Christian Sillaber

QE Research Group
Institut für Informatik
Univeristät Innsbruck

Client-Server-Systeme

1 Intro

2 Einführung / Wiederholung

3 Herausforderungen

4 Architektur verteilter Systeme

5 Architekturmuster

6 Grundlagen der Implementierung
- Socket Programmierung
- Remote Method Invocation (RMI)

Literatur

- George Coulouris et al., Distributed Systems: Concepts and Design, 5th Edition, 2011.
- Elliotte Rusty Harold, Java Network Programming, 4th Edition, 2014.

Defintionen

Ein verteiltes System

- besteht aus mehreren Komponenten welche
- auf unterschiedliche miteinander vernetzte Rechner verteilt sind
- und ihre Aktivitäten ausschließlich durch die Übertragung von Nachrichten koordinieren.

Charakteristika

- Nebenläufigkeit
- Kein globaler Zeitgeber
- Unabhängige Fehler

Was kann geteilt werden?

Hardware, Daten, Funktionen und Dienste

Recap: Begriffe

Service

Ein individueller Teil eines Computer Systems, welcher eine Sammlung verbundener Resourcen verwaltet und deren Funktionalität für Benutzer und Anwendungen bereit stellt.

Server

Ein laufendes Programm (oder ein Prozess) das eingehende Anfragen von Programmen annimmt und entsprechend darauf reagiert

Client

Ein laufendes Programm (oder ein Prozess) das Service-Anfragen an einen Server stellt

Herausforderungen

- Heterogenität
- Offene Systeme
- Sicherheit
- Skalierbarkeit
- Fehlerbehandlung
- Nebenläufigkeit
- Transparenz
- Quality of Service

Heterogenität

Warum?

- Netzwerke
- Hardware
- Betriebssysteme
- Programmiersprachen
- Verschiedene Hersteller / Tools

Middleware

- Software Layer stellt abstrahierte Programmier-Schnittstellen bereit.
- Verbirgt die Heterogenität der darunterliegenden Netzwerke, Hardware, Betriebssysteme und Programmiersprachen.

Offene Systeme

- Ein offenes System veröffentlicht seine wesentlichen Schnittstellen.
- Ein offenes verteiltes System basiert auf der Bereitstellung von einheitlichen Kommunikationsmechanismen und veröffentlichten Schnittstellen, um auf geteilte Ressourcen zuzugreifen.
- Offene verteilte Systeme können aus heterogenen Hard- und Software-Bestandteilen (auch unterschiedlicher Hersteller) aufgebaut sein.
- Um die Funktionsweise des Gesamtsystems zu garantieren, muss jede Komponente den vereinbarten Standards entsprechen und ausreichend getestet und verifiziert sein.

Sicherheit

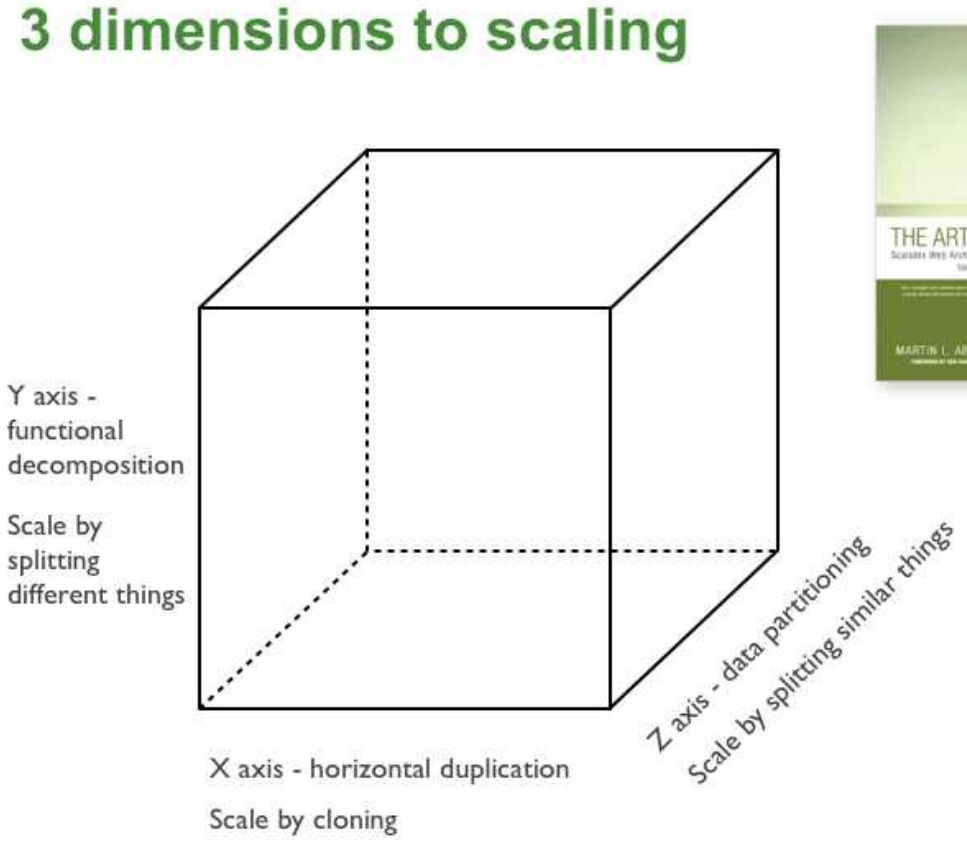
Informationssicherheit verfolgt folgende (grundlegenden) Schutzziele:

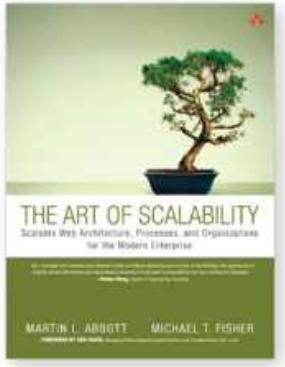
- Vertraulichkeit: Schutz vor Offenlegung an unautorisierte Benutzer.
- Integrität: Schutz gegen unbefugter Manipulation oder Korruption der Daten.
- Verfügbarkeit: Schutz vor Beeinträchtigung des Zugriffs auf Ressourcen.

Skalierbarkeit

Folgende Herausforderungen müssen beim Entwurf von skalierbaren verteilten Systemen berücksichtigt werden:

- Kosten der physikalischen Ressourcen
- Performance-Verluste
- Ausreichende Verfügbarkeit von Ressourcen
- Vermeidung von Performance-Bottlenecks





Fehlerbehandlung

Fehler treten in verteilten Systemen partiell auf. Daher wird die Fehlerbehandlung entsprechend erschwert, mögliche Techniken sind:

- **Fehlererkennung:** z.B. Einsatz von Prüfsummen bei Datenübertragung
- **Fehlermaskierung:** z.B. erneutes Senden von Nachrichten, die nicht korrekt übertragen wurden
- **Fehlertoleranz:** Nicht erkannte oder nicht maskierbare Fehler müssen ebenfalls behandelt werden, z.B. bei Nichtverfügbarkeit von Ressourcen
- **Erholung von Fehlern:** z.B. „roll back“ nach einem Server-Crash
- **Redundanz:** z.B. durch Replikation von kritischen Ressourcen

Nebenläufigkeit

- Mehrere Clients können gleichzeitig eine Ressource anfragen oder auf sie zugreifen
- Services und Applikationen erlauben meistens die nebenläufigen Abarbeitung von mehreren Client-Anfragen
- Daher: Operationen auf geteilte Ressourcen müssen entsprechend synchronisiert werden um die Integrität und Konsistenz der Daten zu gewährleisten

Transparenz

Benutzern und Anwendungsentwicklern wird das verteilte System als Ganzes (anstatt einer Sammlung an sich unabhängiger Komponenten) präsentiert.
Mehrere Aspekte von Transparenz können unterschieden werden:

- Access
- Location
- Nebenläufigkeit
- Replikation
- Fehler
- Mobilität
- Performanz
- Skalierung

Quality of Service

Die wesentlichsten nicht-funktionalen Eigenschaften von Systemen, welche die bereitgestellte Service-Qualität beeinflussen sind:

- Zuverlässigkeit
- Sicherheit
- Performanz
- Anpassbarkeit

Architekturmodelle für verteilte Systeme

- Die Architektur eines Systems beschreibt dessen Struktur auf Basis einzeln spezifizierter Komponenten und deren Interaktionen
- Middleware-Plattformen können die Implementierung unterschiedlicher Arten verteilter Systeme mit unterschiedlichen Architektur-Stilen unterstützen.

Wesentliche Unterscheidungselemente der Architektur verteilter Systeme

- Kommunikationseinheiten
- Kommunikationsparadigma
- Rollen und Verantwortlichkeiten
- Placement

Kommunikationseinheiten

- Welche Einheiten kommunizieren in einem verteilten System und wie ist die Kommunikation zwischen ihnen geregelt
- **Allgemein:** Kommunikationseinheiten sind Prozesse
- **Aber:** Eine stärkere problem-orientierte Abstraktion wird angestrebt
- Anstatt Prozessen werden Objekte, Komponenten, Web Services als Kommunikationseinheiten eingesetzt

Kommunikationsparadigma

Definieren wie Kommunikationseinheiten in einem verteilten System miteinander kommunizieren

Inter-Prozess Kommunikation

Bezieht sich auf Low-Level Unterstützung für die Kommunikation zwischen einzelnen Prozessen in verteilten Systemen (z.B. Socket-Programmierung)

Remote Invocation

Bezieht sich auf eine Reihe von Techniken zur Ausführung von entfernten Operationen/Methoden basierend auf einer Zwei-Wege-Kommunikation zwischen Kommunikationseinheiten

- Request-Reply Protokolle: HTTP, ICMP, etc.
- Remote Procedure Call: CORBA, RMI, etc.

Kommunikationsparadigma

Indirekte Kommunikation

Die Kommunikation erfolgt nicht direkt zwischen den Kommunikationseinheiten, sondern über einen dritten Teilnehmer
Starke Entkopplung zwischen Sender und Empfänger, d.h.

- Sender müssen nicht wissen an wen sie senden (Space uncoupling)
- Sender und Empfänger müssen nicht zur selben Zeit existieren (Time uncoupling)

Message Queues

Punkt-zu-Punkt Service wobei ein Producer-Prozess Nachrichten an die Queue sendet und Consumer-Prozesse diese aus der Queue erhalten. z.B.: Apache Kafka

Kommunikationsparadigma

Tuple Spaces

Prozesse speichern beliebige Elemente strukturierter Daten, sogenannte Tuples, in einen persistenten Tuple-Speicher, andere Prozesse können diese Tuple aus dem Speicher laden oder löschen.
$\rightarrow$ `JavaSpaces`

Distributed Shared Memory

Abstraktion damit sich unterschiedliche Prozesse ohne gemeinsamen Speicher Daten teilen können, der Zugriff erfolgt dabei so als wären die Daten lokal. z.B.: Hazlecast, Terracotta

Rollen und Verantwortlichkeiten

Welche Rollen und Verantwortlichkeiten übernehmen Kommunikationseinheiten, wenn Sie miteinander kommunizieren um eine Aufgabe zu erfüllen?

Client-Server

Client-Prozesse interagieren mit einzelnen Server-Prozessen auf potenziell unterschiedlichen Computern um auf verteilte Ressourcen zuzugreifen, die vom Server-Prozess verwaltet werden.

Peer-to-Peer

Alle Prozesse, die in der Erfüllung einer Aufgabe eingebunden sind, erfüllen gleiche Rollen, interagieren kooperativ als sogenannte Peers ohne Unterscheidung zwischen Client- und Server-Prozessen

Placement

- Wie werden Kommunikationseinheiten auf die zugrunde liegende physikalisch verteilte Infrastruktur abgebildet?
- Das Placement ist für viele Eigenschaften verteilter Systeme entscheidend:
  - Performance
  - Zuverlässigkeit (Reliability)
  - Sicherheit
- Ein optimales Placement muss die Kommunikations-Pattern zwischen den Entitäten, die Verlässlichkeit einzelner Computer sowie deren Auslastung und die Verbindungsqualität zwischen beteiligten Computern berücksichtigen

Placement Strategien

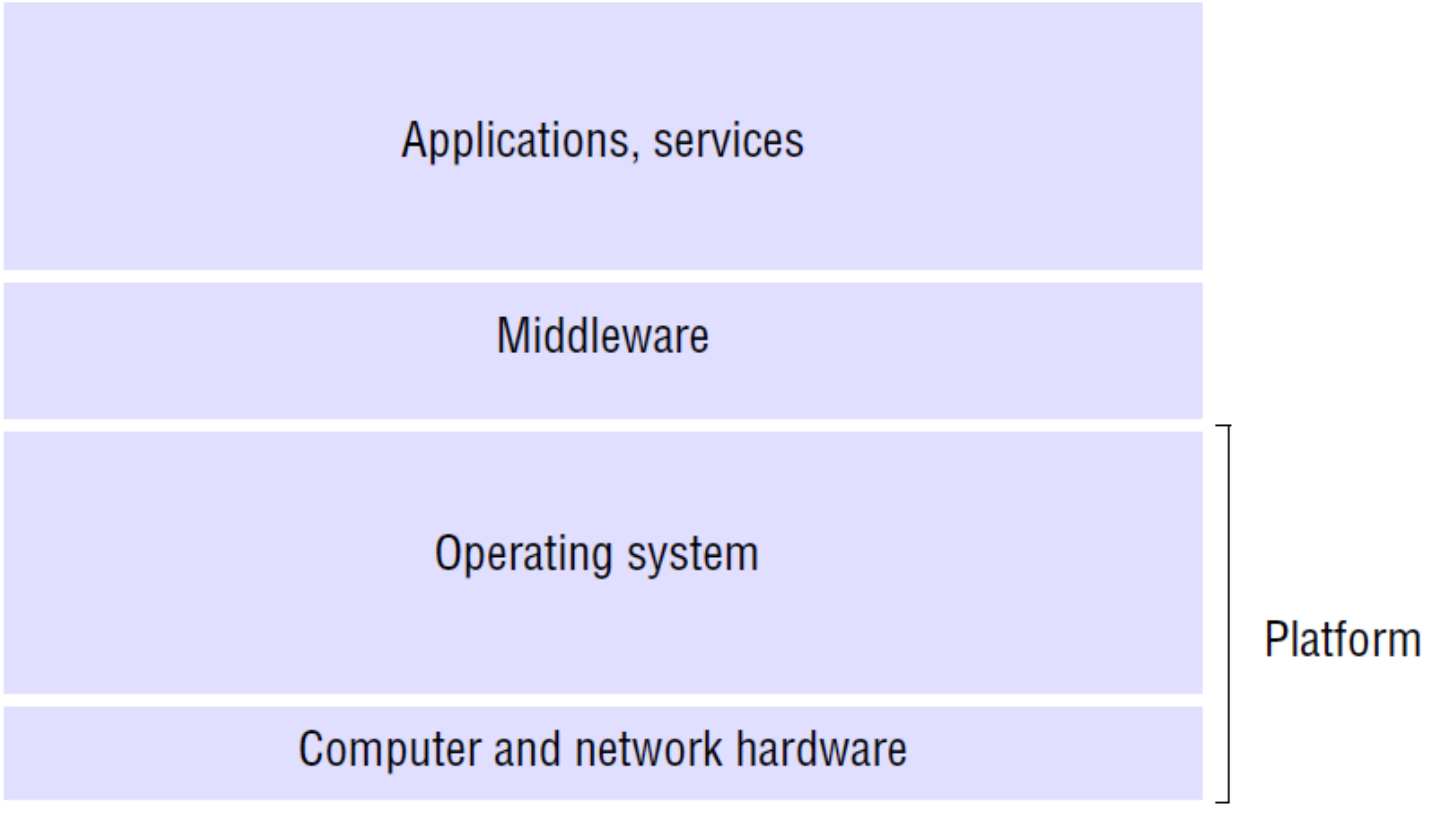
Verteilung

Services auf mehrere Server verteilen: Ein Services wird durch mehrere Server-Prozesse realisiert, die auf unterschiedlichen Computern ausgeführt werden

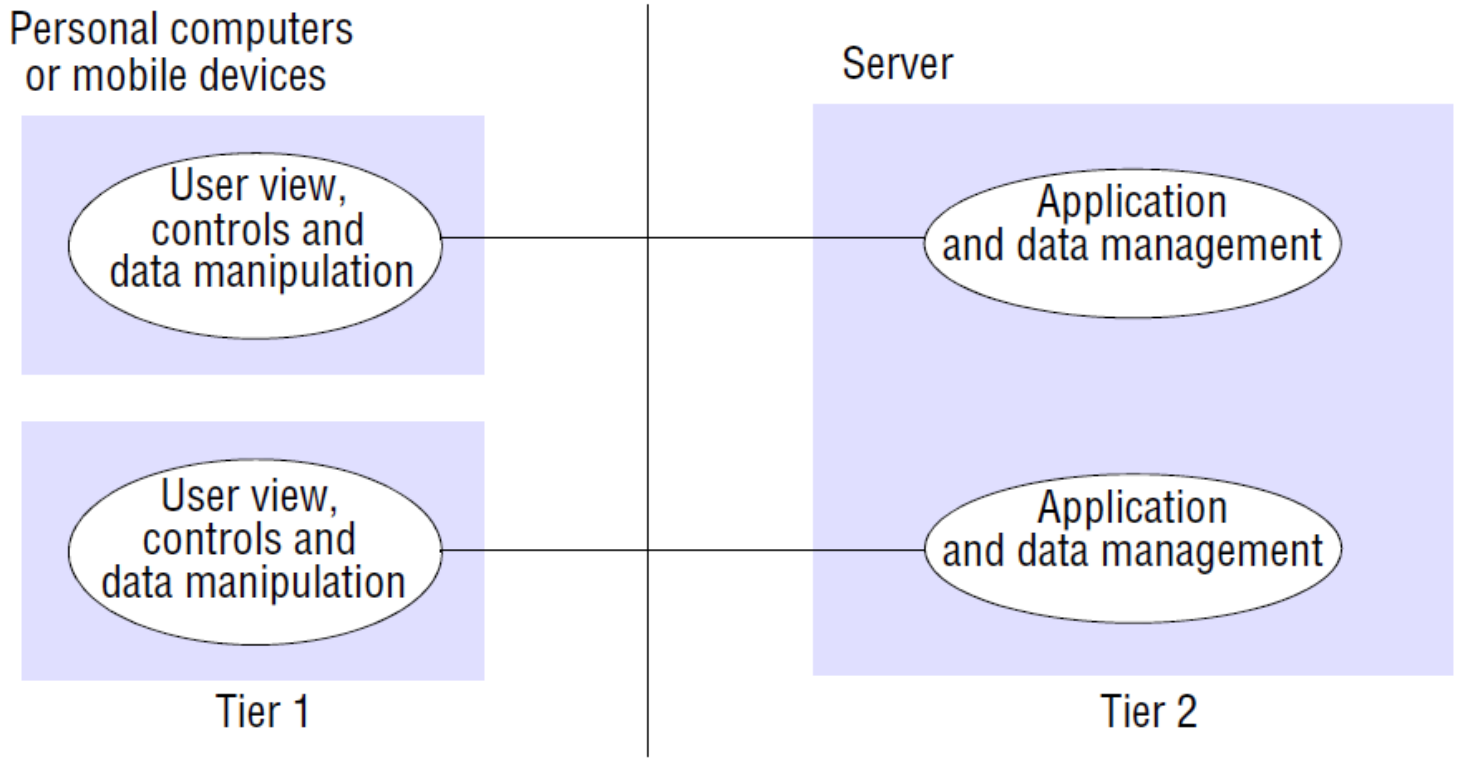
Caching

Objekte werden von einem Cache bereit gestellt, sofern dieser eine aktuelle Kopie gespeichert hat

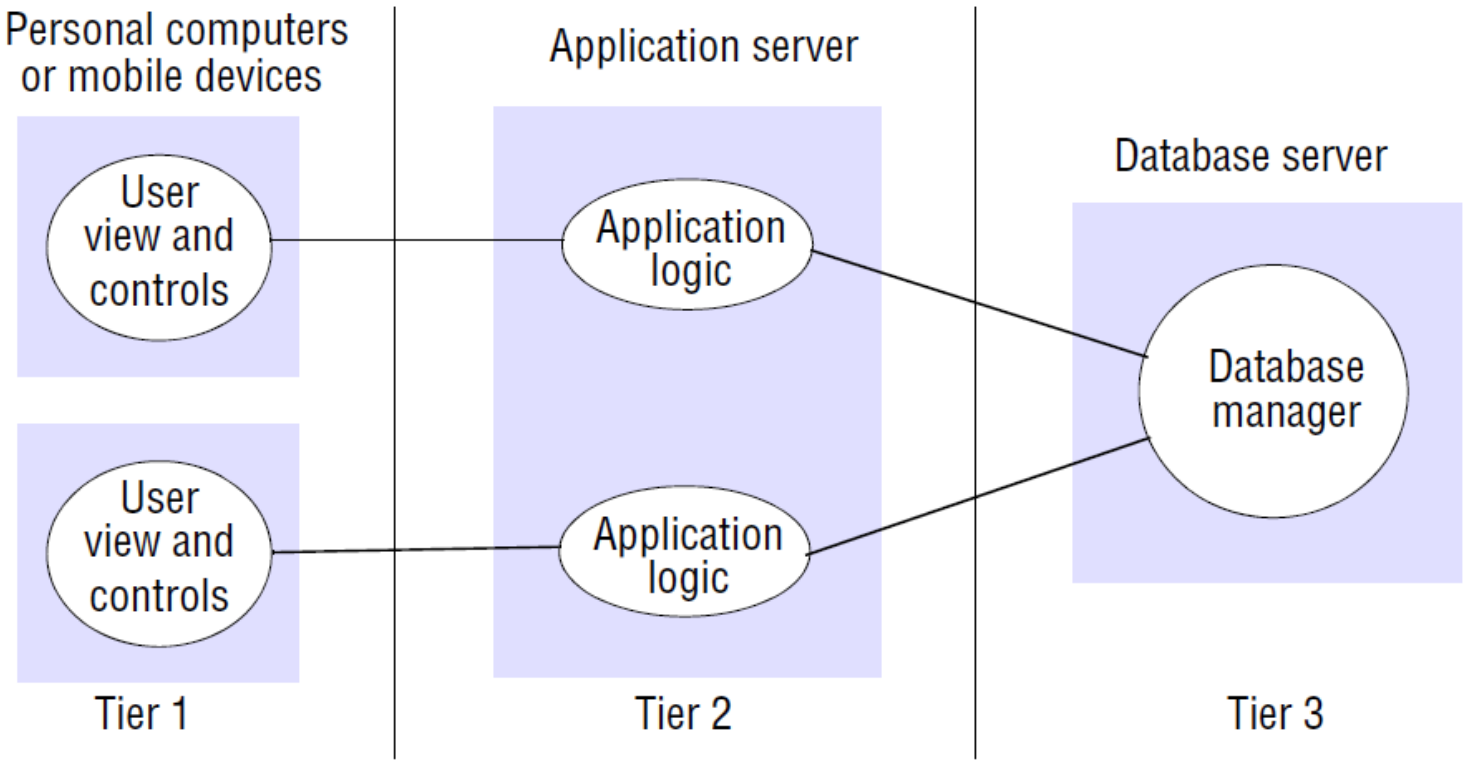
Software und Hardware Service Layer in verteilten Systemen



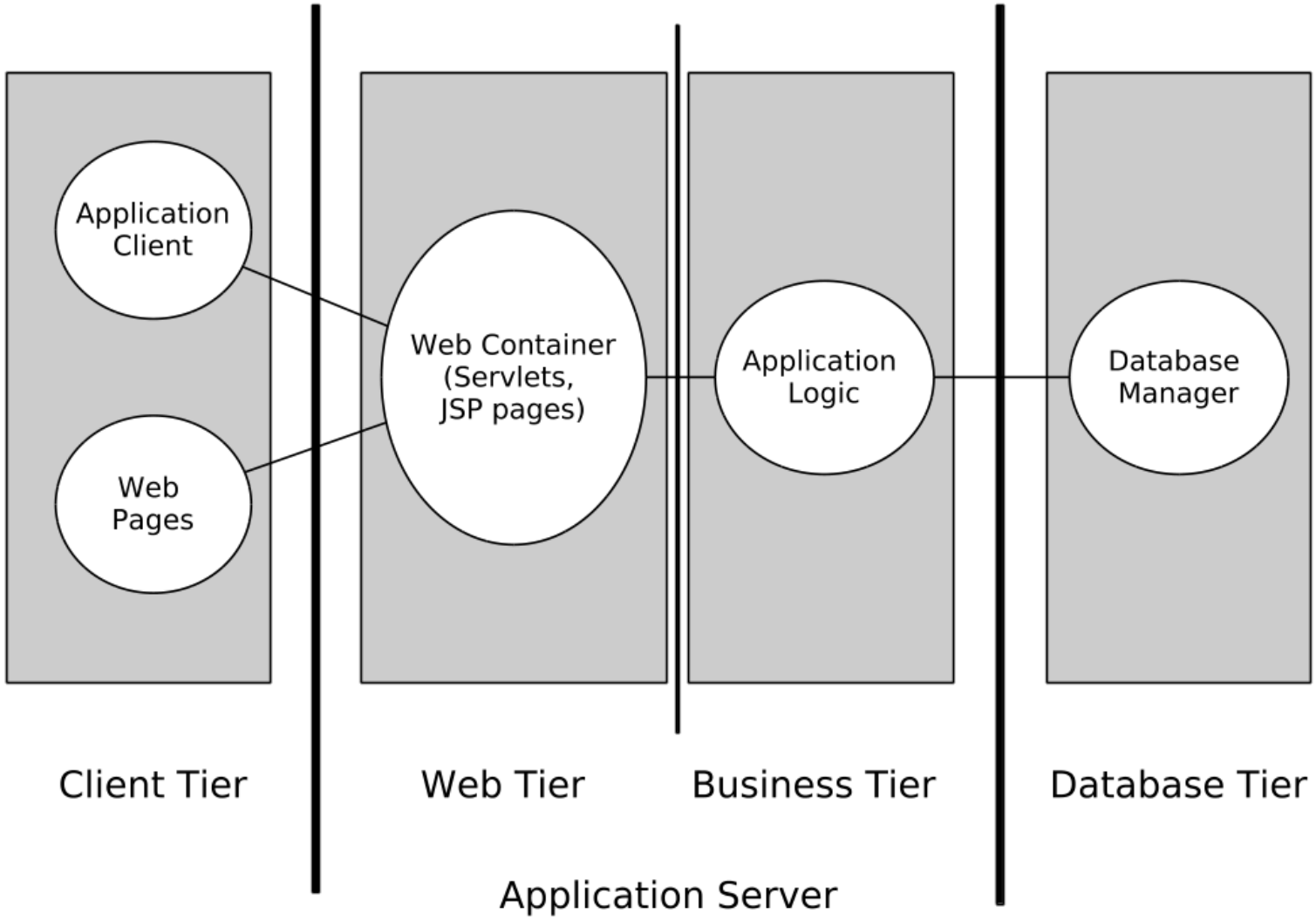
Two Tier Architektur



Three Tier Architektur



Multi Tier Architektur



Weitere Architekturmuster

Thin Clients

Verlagerung der Komplexität von den Geräten der End-Anwender zu Services im Internet (z.B: Cloud Computing). Thin Client bezeichnet dabei einen Software Layer der eine Window-basiertes User Interface für lokale Nutzer bereit stellt, das Zugriff auf entfernte Services bereit stellt.

Proxy

Für Location-Transparenz (vgl. RMI), implementiert Interface von entfernten Objekten für den lokalen Zugriff. Kann auch für Replikation und Caching eingesetzt werden.

Dienst-Vermittlung

Service Broker vermittelt passende Services eines Service Providers an einen Service Requester.

Drei beispielhafte Ansätze zur Entwicklung verteilter Systeme

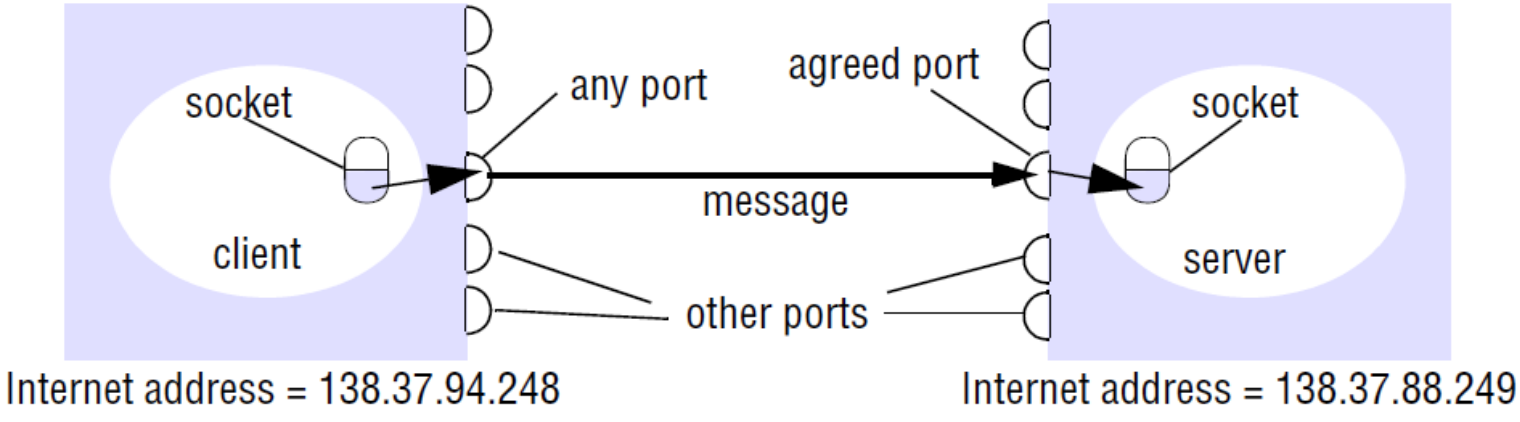
- **Socket-Programmierung**
  - Low-Level Implementierung
  - Vorstellung relevanter Java-Klassen
- **Remote Method Invocation (RMI)**
  - Grobkonzept, Beispielimplementierung
- **Web Services mit dem Play Framework**
  - REST, MVC Pattern in Web-Applikationen

Socket Programmierung

Interprozess Kommunikation

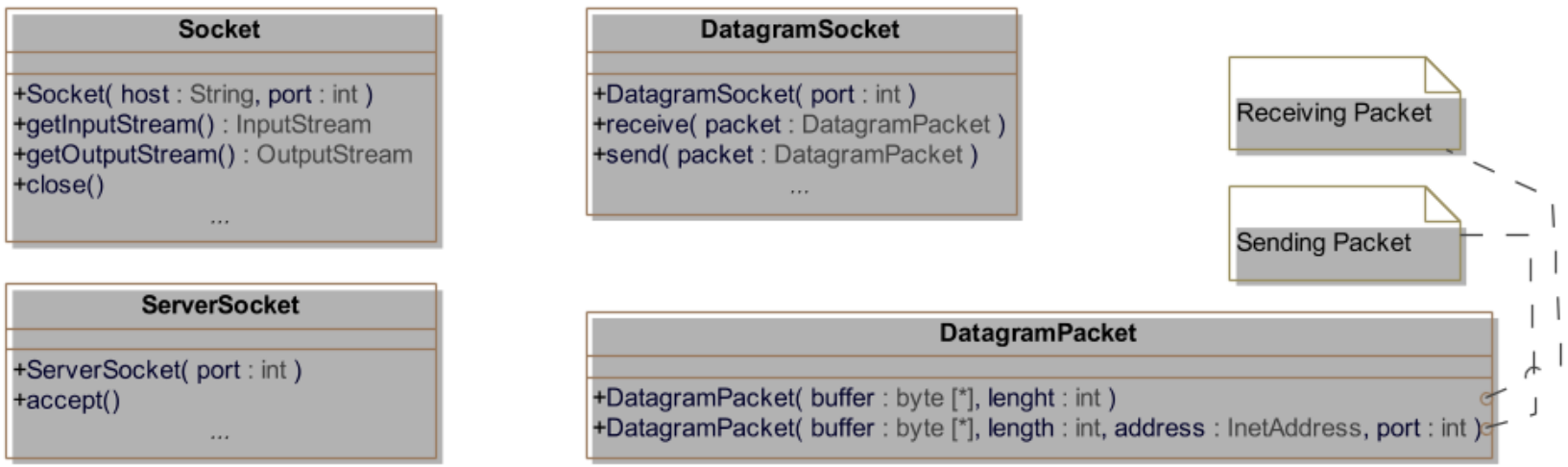
Nachricht wird vom Socket eines Prozesses zum Socket eines anderen Prozesses übertragen

- Um Nachrichten zu empfangen, muss der Socket des empfangenden Prozesses an einen lokalen Port und die zugehörige Internet Adresse des Computers gebunden werden
- Nachrichten können nur von Prozessen empfangen werden, deren Sockets mit dem entsprechenden Port und Adresse assoziert sind



Sockets in Java

- Alle relevanten Klassen sind im Package java.net gekapselt
- Unterstützung für:
  - TCP Sockets: Socket und ServerSocket
  - UDP Sockets: DatagramSocket und DatagramPacket



Sockets in Java

```
1  import java.net.ServerSocket;
2  import java.net.Socket;
3  import java.io.IOException;
4  public class SimpleSocketServer {
5      public static final int SIMPLE_SOCKET_SERVER_PORT = 10123;
6      public static void main(String[] args) {
7          int port = SIMPLE_SOCKET_SERVER_PORT;
8          ServerSocket server = null;
9          try {
10             server = new ServerSocket(port);
11         } catch (IOException ex) {System.err.println("Error registering
               server socket");}
12         while(server != null) {
13             System.out.println("Waiting for client ...");
14             Socket client;
15             try {
16                 client = server.accept();
17                 System.out.println("Client from " +
                       client.getInetAddress() + " connected");
18                 client.close();
19             } catch (IOException ex) {
20                 System.err.println("Error connecting to client");
21             }
22         }
23         System.out.println("terminated.");
24     }
25 }
```

Sockets in Java

```
1  import java.net.Socket;
2  import java.io.IOException;
3
4  public class SimpleSocketClient {
5
6      public static void main(String[] args) {
7          System.out.println("Trying to connect with server ...");
8          Socket server;
9          try {
10             server = new Socket("localhost",
                   SimpleSocketServer.SIMPLE_SOCKET_SERVER_PORT);
11             System.out.println("Connectd to server " +
                   server.getInetAddress());
12             server.close();
13         } catch (IOException ex) {
14             System.err.println("Error connecting to server");
15         }
16         System.out.println("terminated.");
17     }
18
19 }
```

Sockets in Java: Serverseite

Daten senden + empfangen:

```
1   while(server != null) {
2   System.out.println("Waiting for client ...");
3   Socket client;
4   try {
5       client = server.accept();
6       System.out.println("Client from " + client.getInetAddress() + "
            connected");
7       DataOutputStream out =
8               new DataOutputStream(new BufferedOutputStream(
9                   client.getOutputStream()));
10      String response = new Date().toString();
11      out.writeUTF(response);
12      System.out.println("Told client " + response);
13      out.flush();
14      out.close();
15      client.close();
16  } catch (IOException ex) {
17      System.err.println("Error connecting to client");
18  }
```

Sockets in Java: Clientseite

Daten senden + empfangen:

```
1  Socket server;
2  try {
3      server = new Socket("localhost", PORT);
4      System.out.println("Connectd to server " + server.getInetAddress());
5      DataInputStream in = new DataInputStream(
6              new BufferedInputStream(server.getInputStream()));
7      String serverResponse = in.readUTF();
8      System.out.println("Server said " + serverResponse);
9      in.close();
10     server.close();
11 } catch (IOException ex) {
12     System.err.println("Error connecting to server");
13 }
```

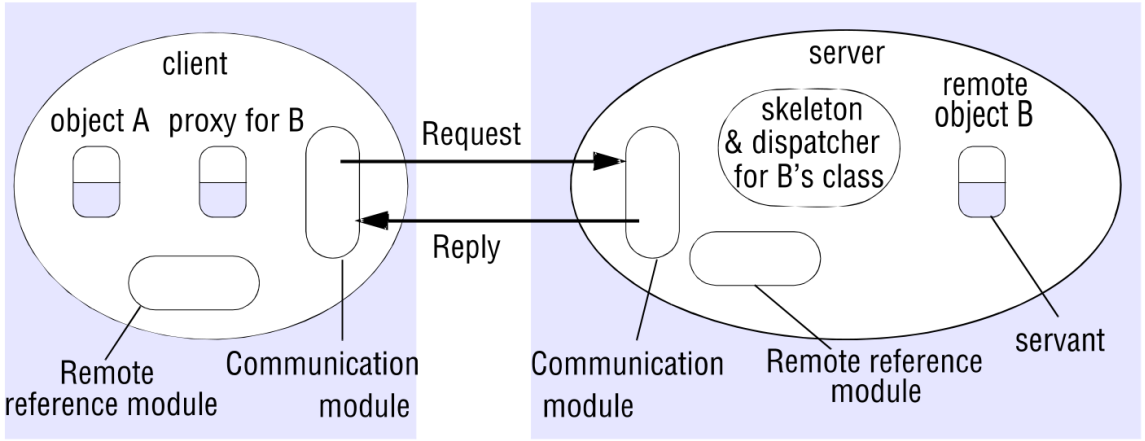
Remote Method Invocation (RMI)

Remote Method Invocation

Erlaubt es einem Objekt, das in einer JVM läuft, die Methoden eines Objekts aus einer anderen JVM asuzuführen.

- Remote-Aufruf wird abstrahiert $\rightarrow$ wie lokaler Aufruf
- Ein Server stellt Remote Objects zur Verfügung
- Client hält Remote Object Referenz und ruft damit Methoden des Remote Objects auf
- Remote Object Referenzen werden über einen eigene Registry bereit gestellt:
  - Server registriert Remote Objects mit eindeutigem Namen
  - Client kann damit gewünschte Remote Object Referenz anfragen

RMI - Funktionsweise



Proxy

1 Bildet Anfragenachricht (inkl. Marshalling)

8 Empfängt Ergebnisnachricht (inkl. Unmarshalling)

9 Leitet Ergebnis weiter

Skeleton

4 Empfängt Anfragenachricht (inkl. Unmarshalling)

5 Ruft Methode im Remote Object auf

6 Bildet Ergebnisnachricht (inkl. Marshalling)

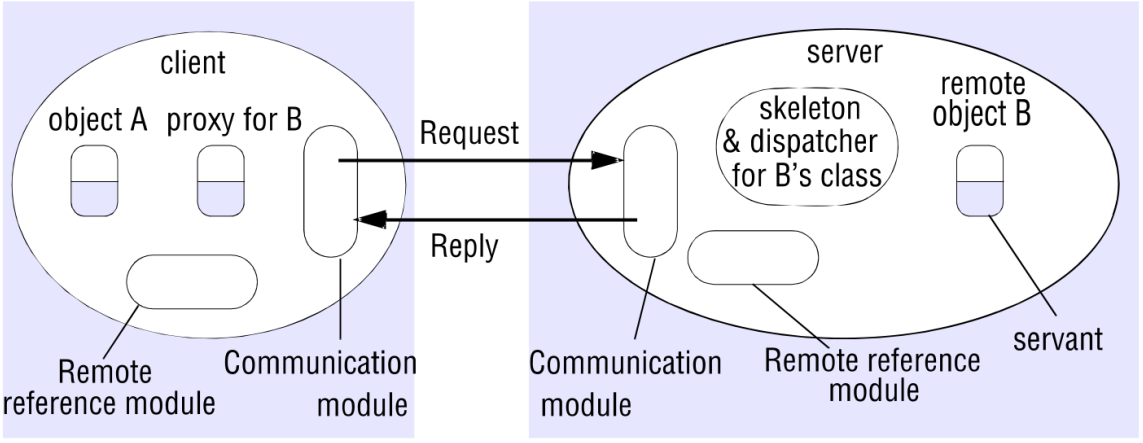
Dispatcher

2 Empfängt Anfragenachricht

3 Ruft entspr. Skeleton-Methode auf

7 Sendet Ergebnisnachricht

RMI - Funktionsweise



- Die Klassen für Proxies, Dispatcher und Skeletons können automatisch erzeugt werden.
- **Remote Reference Module:** Verantwortlich für Übersetzung zwischen lokalen und entfernten Objekten, erzeugt Remote Object Referenzen.
- **Communication Module:** Setzt das Request-Reply-Protokoll um. Server-seitig: Wählt entsprechenden Dispatcher.

Java RMI

- Relevante Klassen und Interfaces im Package `java.rmi` gekapselt
- **Remote Interfaces** müssen die Schnittstelle `java.rmi.Remote` erweitern
  - Alle Methoden müssen zusätzlich `java.rmi.RemoteException` werfen
- **Remote Objects** implementieren das gewünschte Remote Interface und erweitern `java.rmi.server.UnicastRemoteObject`
- Remote Objects in Methodenparametern und Rückgabewerten werden als Remote Object Referenzen übergeben
- Non-Remote Objects (serialisierbar) werden by-value übergeben, d.h. ein neues Objekt wird beim Client erzeugt

Java RMI

- Auf die Registry für Remote Objects kann über die statischen Methoden der Klasse `java.rmi.Naming` zugegriffen werden: rebund(), bind(), unbind(), lookup().
- Die Methode `Naming.rebind(String name, Remote obj)` wird vom Server zur Registrierung eines Remote Objects aufgerufen
- Der Client erhält Referenzen auf Remote Objects der Registry via `Naming.lookup(String name)`
- Die Klasse `java.rmi.registry.LocateRegistry` bietet statische Methoden zur Erzeugung von und den Zugriff auf RMI Registry Instanzen
- Empfehlung RMI-Tutorial: `http://docs.oracle.com/javase/tutorial/rmi/`

Weiteres

Microservices

Anwendung des Scale-Cubes: Jeder Service implementiert eine Sammlung von wenigen ähnlichen Funktionen. Vollständig unabhängig von anderen MS. Komunikation via HTTP/Rest oder AMQP. Jeder MS kann eigene DB haben.

AMQP

Advanced Message Queuing Protocol: standardisiertes binäres Netzwerkprotokoll. `http://www.amqp.org/`

Play Framework

Verschiende Konzepte und Technologien, zB:

- RabbitMQ: Multi-Client and Protocol Messaging/Scheduler (AMQP)
- Akka: Actor/Transactor Model