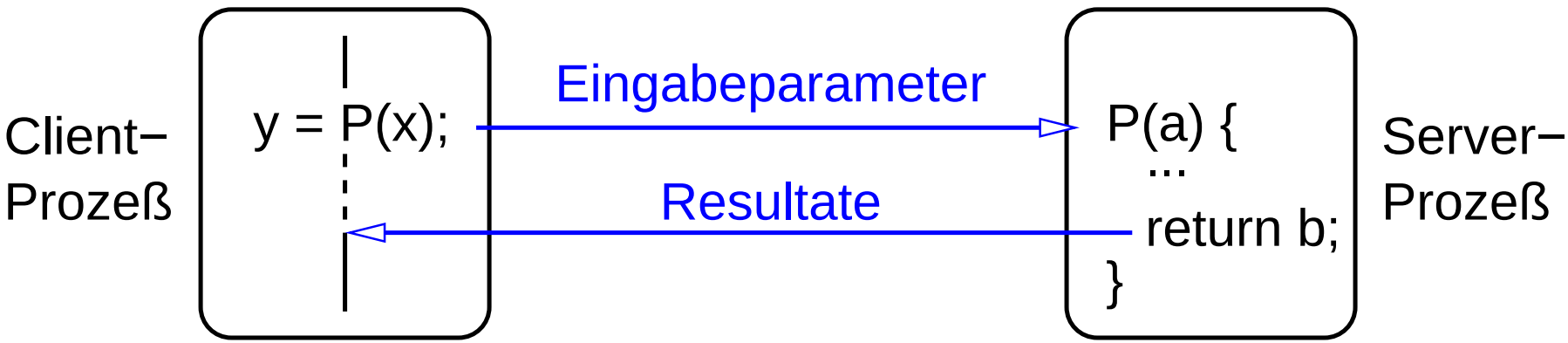
- Sender stellt Nachricht in Warteschlange des Empfängers



- Empfänger nimmt Nachricht an, sobald er bereit ist
- Weitgehende Entkopplung von Sender und Empfänger
- Keine Methoden- oder Prozeduraufrufe
  - Daten werden von der Anwendung verpackt und verschickt
  - keine automatische Antwort-Nachricht

Entfernter Prozeduraufruf (RPC, *Remote Procedure Call*)

- Ermöglicht einem Client den Aufruf einer Prozedur in einem entfernten Server-Prozeß



- Kommunikation nach Anfrage / Antwort-Prinzip

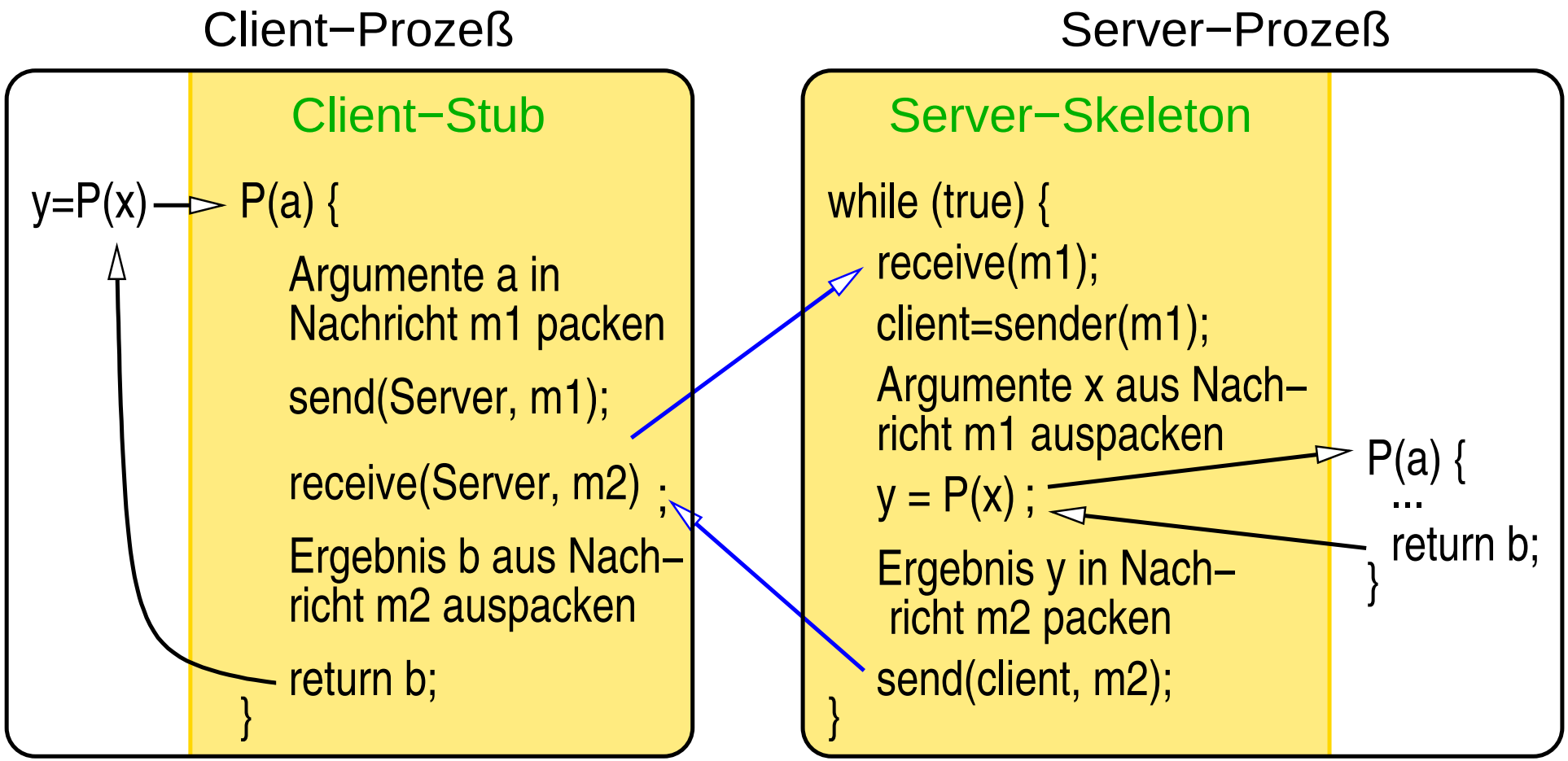
Entfernter Methodenaufruf (RMI, *Remote Method Invocation*)

- Ermöglicht einem Objekt, Methoden eines entfernten Objekts aufzurufen
- Prinzipiell sehr ähnlich zu RPC

Gemeinsame Grundkonzepte entfernter Aufrufe

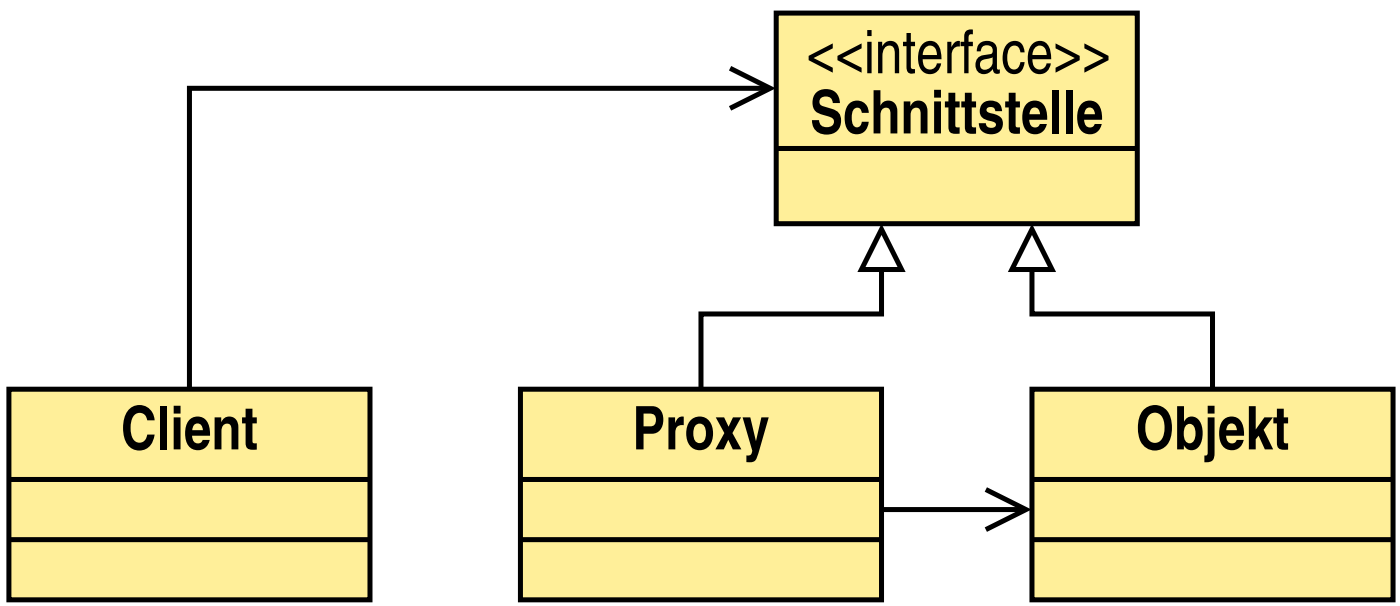
- Client und Server werden durch Schnittstellendefinition entkoppelt
  - legt Namen der Aufrufe, Parameter und Rückgabewerte fest
- Einführung von ***Client-Stubs*** und ***Server-Stubs*** (***Skeletons***) als Zugriffsschnittstelle
  - werden automatisch aus Schnittstellendefinition generiert
    - IDL-Compiler, *Interface Definition Language*
  - sind verantwortlich für *Marshalling / Unmarshalling* sowie für die eigentliche Kommunikation
  - realisieren Zugriffs- und Ortstransparenz

Funktionsweise der Client- und Server-Stubs (RPC)

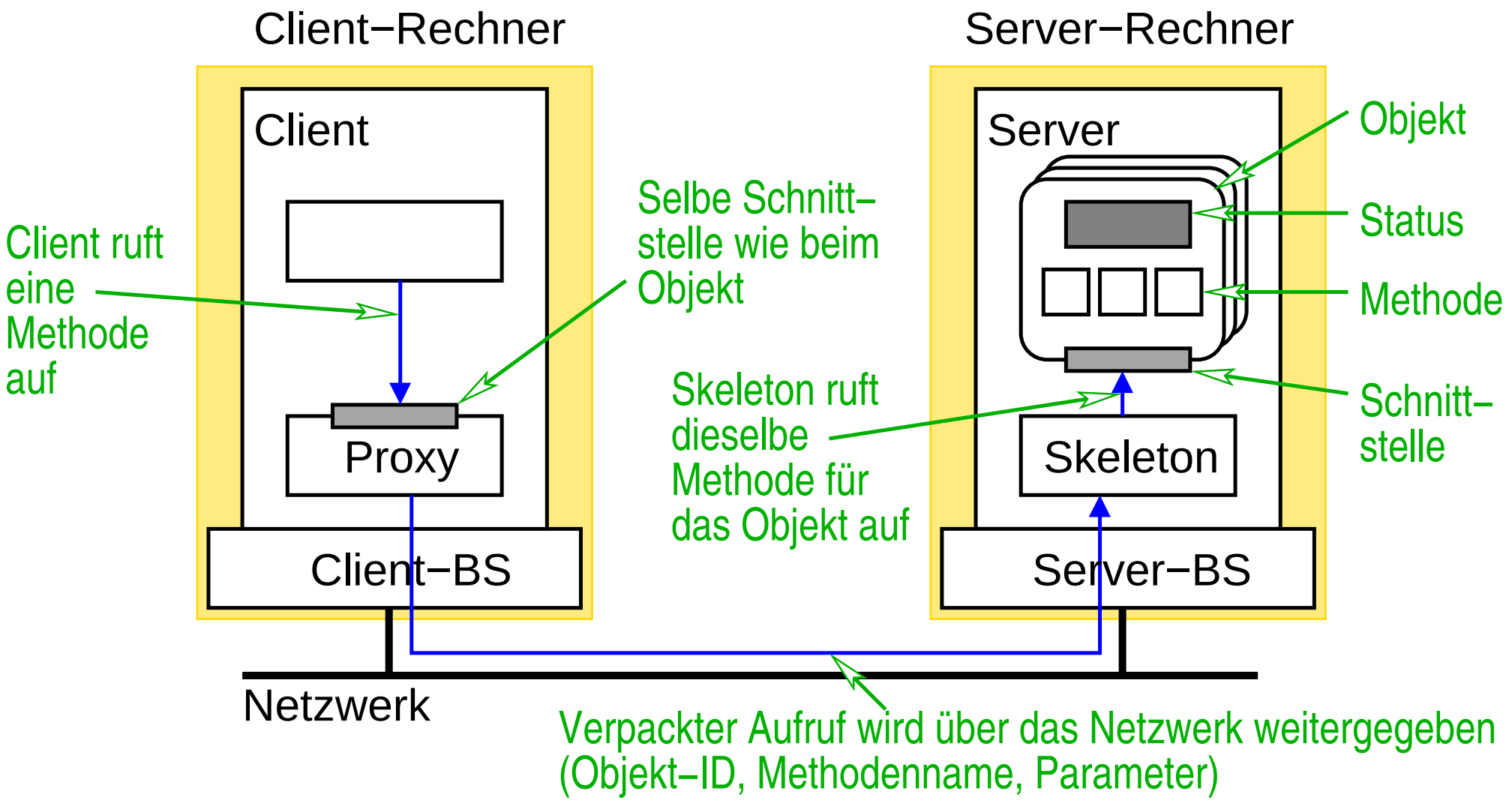


Basis von RMI: Das Proxy-Pattern

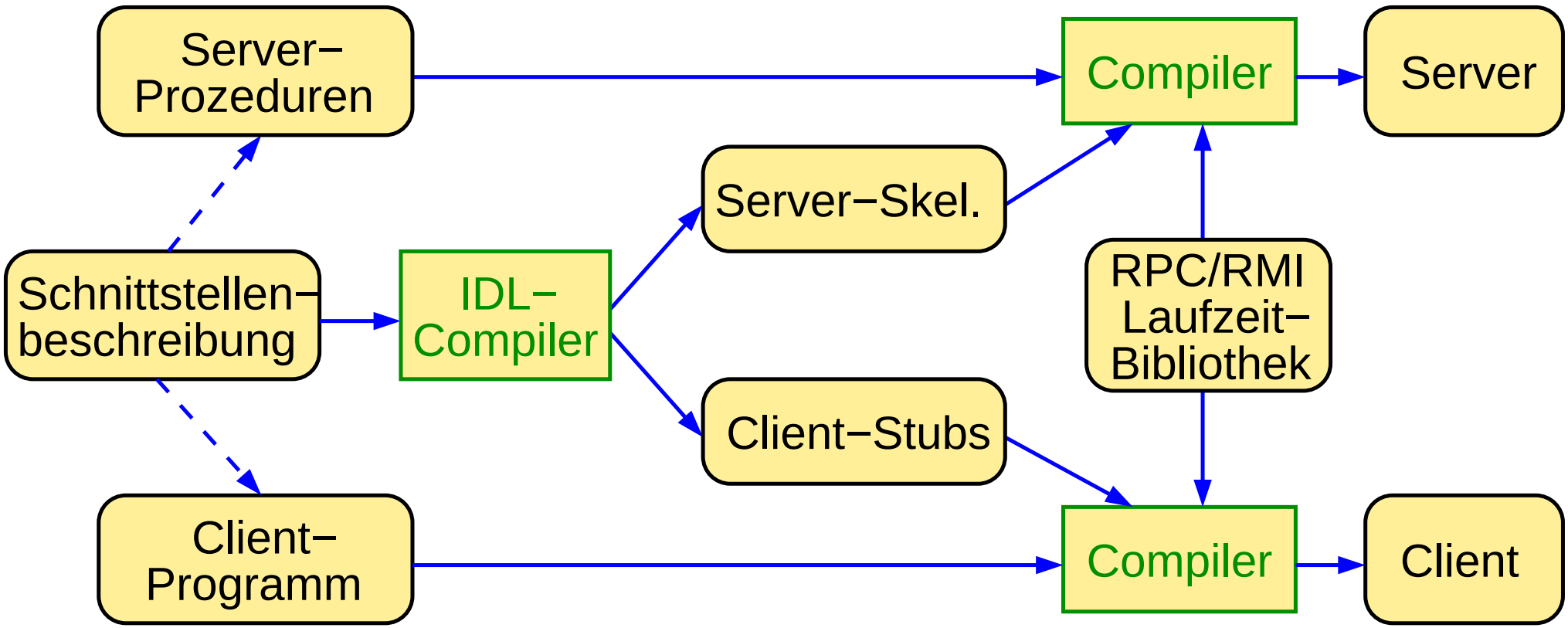
- Client arbeitet mit Stellvertreterobjekt (**Proxy**) des eigentlichen Serverobjekts
  - Proxy und Serverobjekt implementieren dieselbe Schnittstelle
  - Client kennt / nutzt lediglich diese Schnittstelle



Ablauf eines entfernten Methodenaufrufs



Erstellung eines Client/Server-Programms



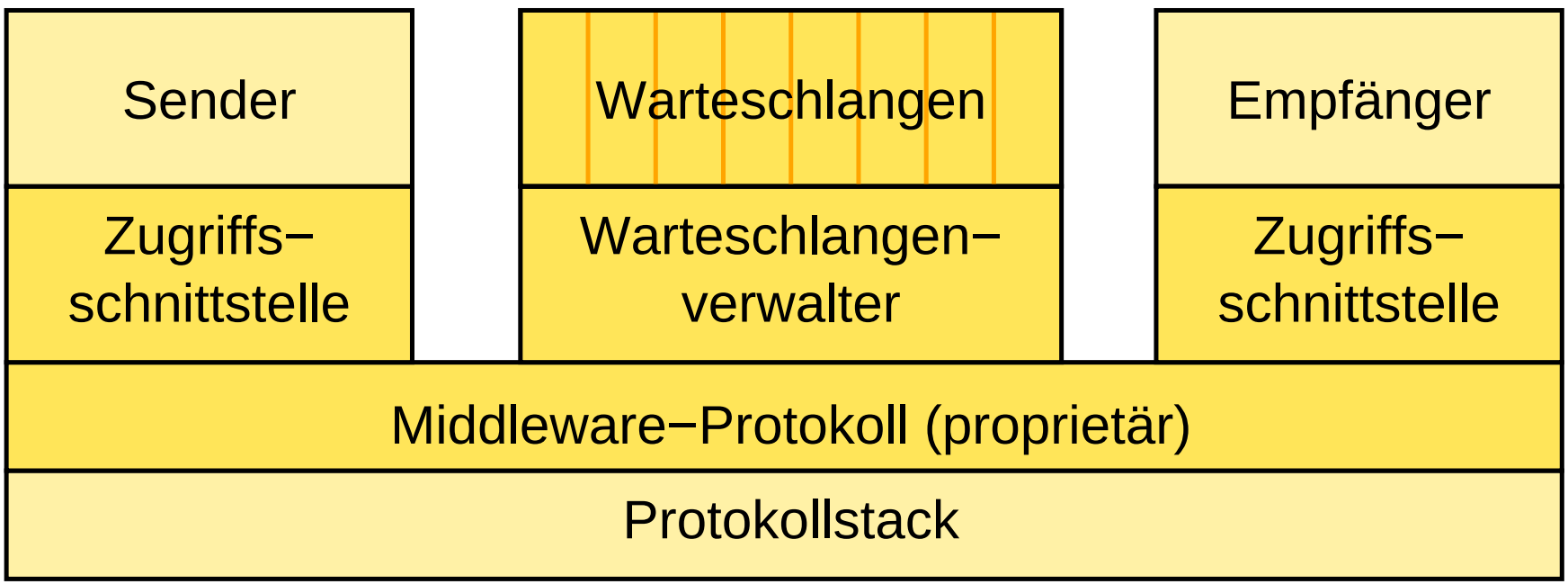
- Gilt prinzipiell für alle Realisierungen entfernten Aufrufe

2.2.3 Middleware-Technologien

- Realisieren (mindestens) eines der Programmiermodelle
  - setzen auf offene Standards / standardisierte Schnittstellen
- Entfernter Prozeduraufruf
  - SUN RPC, DCE RPC, Web Services (☞ **CSP: 10**), ...
- Entfernter Methodenaufruf
  - Java RMI (☞ **3**), CORBA (☞ **CSP: 6**), ...
- Nachrichtenorientierte Middleware-Technologien
  - MOM: *Message Oriented Middleware*, *Messaging Systeme*
  - vorwiegend für EAI
  - Java Message Service, WebSphereMQ (MQSeries), ...

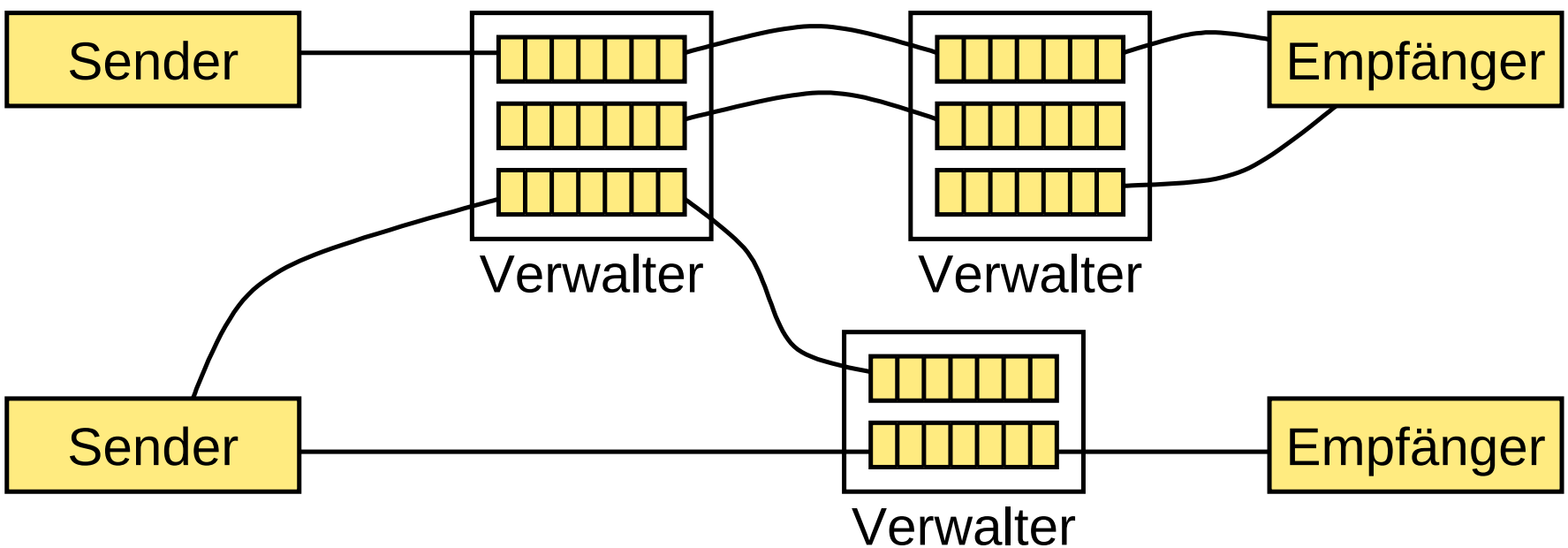
2.2.4 *Message Oriented Middleware* (MOM)

- Middleware-Technologie zum nachrichtenorientierten Modell
- Neben Nachrichtenübermittlung weitere Dienste, v.a. Warteschlangenverwaltung



Wateschlangen-Infrastruktur

- Zugriff auf Warteschlangen nur lokal möglich
  - lokal: selber Rechner oder selbes Subnetz
- Transport von Nachrichten über Subnetzgrenzen hinweg durch Warteschlangenverwalter (Router)



Varianten des Nachrichtenaustauschs

- Punkt-zu-Punkt-Kommunikation (*Point-to-Point*)
  - Kommunikation zwischen zwei festgelegten Prozessen
  - einfachstes Modell: asynchrone Kommunikation
  - Erweiterung: *Request-Reply*-Modell
    - ermöglicht synchrone Kommunikation über asynchrone Middleware
- Broadcast-Kommunikation
  - Nachricht wird an alle erreichbaren Sender versendet
  - eine Umsetzung: *Publish-Subscribe*-Modell
    - *Publisher* veröffentlichen Nachrichten zu einem Thema
    - *Subscriber* abonnieren bestimmte Themen
    - Vermittlung durch *Broker*

Beispiel: Java Message Service

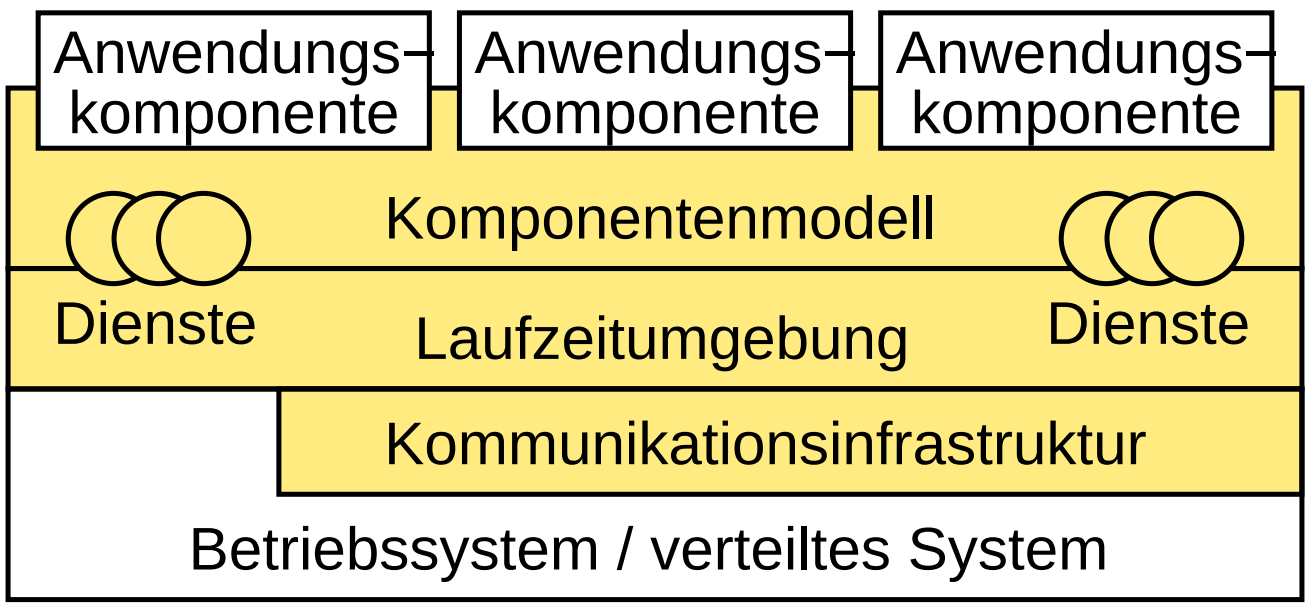
- Teil der Java Enterprise Edition (Java EE)
- Einheitliche Java-Schnittstelle für Dienste von MOM-Servern
- Unterscheidet zwei Rollen:
  - JMS-Provider: der jeweilige MOM-Server
  - JMS-Client: Sender bzw. Empfänger von Nachrichten
- JMS unterstützt:
  - asynchrone Punkt-zu-Punkt-Kommunikation
  - *Request-Reply*-Modell
  - *Publish-Subscribe*-Modell
- JMS definiert jeweils Zugriffsobjekte und Methoden

2.2.5 Zusammenfassung

- Aufgaben: Kommunikations, Behandlung der Heterogenität, Fehlerbehandlung
- Programmiermodelle:
  - nachrichtenorientiertes Modell (asynchron)
    - Basis: Nachrichtenwarteschlangen
    - Verfeinerungen:
      - *Request-Reply*-Modell (synchron)
      - *Publish-Subscribe*-Modell (Broadcast)
  - entfernte Prozedur- bzw. Methodenaufrufe
    - synchron: Anfrage und Antwort
    - generierte Stubs für *(Un-)Marshalling*

2.3 Anwendungsorientierte Middleware

- Setzt auf kommunikationsorientierter Middleware auf
- Erweitert diese um:
  - Laufzeitumgebung
  - Dienste
  - Komponentenmodell



2.3.1 Laufzeitumgebung

- Setzt auf Knoten-Betriebssystemen des verteilten Systems auf
  - Betriebssystem (BS) verwaltet Prozesse, Speicher, E/A, ...
  - liefert Basisfunktionalität
    - Starten / Beenden von Prozessen, Scheduling, ...
    - Interprozeßkommunikation, Synchronisation, ...
- Laufzeitumgebung erweitert Funktionalität des BS um:
  - verbesserte Ressourcenverwaltung
    - u.a. Nebenläufigkeit, Verbindungsverwaltung
  - Verbesserung der Verfügbarkeit
  - Verbesserte Sicherheitsmechanismen

Ressourcenverwaltung

- Geht bei Middleware über die einfache BS-Funktionalität hinaus
  - z.B. unabhängig verwaltete Hauptspeicherbereiche mit individuellen Sicherheitskriterien
  - *Pooling* von Prozessen, Threads, Verbindungen
    - werden auf Vorrat angelegt und bei Bedarf zur Verfügung gestellt
  - möglich, da Middleware spezifisch für bestimmte Klasse von Anwendungen ist
- Ziel: verbesserte Performance, Skalierbarkeit und Verfügbarkeit

Nebenläufigkeit

- Nebenläufigkeit in diesem Zusammenhang:
  - isolierte parallele Bearbeitung von Aufträgen
- Nebenläufigkeit realisierbar durch Prozesse oder Threads
  - Thread (leichtgewichtige Prozesse): Ablauf„fäden“ innerhalb von Prozessen
    - Threads im selben Prozeß teilen sich alle Ressourcen
  - Vor- und Nachteile:
    - Prozesse: hoher Ressourcenbedarf, nicht gut skalierbar, guter Schutz, bei geringer Nebenläufigkeit
    - Threads: gut skalierbar, kein gegenseitiger Schutz, bei hoher Nebenläufigkeit

Nebenläufigkeit ...

- Middleware übernimmt automatische Erzeugung / Verwaltung von Threads bei nebenläufigen Aufträgen, z.B.
  - *single-threaded*
    - nur ein Thread, sequentielle Abarbeitung
  - *thread-per-request*
    - für jede Anfrage wird ein neuer Thread erzeugt
  - *thread-per-session*
    - für jede Sitzung (Client) wird ein neuer Thread erzeugt
  - *thread pool*
    - feste Anzahl von Threads, eingehende Anfragen werden automatisch verteilt
      - spart Kosten der Threaderzeugung ein
      - beschränkt Ressourcenverbrauch

Verbindungsverwaltung

- Verbindungen hier: Endpunkte von Kommunikationskanälen
  - treten an *Tier*-Grenzen (zwischen Prozeßräumen) auf
    - z.B. Client-Server-Schnittstelle, Datenbankzugriff
  - sind im aktiven Zustand einem Prozeß / Thread zugeordnet
  - benötigen Ressourcen (Speicher, Prozessorzeit)
  - Auf- und Abbau ist aufwendig
- Zur Schonung der Ressourcen: *Pooling* von Verbindungen
  - Verbindungen werden auf Vorrat initialisiert und in Pool gestellt
  - jeder Thread / Prozeß erhält bei Bedarf eine Verbindung
  - nach Verwendung: zurückstellen in Pool

Verfügbarkeit

- Anforderung an die Anwendung,
  Umsetzung aber vor allem durch die Umgebung
- Ausfallzeiten entstehen durch
  - Ausfall einer Hard- oder Softwarekomponente
  - Überlastung einer Hard- oder Softwarekomponente
  - Wartung einer Hard- oder Softwarekomponente
- Häufige Technik zur Sicherung der Verfügbarkeit: Cluster
  - Replikation von Hard- und Software
  - Cluster tritt nach außen als eine Einheit auf
  - zwei Arten: Fail-over Cluster / Load-balancing Cluster

Sicherheit

- Verteilte Anwendungen sind durch ihre Verteiltheit angreifbar
- Middleware unterstützt unterschiedliche Sicherheitsmodelle
- Sicherheitsanforderungen:
  - **Authentifizierung**:
    - stellt Identität des Anwenders / einer Komponente sicher
    - z.B. durch Paßwort-Abfrage (bei Benutzer) oder krypto-graphische Techniken u. Zertifikate (bei Komponenten)
  - **Autorisierung**
    - Festlegung von Zugriffsrechten für Benutzer auf konkrete Dienste
      - bzw. auch feiner: Methoden und Attribute
    - Überprüfung setzt sichere Authentifizierung voraus

Sicherheit ...

- Sicherheitsanforderungen ...:
  - **Vertraulichkeit**
    - Abhören von Daten während der Übertragung im Netz ist nicht möglich
    - Technik: Verschlüsselung
  - **Integrität**
    - Übertragene Daten können nicht unbemerkt verändert werden
    - Techniken: kryptographische Prüfsumme (*Message Digest, Fingerprint*), digitale Signatur
      - digitale Signatur stellt auch Authentizität des Absenders sicher

Sicherheit ...

- Sicherheitsmechanismen:
  - Verschlüsselung
    - symmetrisch (z.B. IDEA, AES)
      - selber Schlüssel zum Ver- und Entschlüsseln
    - asymmetrisch (*Public-Key*-Verfahren, z.B. RSA)
      - öffentlicher Schlüssel zum Verschlüsseln
      - privater Schlüssel zum Entschlüsseln
  - Digitale Signatur
    - sichert Integrität einer Nachricht und Authentizität des Senders sowie Verbindlichkeit
  - Zertifikat
    - beglaubigt Zusammengehörigkeit von öffentlichem Schlüssel und Person (bzw. Komponente)

Verteilte Systeme

SS 2013

06.05.2013

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 6. Mai 2013

Namensdienst (Verzeichnisdienst) (☞ 4)

- Veröffentlichung von verfügbaren Diensten
  - im Intranet oder Internet
- Zuordnung von Namen zu Referenzen (Adressen)
  - Name dient als eindeutiger / unveränderlicher Identifikator
  - Client kann über den Namen die Adresse eines Servers erfragen
    - Adresse kann sich z.B. bei Neustart ändern
  - Ziel: Entkopplung von Client und Server
- Beispiele: JNDI, RMI Registry, CORBA Interoperable Naming Service, UDDI Registry, LDAP Server, ...

Sitzungsverwaltung

- In interaktiven Systemen: jeder Instanz eines Clients wird eine eigene **Sitzung** (***Session***) zugeordnet
  - gelöscht beim Beenden des Clients oder beim Abmelden
- Sitzung speichert alle relevanten Daten (im Hauptspeicher)
  - z.B. Kennung des Anwenders, Browsertyp, „Warenkorb“, ...
  - Speicherung im Server oder im Client
  - transiente Daten: werden am Ende der Sitzung gelöscht
  - persistente Daten: werden am Ende der Sitzung auf Datenträger geschrieben (Datenbank)
- Middleware realisiert / unterstützt die Zuordnung von Anfragen zu Sitzungen (oft transparent)
  - z.B. Cookies, HTTPSessions, Session Beans, ...

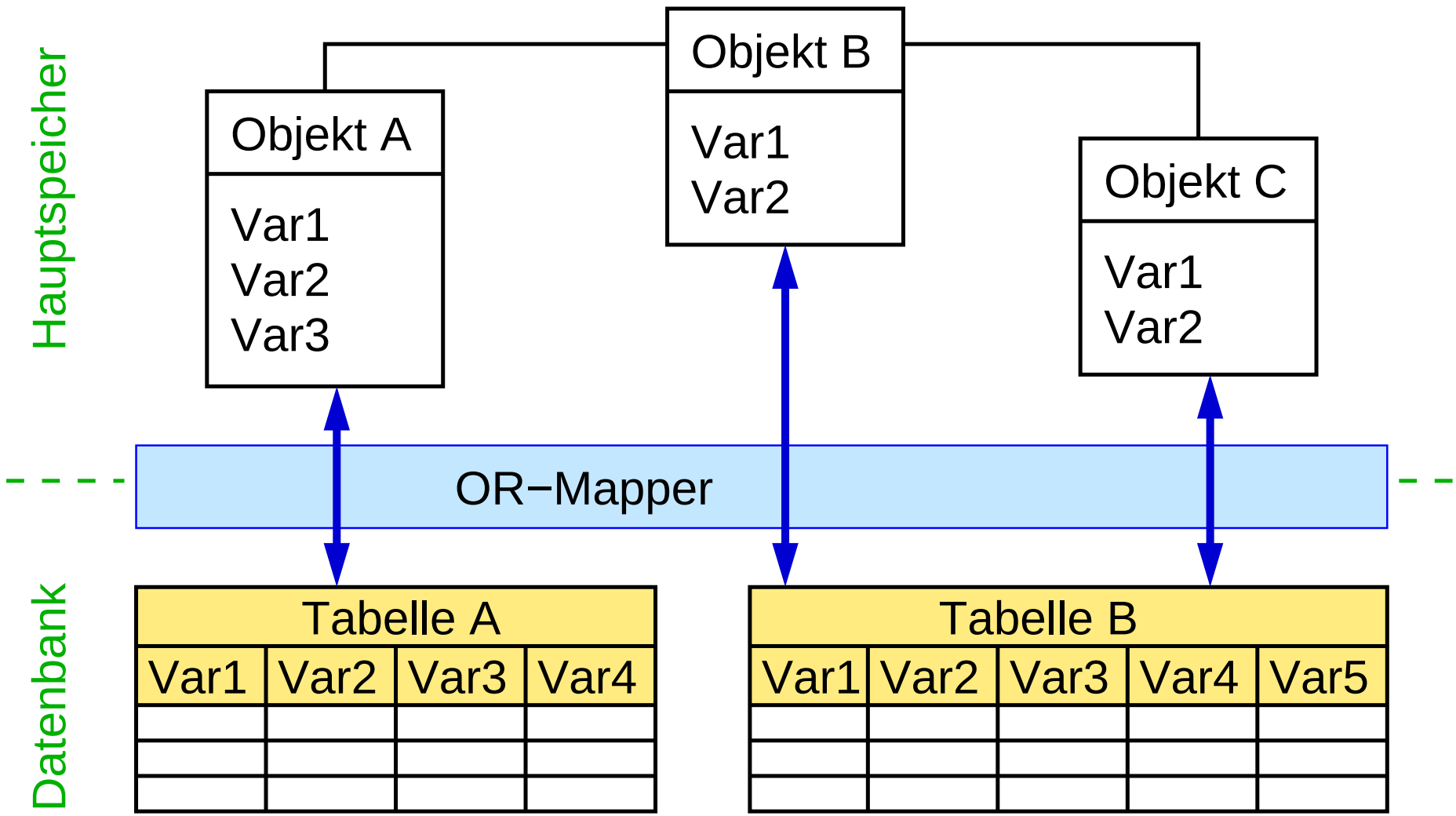
Transaktionsverwaltung (☞ 7.4)

- Dienst für interaktive, datenzentrierte Anwendungen
  - Konsistenz / Integrität der Daten ist wichtig
  - d.h. gesamter (ggf. verteilter) Datenbestand muß einen in sich gültigen Zustand repräsentieren
- Typischer Ablauf in Anwendungen:
  1. Client fordert Daten an
  2. Client verändert die Daten
  3. Client fordert das Rückschreiben der Daten an
  - Problem: die Schritte 1 - 3 könnten von zwei Clients genau gleichzeitig durchgeführt werden
- Transaktionsverwaltung erlaubt Durchführung einer Folge von Aktionen als atomare Einheit

Persistenzdienst

- Persistenz: Gesamtheit aller Maßnahmen zur dauerhaften Speicherung von Hauptspeicher-Daten
- Persistenzdienst: intelligente Schnittstelle zur Datenbank
  - in Middleware integriert oder als eigenständige Komponente
  - neben Transaktionsverwaltung wichtigster Dienst für daten-zentrierte Anwendungen
- Häufigste Art: objektrelationaler Mapper (OR-Mapper)
  - bildet Objekte im Hauptspeicher auf Tabellen in relationaler Datenbank ab
  - Regeln werden von Anwendungsentwickler vorgegeben

Persistenzdienst ...



2.3.3 Komponentenmodell

- Komponenten: „große“ Objekte zur Strukturierung von Anwendungen
- Ein Komponentenmodell definiert:
  - Komponentenbegriff
    - Struktur und Eigenschaften der Komponenten
    - vorgeschriebene und optionale Schnittstellen
  - Schnittstellenverträge
    - wie interagieren Komponenten untereinander und mit der Laufzeitumgebung?
  - Komponenten-Laufzeitumgebung
    - Verwaltung des Lebenszyklus der Komponenten
    - implizite Bereitstellung von Diensten: Komponente teilt nur Anforderungen mit (z.B. Persistenz)

2.3.4 Middleware-Technologien

- *Object Request Broker* (ORB)
  - verteilte Objekte, entfernte Methodenaufrufe
  - Vielzahl an Diensten, nur grundlegende Laufzeitumgebung
  - Beispiel: CORBA
- *Application Server*
  - Fokus: Unterstützung der Anwendungslogik (*Middle-Tier*)
  - Dienste, Laufzeitumgebung und Komponentenmodell
  - heute nur noch als Teil einer Middleware-Plattform
- Middleware-Plattformen
  - Erweiterung von Appl. Servern: Unterstützung aller *Tiers*
    - neben verteilten Anwendungen auch EAI
  - Beispiele: Java EE / EJB, .NET / COM, CORBA 3.0 / CCM

2.3.5 Zusammenfassung

Anwendungsorientierte Middleware

- Laufzeitumgebung
  - Ressourcenverwaltung, Verfügbarkeit, Sicherheit
- Dienste
  - Namensdienst, Sitzungsverwaltung, Transaktionsverwaltung, Persistenzdienst
- Komponentenmodell
  - Komponentenbegriff, Schnittstellenverträge, Laufzeitumgebung