Web-Anwendungen: Back-End

Web-Engineering II

Prof. Dr. Sebastian von Klinski

Back-End

- Ausgeführt auf dem Server
- Stellt Services für Client-Anwendungen zur Verfügung
- Kapselt häufig
  - Anwendungslogik
  - Zugriff auf Persistenz-Medium (in der Regel Datenbank)
  - Austausch zwischen unterschiedlichen Sessions
- Zielsetzungen für Umsetzung
  - Möglichst schnell die Anfragen der Clients beantworten
  - Flexibel an die Bedürfnisse des Client anpassbar
  - Flexibler Einsatz auch für unterschiedliche Clients (Web, Mobile, System, etc.)

Qualitätskriterien für Back-end Dienste

- Availability
  - Verfügbarkeit (geringe Ausfallzeiten)
- Accessibility
  - einfacher Zugang (z.B. durch Nutzung von gängigen Protokollen)
- Integrity/ reliability
  - Fehlertoleranz; stürzt er ab bei Fehlnutzung?
- Throughput
  - Durchsatz; wie viele Anfragen können gleichzeitig bearbeitet werden?
- Latency
  - Antwortzeit, bis Ergebnis erhalten wurde
- Conformance
  - Einhaltung von Standards, z.B. Message-Standards wie ebXML, etc.
- Security
  - Vertraulichkeit, Authentifizierung, etc.

Back-End: Fragestellungen für diese Vorlesung

- Was ist bei der Umsetzung von flexibel einsetzbaren Services zu beachten?
  - Kopplung zwischen Client und Server
- Wie kann die Schnelligkeit von Services optimiert werden?
  - Möglichkeiten der Performance-Optimierung
  - Zustandsbehaftet/ zustandslos

Kopplung zwischen Client und Server

Kopplung

- Mit Kopplung wird die Verbindung/ Kommunikation zwischen Client und Server beschrieben
- **Enge Kopplung**
  - Viele/ starke Abhängigkeiten zwischen Client und Server
  - Änderung an Client/ Server führt auch zu Änderungen beim anderen Kommunikationspartner
  - Klare Vorgaben zur Implementierung (Programmiersprachen, Protokolle, etc.)
- **Lose Kopplung**
  - Wenige Abhängigkeiten zwischen Client und Server
  - Möglichst einfache Schnittstelle, die sich selten oder nie ändert
  - Client und Server können unabhängig voneinander entwickelt werden
  - Wenige Vorgabe bezüglich der Implementierung (Programmiersprache, Netzwerkprotokoll, etc.)

Früher enge Kopplung

- In Vergangenheit waren Bandbreite und Rechenleistung limitierende Faktoren
- Kopplung zwischen Client und Server häufig eng, um Bandbreite und Rechenleistung zu sparen
  - Optimierung der übertragenen Daten und des Kommunikationsprozesses
  - Häufig binäre Kommunikationsprotokolle
  - Netzwerkkommunikation als Fortsetzung lokaler Operationen gesehen
  - Beispiele: RPC Remote Procedurce Call, DCOM
  - Entfernte Methodenaufrufe wie Aufrufe bei lokalem Objekt/ einer Klasse
- Feste Vorgaben bezüglich Netzwerkprotokoll, Message-Protokoll, Kommunikationsprozess, etc.
- Kaum wiederverwendbar
- Aufwändige Anpassung
- Schwierige Versionierung der Kommunikation

Heute eher lose Kopplung

- Heute sind Bandbreite und Rechenleistung keine limitierenden Faktoren mehr
- Ziel ist vielmehr Wiederverwendung von Programm-Code
- Entwicklungsleistung und Entwicklungsgeschwindigkeit sind limitierende Faktoren
- Wiederverwendung → Weniger Implementierungsaufwand, um Anwendungen umzusetzen

Ansatz

- einfache Schnittstelle mit wenige Vorgaben für Kommunikation zwischen Client und Server
- Nutzung durch unterschiedliche Clients mit unterschiedlichen Technologien
- Nur sehr elementare Aspekte werden vorgeschrieben

→ Entwicklung von Web-Services

Web-Services

Definitionen Web Services

W3C

- A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

… oder

- “a Web Service is an application component accessible over open protocols”.

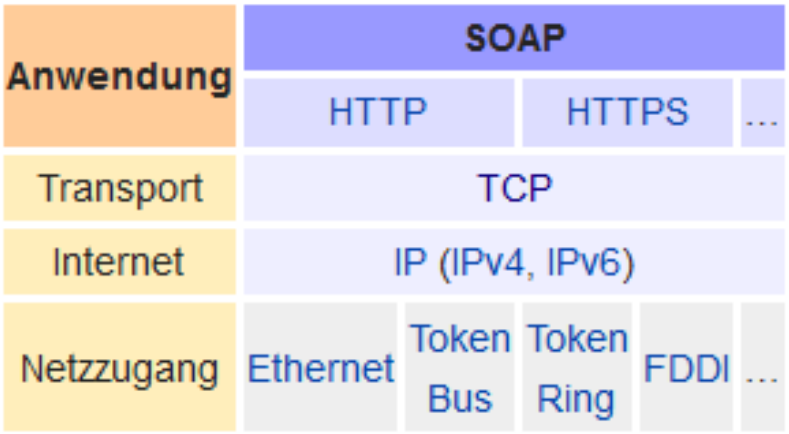
Web-Services

- Wesentliche Aspekte
  - Kommunikation zwischen Systemen über ASCII-Nachrichten
  - In der Regel im Internet über HTTP und HTTPS
  - Identifikation der Services über URI (Uniform Resource Identifier)
- Anmerkungen
  - ASCII kann jeder Rechner verstehen
  - HTTP/ HTTPS kann auch jeder Rechner
  - URI ist durch Internet allgegenwärtig
- Laut W3C: Bestandteile von Web-Services
  - UDDI (Universal Description, Discovery and Integration): Verzeichnisdienst zur Registrierung von Webservices
  - WSDL (Web Services Description Language): Beschreibung des Service
  - Gängige Kommunikationsprotokolle: SOAP, XML-RPC, REST
- In der Praxis kaum Einsatz von UDDI und WSDL

SOAP – Message Protokoll

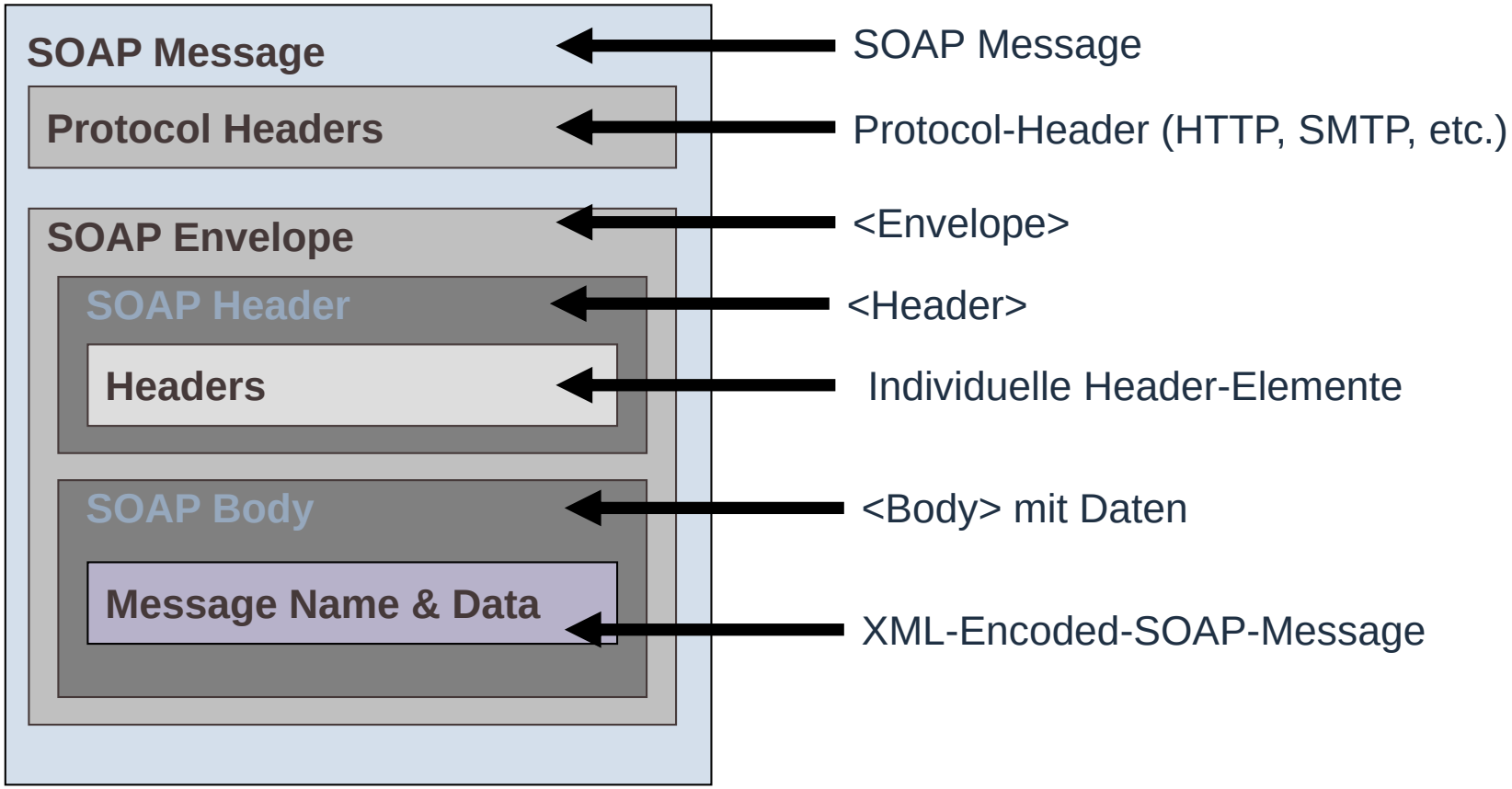
- Verbreitet vor allem in Anfängen von Web-Services
- Industrieller Standard des World Wide Web Consortiums (W3C)
- Message-Protokoll üblicherweise auf Basis von XML, ist aber nicht zwingend
- Gängige Netzwerkprotokolle: HTTP und TCP
- WSDL (Web-Service Description Language) zur Beschreibung der Services
- Grundlage für Service-orientierte Architekturen (SOA)

SOAP im TCP/IP-Protokollstapel:



Quelle: Wikipedia

SOAP Message-Struktur



SOAP Request Beispiel: RPC

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type: application/soap+xml; charset=utf-8 Content-Length: 150

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle=http://www.w3.org/2001/12/soap-encoding”>

    <soap:Body xmlns:m="http://www.stock.org/stock">
            <m:GetStockPrice>
                <m:StockName>IBM</m:StockName>
            </m:GetStockPrice>
    </soap:Body>
</soap:Envelope>
```

SOAP Response Beispiel

```
HTTP/1.1 200 OK
Content-Type: application/soap; charset=utf-8
Content-Length: 126

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

    <soap:Body xmlns:m="http://www.stock.org/stock">
        <m:GetStockPriceResponse>
            <m:Price>34.5</m:Price>
        </m:GetStockPriceResponse>
    </soap:Body>
</soap:Envelope>
```

Beispiel Web-Service-Message für Auftrag: Message-based Call

```
<s:Envelop
   xmlns:s=„http://www.w3.org/2001/06/soap-envelop“
   s:encodingStyle=„http://www.w3.org/2001/06/literal“>
   <s:Body>
      <Auftrag type=„Standardauftrag“ id=„123123123123“>
         <Kunde id=„123123123“>
            <KundenName>ABC Company</Name>
            <Adresse>
               <Strasse>Stralauer Allee 17</Strasse>
               <Stadt>Berlin</Stadt>
            </Adresse>
         </Kunde>
         <INCO>30 Tage, netto</INCO>
      </Auftrag>
   </s:Body>
</s:Envelop>
```

SOAP: Ablauf der Kommunikation

- Client sendet einen SOAP Request
- Erhält HTTP Success-Code und SOAP-Response oder HTTP-Error-Code
- Message-Struktur erlaubt auch komplexe Antworten mit einem Response-Dokument
- Aber:
  - Parsing aufwändig
  - Daher bei vielen Anfragen langsame Kommunkation
- Bei Anwendungen mit hohen Performance-Anforderungen problematisch

Möglichkeiten für Performance-Optimierung

Relevante Aspekte

Performance-Optimierung kann unterschiedliche Zielsetzungen verfolgen

Verfügbarkeit (Availability)

Kontinuierliche Verfügbarkeit eines Services oder Funktion (z.B. 24x7)

Skalierbarkeit

Volumen der Anfragen, die an das System pro Zeiteinheit gestellt werden.

Zahl der Nutzer deutlich angestiegen ist oder die Nutzung des Systems

Zugriffszeiten (Performance)

Zeitraum zwischen dem Absenden einer Anfrage und dem Erhalten der Antwort

Wartbarkeit

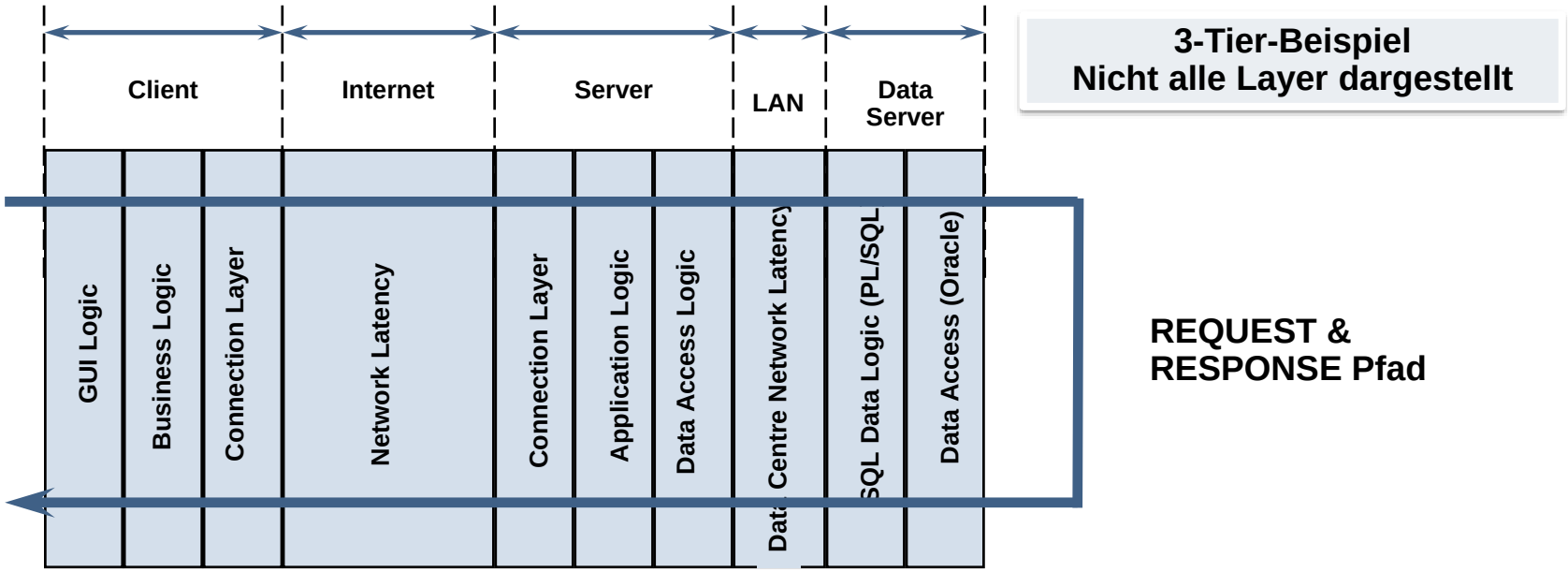
Möglichkeit auch während des Betriebs Hardware- und Software-Komponenten auszutauschen

Ausfallsicherheit

Automatischer Wechsel zu Back-Up-Systemen

Datensicherung

Anfragen bei verteilten Systemen



- Bei verteilten Anwendungen durchlaufen Anfragen zahlreiche Schichten in jedem Teil des Gesamtsystem
- Performance von verteilten Anwendungen ist so gut, wie das schwächste Glied in der Kommunikationspipeline (Bottleneck)

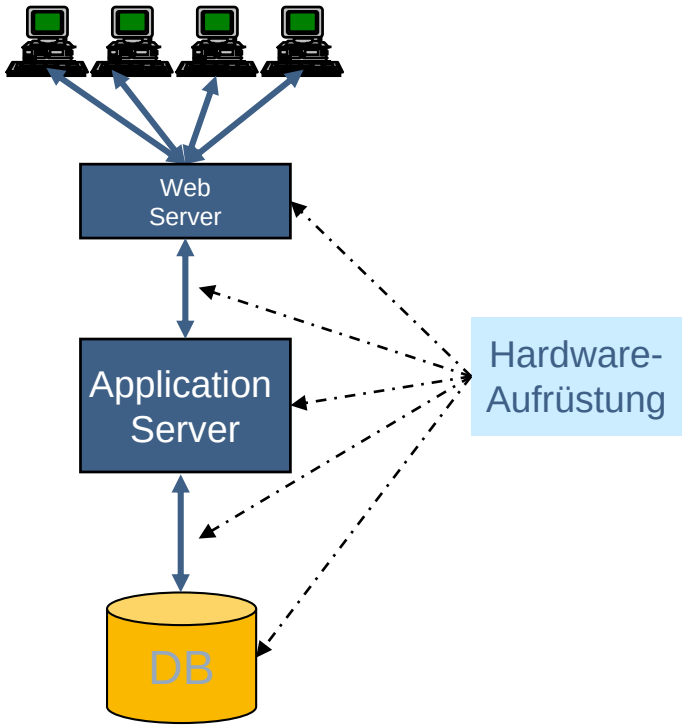
Performance-Optimierung bei verteilten Systemen

Ansätze für Optimierung

- Optimierung der eigenen Software-Komponenten
- Optimierung der Kommunikation zwischen den verteilten Komponenten
- Verteilen der Last auf mehrere Server: Load Balancing
  - Vertikales Hardware Load Balancing
  - Vertikale Prozessverteilung
  - Horizontales Load Balancing (Hardware und Software)
  - Netzwerk Load Balancing

Vertikales Hardware Load Balancing

Aufrüsten der Hardware



Nachteile

Hohe Kosten: stets aktuelle Hardware ist deutlich teurer als durchschnittliche Hardware bei meist vergleichbar geringen Performance-Gewinnen

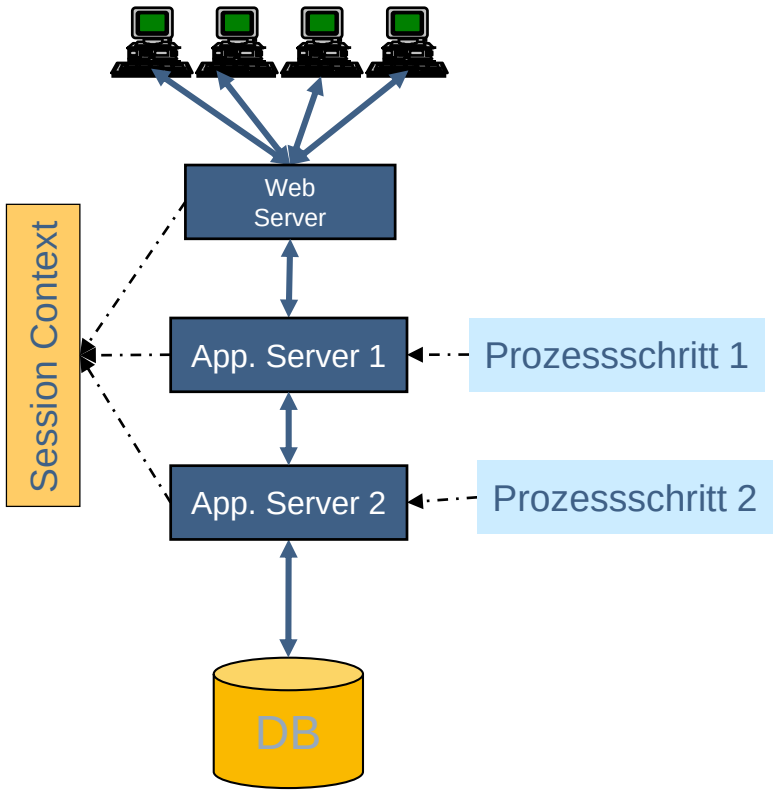
Ausfallzeiten: Falls kein zweites System mit der gleichen Hardware, Konfiguration, Daten- und Programmbestand zur Verfügung steht, ist ein Abschalten für Wartung und Weiterentwicklung unvermeidbar. Die Gefahr von Totalausfällen ist sehr groß.

Begrenzte Skalierung: Hardware-basierte Skalierung ist begrenzt. Ein kontinuierlicher Ersatz der gesamten Hardware ist oft zu kostspielig. Performance-Begrenzungen wirken sich dann auch auf die Möglichkeiten des Systems auswirken.

Kostspielige Skalierungspakete: In der Regel reicht keine selektive Skalierung (z.B. CPU) aus, da meist auch anderen Komponenten (z.B. RAM) mit erweitert werden müssen.

Vertikale Prozessverteilung

Schritte eines Prozesses oder die Umsetzungsebenen werde sequentiell auf mehrere Server verteilt.



Beispiele

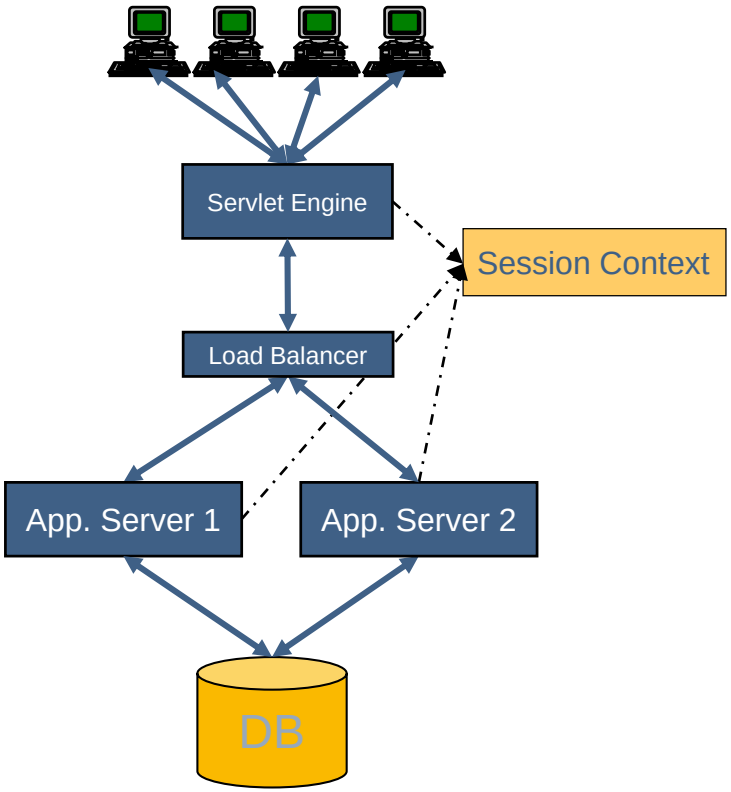
Prozessschrittverteilung:
Multi-Tier-Architekturen sind ein Beispiel für das Aufteilen von Aufgaben wie GUI-Rendering, Business Logik, Integration mit anderen Servern, Datenabstraktion und Datenzugriff auf mehrere Server

Umsetzungsebenen:
Die Trennung von der Geschäftslogik von dem Datenmanagement ist ein Beispiel, wie übergreifende Objektkonzepte auf dem einen Server, Elementartransaktionen wie das Lesen und Schreiben von einem anderen Server ausgeführt werden.

Nachteil
Nicht immer reicht die Skalierung entlang einer Pipeline aus
Zusätzlicher Aufwand für Trennung der Geschäftsprozesse auf mehrere Server (Halten des Session Contextes)
Kommunikation zwischen Prozessen ist sehr langsam

Horizontales Load Balancing

Anfragen werde über mehrere Server mit den gleichen Services verteilt.



Beschreibung

Oft als Clustering bezeichnet.

Verteilung von Anfragen an den Server mit geringsten Belastung oder meisten noch ungenutzten Ressourcen

Verteilung der Anfragen softwarebasiert (z.B. über entsprechenden Application Server mit Load Balancing Funktionalität) oder hardwarebasiert (entsprechende Hardware Load Balancer)

Wichtiger Aspekt ist das Vorhalten des Session Contextes. Bei Transaktionen, die über mehrere Anfragen hinweg gehen, muss der Session Context eines Benutzers über mehrere Server hinweg abrufbar sein.

Vorteile

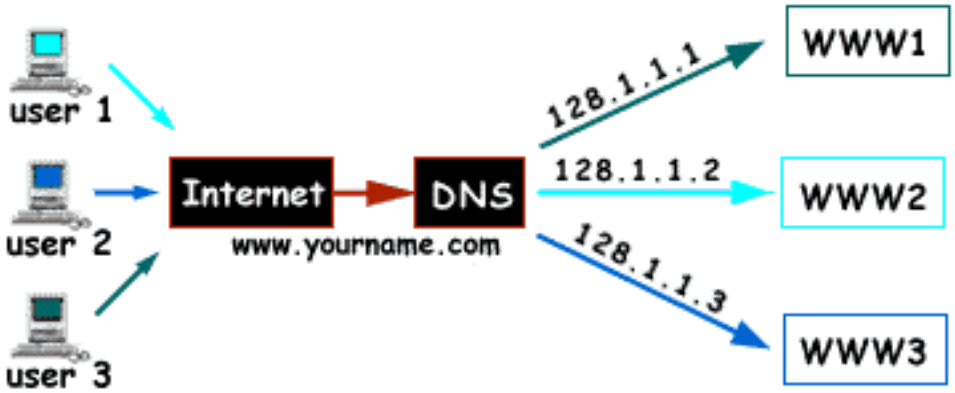
Einfachere und flexiblere Skalierung
Höhere Ausfallsicherheit
Geringere Kosten für Hardware
In der Regel keine Anpassung der Software erforderlich
Unbegrenzte Skalierung

Horizontales Load Balancing

Wozu Session-Propagation beim Horizontalen Load-Balancing?

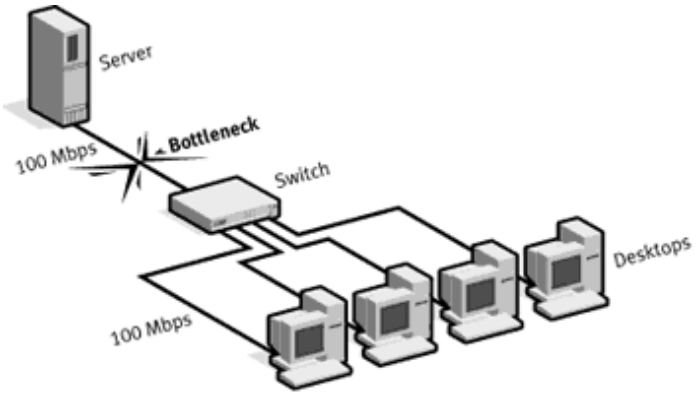
Beispiel: Web-Shop

- Wenn sich ein Nutzer im Web-Portal einloggt und Artikel in seinen Shopping-Cart legt, wird eine Session angelegt, in der alle Informationen erfasst werden
- Wenn er bei jeder Aktion auf einen anderen Web-Server geleitet wird, müsste er sich theoretisch immer wieder einloggen, um auf die Informationen wie den Shopping-Cart zugreifen zu können
- Durch Session-Propagation können Session-Daten zwischen verschiedenen Servern eines Clusters ausgetauscht werden



Network Load Balancing

Engpässe des Netzwerks werden angegangen, um Probleme des Durchsatzes zu beheben.



Hohe Anzahl von Anfragen überlastet Netzwerke schnell

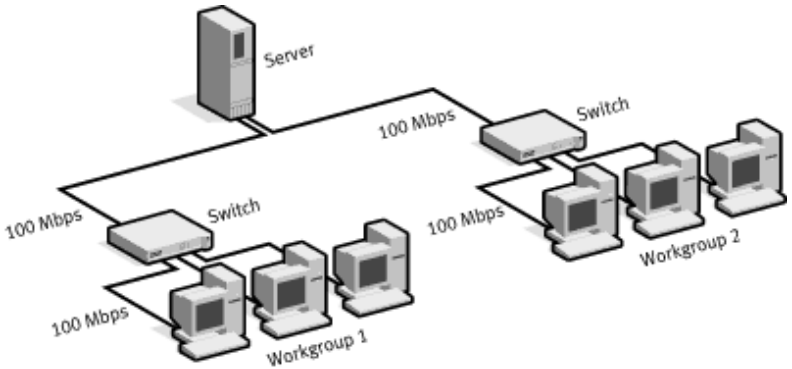
Austausch von Hub durch Switches

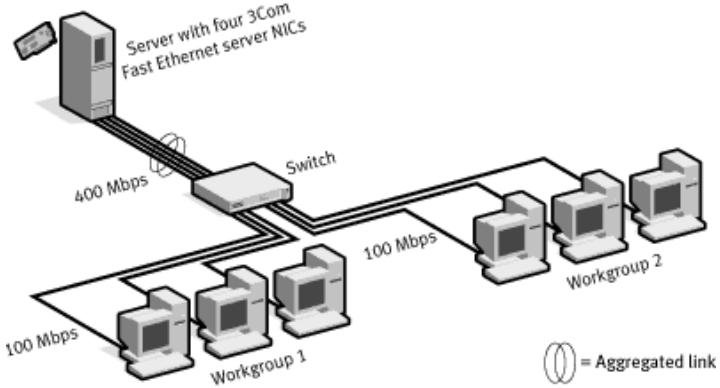
Aufteilen der Clients auf mehrere Switches.

Einsetzen von Switches mit mehreren logischen Eingängen

Erkenntnis

Netzwerkbasierte Anwendungen weisen die beste Performance auf, wenn die Nutzung des Netzwerks minimiert wird.





Performance-Optimierung bei verteilten Systemen

Empfehlungen

- Performance-Optimierung stets aufs Gesamtsystem beziehen
- Nicht vorsorglich alle Aspekte optimieren (unnötig komplexe Software)
- Erst optimieren, wenn Bottlenecks offensichtlich sind

Durch Erfahrung weiß man häufig schon im Vorfeld, wo Bottlenecks sein werden.

Welche Relevanz hat der Umgang mit dem Zustand für die Performance?

Zustand

- Zustand beschreibt den aktuellen Stand einer Anwendung
  - Gängiges Beispiel: Nutzer ist eingeloggt/ nicht eingeloggt
  - Informationen zu dem eingeloggten Nutzer
  - Auch geladene Daten aus der Datenbank
- Zustand kann auch Informationen aktuellen Client-Status umfassen
  - Welche Fenster sind geöffnet?
  - Was darf der Nutzer machen?

Zustand

- Performance-Probleme treten in der Regel aufgrund großen Nutzerzahlen auf
- Ursachen für lange Antwortzeiten können sein
  - Schlechte Programmierung
  - Programmierung ist gut, aber es sind einfach zu viele Anfragen

- Schnelles Antwortverhalten der Anwendung zwingend für kommerziellen Erfolg
- Horizontales Load-Balancing ist gängige Lösung, doch Session Propagation erhöht Fehleranfälligkeit des Systems und schafft zusätzlichen Traffic
- Alternativer Ansatz: Umgang mit dem „Zustand“ überdenken

Grundlagen

- Zustandsbehaftet
  - Der Server hat beispielsweise für eingeloggte Nutzer eine Session, in der alle relevanten Daten einhalten sind
  - Zwischen zwei Anfragen des Nutzers müssen einmal geladene Daten nicht erneut geholt werden, sondern werden aus der Session übernommen
  - Identifikation der Session z.B. über Cockies oder in URL
- Zustandslos
  - Mehrere Anfragen sind – auch desselben Clients – voneinander unabhängige Transaktionen
  - Anfragen haben keinen Bezug zu früheren Anfragen
  - Es gibt keine Sessions
  - Keine Sitzungsinformationen werden ausgetauscht und/oder verwaltet

Zustandslos? Wieso ist das wichtig?

- Vorteile
  - Einfache Skalierung: Horizontales Load-Balancing kann problemlos umgesetzt werden, weil Server keine Informationen austauschen müssen
  - Einfach Skalierung auch innerhalb von Server über Threads, da keine Synchronisation zwischen den Threads notwendig ist
  - Kein Session-Propagation oder ähnliches nötig
  - Speicher kann nach Anfrage wieder freigegeben werden → Keine/ wenige Memory Leaks

Zustandslos? Wieso ist das wichtig?

- Zustandslosigkeit hat auch Nachteile!
- Kein Zustand → Alle Daten müssen für jede Anfrage aus Persistenz-Medium gelesen werden
- Bei komplexer Fachlogik und komplexen Anwendungsprozessen
  - Wiederholtes Laden aller relevanten Daten
  - Umgang mit komplexen Rechteprofil schwierig
  - Validierungen für Ausführen von Transaktionen häufig wichtig
- Daher eBusiness-Architekturen: eher zustandsbehaftet
  - Wenn Services auf unterschiedlichen Server genutzt werden sollen → Weiterleiten der Session-Informationen notwendig
  - Wenn Server ausfällt, ist möglicherweise Zustand weg → anderer Server kann nicht ohne weiteres übernehmen

Zusammenfassung

- Zentrale Design-Aspekte bei Back-end
  - Lose Kopplung zwischen Client und Server
  - Gute Möglichkeiten für Performance-Optimierung
  - Zustandslos/ zustandsbehaftet
- Gängige Ziele
  - Lose Kopplung mit Client → Wiederverwendung für andere Anwendungen
  - Möglichst zustandslos → einfache Skalierung
  - Wiederverwendbarkeit durch saubere Kapselung (klar definierte Anfragen und Antworten)
  - Ausfallsicherheit (z.B. durch horizontales Load-Balancing)

REST

Beispiel: Wiederverwendbarkeit

- Bereitstellen von REST-Services für externe Nutzer
  - Google Maps: https://developers.google.com/maps
  - Flickr: https://www.flickr.com/services/developer/api/
- Eigene Anwendung schreiben: Welche Flickr Bilder wurden in der Nähe (Berlin) geschossen?
  1. **Long/Lat Geokoordinaten abrufen**
     https://maps.googleapis.com/maps/api/geocode/json?address=Berlin **
  2. **Flickr API abfragen**
     https://api.flickr.com/services/rest/?method=flickr.photos.search&lat=52&lon=+13.404954&format=json&nojsoncallback=1 **
  3. **Bilder anzeigen**
     - user-id und photo-id aus Ergebnis verwenden und in URL einbauen
     https://www.flickr.com/photos/[usr]/[photo]
- Insbesondere bei Bereitstellung für externe Nutzer ist einfache Schnittstelle von zentraler Bedeutung!

REST (Representational State Transfer)

- Verbreiteter Ansatz zur Umsetzung von Web-Services
- Bezeichnung „Representational State Transfer“: Übergang einer Ressource vom aktuellen Zustand zum nächsten Zustand über eine URI
- Programmierparadigma für verteilte Systeme, insbesondere Webservices
- Maschine-zu-Maschine-Kommunikation
- Alternative zu ähnlichen Verfahren wie SOAP/ WSDL oder RPC
- Deutlich prägnanter als SOAP, aber keine Validierung per XSD
- Keine Methodeninformation in den URI
- WWW umfasst bereits nötige Infrastruktur (z.B. Web- und Application-Server, HTTP-fähige Clients, HTML- und XML-Parser, Sicherheitsmechanismen)

REST-Services sollten zustandslos sein

- Jede REST-Nachricht enthält alle Informationen, die für den Server bzw. Client notwendig sind, um die Nachricht zu verstehen und zu verarbeiten.
- Der Server soll keine Zustandsinformationen zwischen zwei Nachrichten speichern.
- Begünstigt die Skalierbarkeit eines Webservices: einfache Lastverteilung, höhere Ausfallsicherheit
- Häufig verwenden Web-Anwendungen Cookies und andere Techniken, um Zustandsinformationen auf der Client-Seite zu halten

REST

- Gängige Protokolle: HTTP und HTTPS
- Geht leicht durch Firewalls
- Lose Kopplung durch Kodierung der Nachrichten als ASCII-Texte

HTTP GET
`/api/users/33245/messages`
Gib mir alle Messages des Nutzers 33245

HTTP DELETE
`/api/message/332456`
Lösche Message 332456

- Nutzung von HTTP-Methoden
- URLs für Ressourcenidentifikation

REST

- REST-Services sehr leicht umzusetzen
- Umfangreiche Tool-Unterstützung
- Node.js derzeit sehr populäre Plattform