Technische Hochschule Ingolstadt
Fakultät für Elektrotechnik und Informatik

Zukunft in Bewegung

Skalierbarkeit

Architektur und Entwurfsmuster der Software Technik

Prof. Dr. Bernd Hafenrichter 29.05.2018

Motivation

Skalierbarkeit ist die Eigenschaft eines Software-Systems, mit einem größeren Lastaufkommen fertig zu werden, indem zusätzlich bereitgestellte Ressourcen genutzt werden. Man spricht dann davon, dass das System das größere Lastaufkommen kompensieren kann

Arten der Skalierbarkeit

- **Lastskalierbarkeit:** konstantes Systemverhalten über größere Lastbereiche hinweg. Keine oder nur geringe Verzögerungen bei kleiner, mittlerer oder hoher Last.

- **Räumliche Skalierbarkeit:** Der Speicherplatz steigt nicht inakzeptabel hoch an, wenn die Anzahl der zu verwalteten Objekte steigt.

- **Zeitlich-räumliche Skalierbarkeit:** Die zeitlich-räumlichen Skalierbarkeit dagegen zeigt an, wie das System mit der erhöhten Antwortzeit umgeht, wenn die Anzahl der Datenobjekte ansteigt.

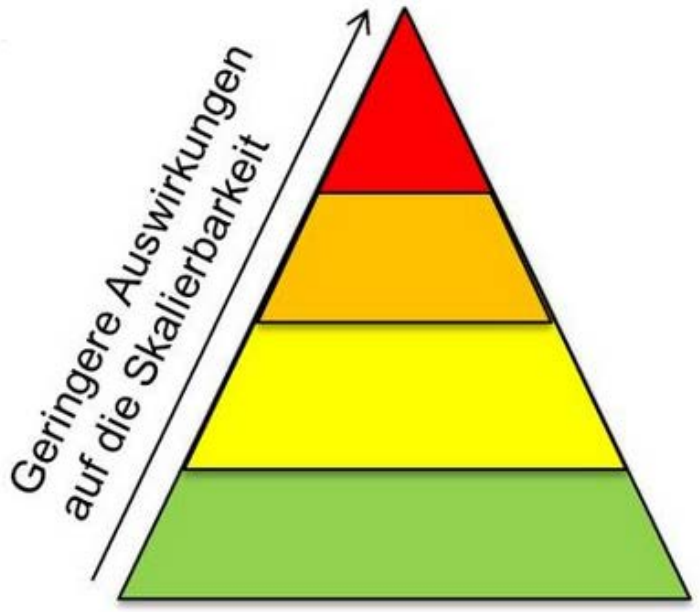
- **Strukturelle Skalierbarkeit:** Eine gute strukturelle Skalierbarkeit sagt aus, dass sich die Software-Architektur des Systems nicht hinderlich auf die Erhöhung der Anzahl der Datenobjekte auswirkt.

Arten der Skalierbarkeit

- **Geographische Skalierbarkeit:** Bei der geographischen Skalierbarkeit geht es darum, das die geographische Entfernung des Systems von Benutzer und Ressourcen keinen oder nur wenig Einfluss auf die Leistungsfähigkeit des Systems hat.

- **Administrative Skalierbarkeit:** Bei der administrativen Skalierbarkeit geht es darum, dass die ggf. variierende Anzahl der Organisationen, die sich ein System teilen, nicht beschränkt ist. Anhand dieser Art der Skalierbarkeit soll das Managen, Überwachen und der Gebrauch eines Software-Systems einfach zu bewerkstelligen und von überall auf der Welt aus möglich sein.

Maßnahmen zur Verbesserung der Skalierbarkeit



Die Skalierbarkeitspyramide

Maßnahmen zur Verbesserung der Skalierbarkeit

Grundsätzlich

- Wahl der richtigen Skalierungsstrategie

Skalieren der Persistenzsschicht

- Verteilung der Datenbanklast

Refactoring der Applikationslogik

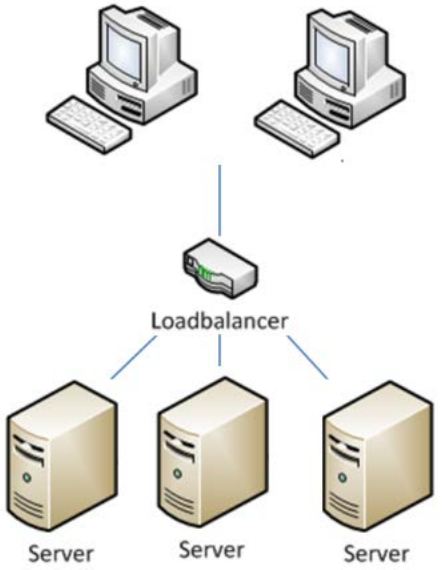
- Zustandslosigkeit fördern
- Asynchrone Kommunikation vor synchroner Kommunikation
- Code und Ressourcen trennen
- Minimieren von Ressourcenkonflikte
- Trennung von transaktionale Aufrufe und nicht-transaktionalen Aufrufen
- Schichten- und Servicebildung

Maßnahmen zur Verbesserung der Skalierbarkeit: Skalierungsstrategie

- **Scale-Up (vertikale Skalierung)**
  - Steigerung der Leistung **eines Systems** durch das Hinzufügen/Austausch von Ressourcen eines Knotens/Rechner
    - Vergrößern des Speicherplatzes
    - Hinzufügen einer CPU
    - Einbauen einer leistungsstärkeren Netzwerkkarte
    - Einbau von SSD-Festplatten
  - Diese Skalierungsstrategie ist unabhängig von der Art der implementierten Software
  - Keine Änderungen am Quellcode
  - Skalierungsstrategie ist begrenzt durch die Verfügbare Hardware und deren Kosten

Maßnahmen zur Verbesserung der Skalierbarkeit: Skalierungsstrategie

- **Scale-Out (horizontale Skalierung)**
  - Steigerung der Leistung des Systems durch Hinzufügen zusätzlicher Rechner/Knoten
  - Parallelbetrieb identischer Software auf mehreren Knoten
  - Elastizität: Skalierbarkeit der Anwendung während des laufenden Betriebs
  - Scale-Out ist sehr stark von der implementierten Software-Architektur abhängig.
  - Refrakturierung der Applikationen um die Effekte bei einem Scale-Out nutzen zu können.
    - Aufspalten eines Monolithen in mehrere, unabhängige einzelne Applikationen



Maßnahmen zur Verbesserung der Skalierbarkeit: Datenbanklast richtig verteilen

Strategien zur Skalierung der Persistenzschicht

- Skalierung der Datenbankzugriffe
- Skalierung hinsichtlich Datenbankeinträge – De-Normalisierung
- Fragmentierung auf Tabellenebene
  - Horizontalen Fragmentierung
  - Vertikalen Fragmentierung
- Fragmentieren/Aufteilen der Datenbank
- Überdenken des Transaktionskonzeptes
- Wechsel der Datenbanktechnologie

Maßnahmen zur Verbesserung der Skalierbarkeit: Datenbanklast richtig verteilen

Skalierung der Datenbankzugriffe

Problem:

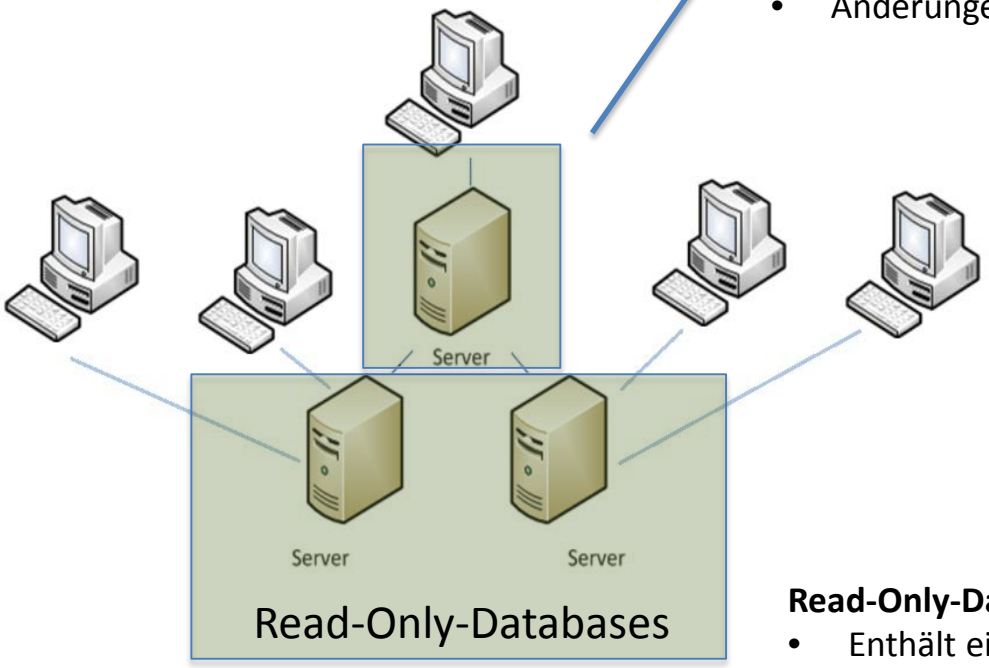
- Read-Only-Anfragen werden durch Schreibzugriffe auf eine Datenbankbank blockiert.
- Ursache: Einhaltung des ACID-Prinzips durch Sperren (Synchronisation)
- Beispiel: Buchreservierungsplattform welche ein Verhältnis zwischen Lese- und Schreibzugriffen von 400:1

Lösung

- Annahme: Eine Datenbank hat wesentlich mehr Lese als Schreibzugriffe
- Erstelle mehrere Read-Only-Kopien der Datenbank und verteile den lesenden Zugriff auf diese Kopien.
- Hohe Parallelität das Leseoperationen nicht durch Schreibvorgänge blockiert werden.
- Änderungen werden auf einem zentralen Master durchgeführt.
- Änderungen werden auf die Read-Only-Kopien verteilt
- Achtung: Die Aktualisierung der Kopien ist nicht 100%-synchron. Es existiert ein „gewisser“ Zeitversatzt (Latenz)

Maßnahmen zur Verbesserung der Skalierbarkeit: Datenbanklast richtig verteilen

- **Skalierung der Datenbankzugriffe**



Master-Datenbank
- Zeichnet alle Änderungen der Daten auf
- Änderungen werden an Slaves repliziert

Read-Only-Databases
- Enthält eine Read-Only Kopie
- Änderungen werden zeitversetzt in die Slave-DB übernommen

Maßnahmen zur Verbesserung der Skalierbarkeit: Datenbanklast richtig verteilen

- **Skalierung hinsichtlich Datenbankeinträge – Denormalisierung**

  Problem:
  - Komplexe Abfragen (mit vielen Join-Operation) sind aufwendig in der Berechnung und dadurch in performant

  Lösung:
  - Bewusste Rücknahme einer Normalisierung zum Zweck der Verbesserung des Laufzeitverhaltens einer Datenbankanwendung.

  Mögliche Maßnahmen:
  - Ablage von Master-Detail-Daten in einem Blob-Feld der Master-Tabelle
  - Redundante Ablage von Daten
    - Tabellen mehrfach speichern
    - Felder der Parent-Tabelle im Child mit speichern

Maßnahmen zur Verbesserung der Skalierbarkeit: Datenbanklast richtig verteilen

Fragmentierung – Horizontalen Fragmentierung auf Tabellenebene

Problem:

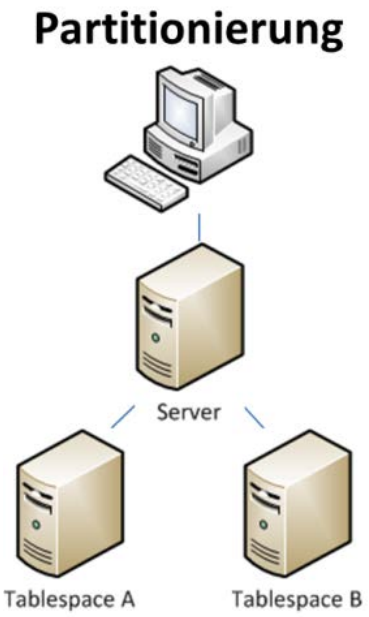
- Das IO-Subsystem einer Datenbank ist ein Flaschenhals bei konkurrierenden Zugriffen.

Grundidee:

- Aufteilen der der Gesamtheit der Datensätze einer Tabelle auf unterschiedliche (physische) Tabellen

Horizontale Partitionierung:

- Die Datensätze einer Tabelle werden auf unterschiedlichen IO-Subsystemen gespeichert (=Tablespace)
- Das Datenbanksystem verteilt eingehende Anfragen transparent auf die verschiedenen Tablespaces auf
- Partitionierung ist transparent für die Applikationslogik



Maßnahmen zur Verbesserung der Skalierbarkeit: Datenbanklast richtig verteilen

Fragmentierung – Horizontalen Fragmentierung auf Tabellenebene

Problem:

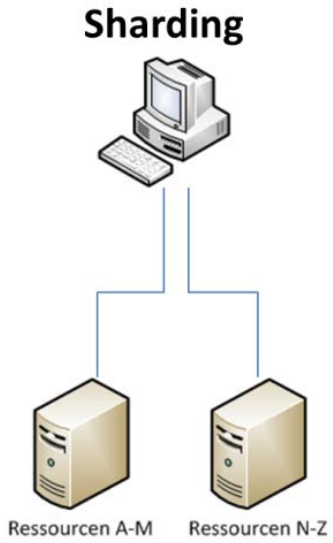
- Das IO-Subsystem einer Datenbank ist ein Flaschenhals bei konkurrierenden Zugriffen.

Grundidee:

- Aufteilen der der Gesamtheit der Datensätze einer Tabelle auf unterschiedliche (physische) Tabellen

Sharding:

- Die Teile einer Tabelle können über mehrere unabhängigen Datenbanken verteilt werden
- Im Idealfall beziehen sich Anfragen immer genau auf einen Teil der Tabelle. Ein lesen der gesamten Daten ist nicht notwendig.
- Anpassung der Applikationslogik notwendig.
- Problematisch bei übergreifenden Aktualisierungen da Transaktionssemantik (ACID) evtl. schwer herzustellen ist



Maßnahmen zur Verbesserung der Skalierbarkeit: Datenbanklast richtig verteilen

Fragmentierung – Vertikale Fragmentierung

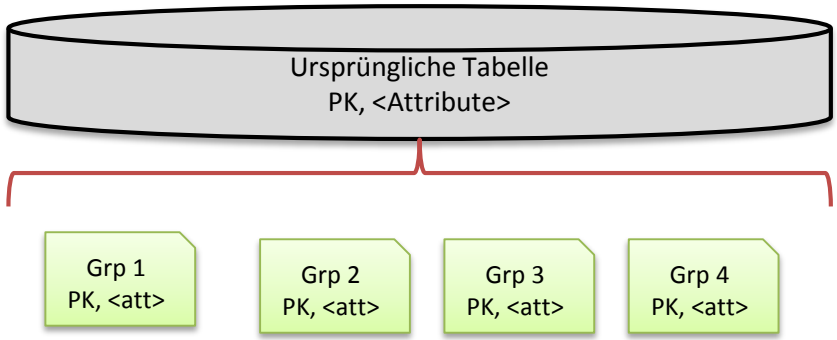
Problem:

- Bei großen Tabellen (viele Attribute/Spalten) werden unnötig viele Daten gelesen.
- Die Lese-Häufigkeit der einzelnen Spalten variiert sehr start

Grundidee:

- Gruppen von Nicht-Schlüssel-Attributen die zu einer Tabelle gehören werden auf verschiedene Tabelle aufgeteilt.

- Die Gruppen werden nach Zugriffshäufigkeit aufgeteilt.
  - Eine Gruppe mit Attributen die immer benötigt werden
  - Weitere Gruppen mit Attributen die selten gelesen werden bzw. große Datenblöcke enthalten (lazy loading)



Maßnahmen zur Verbesserung der Skalierbarkeit: Datenbanklast richtig verteilen

Fragmentieren/Aufteilen der Datenbank

Problem:

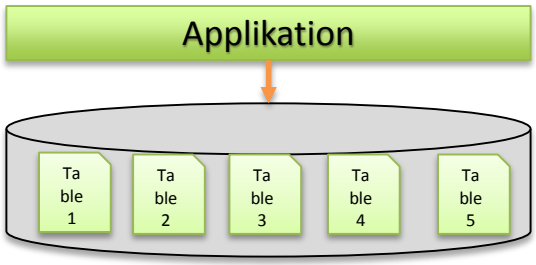
- Eine Datenbank enthält Tabellen welche logisch nicht zusammenhängend sind

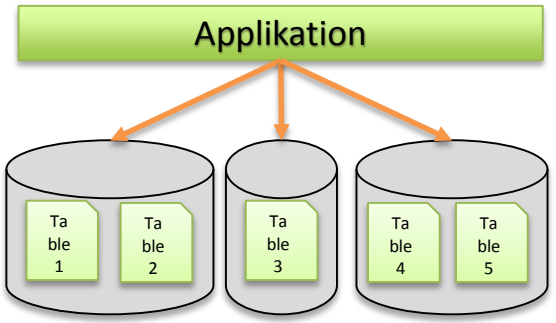
Grundidee:

- Aufteilen der Tabellen einer (Gesamt-)Datenbank auf unterschiedliche Datenbanksysteme.

Ansatz:

- Aufteilen der Gesamtdatenbank in logisch zusammenhängenden Gruppen von Tabellen
- Jede Gruppe wird auf einem eigenständigen Datenbanksystem gespeichert
- Anpassen der Applikationslogik um die unterschiedlichen Zugriffspfade zu realisieren
- Höhere Durchsatz weil die Datenbanklast auf unterschiedliche Systeme aufgeteilt wird und Transaktionen dadurch kleiner werden
- Problematisch bei übergreifenden Aktualisierungen da Transaktionssemantik (ACID) evtl. schwer herzustellen ist





Da 1 mal 1 des Transaktionshandling - Transaktionsprinzipien

Der Architekt muss entsprechend den Anforderungen an die Transaktionalität der verarbeiteten Daten die richtige Auswahl bzgl. des einzusetzenden Transaktionskonzeptes treffen.

Bei einem Einsatz von horizontalen Skalierung ist zwingend das Transaktionskonzept zu überdenken und entsprechend zu implementiert.

Daten, mit strikten Anforderungen an Transaktionalität benötigen die ACID-Prinzipien, welche durch transaktionale Datenbanksysteme implementiert werden:

ACID – Prinzip bei relationalen Datenbanken

- Atomar
- Consistent
- Isoliert
- Dauerhaft

Maßnahmen zur Verbesserung der Skalierbarkeit: Trennung von transaktionale Aufrufe und nicht-transaktionalen Aufrufen

Problem

- Transaktionale Aufrufe erfordern zur Einhaltung des ACID-Prinzips ein Synchronisation der zugrundeliegenden Ressourcen (z.B. Tabellen, Pages, Rows, Indizes). Dies erfolgt in der Regel durch Sperren (Locks)
- Eine Mischung aus transaktionalen und nicht-transaktionalen Aufrufen führt zu einer unnötigen Verzögerung von Anfragen da mehr Sperren innerhalb einer Transaktion existieren

Lösung:

- Trennen von lesenden und schreibenden Zugriffen.
- Dadurch erfolgt eine Verringerung von Sperren
- Erhöhung der Parallelität

- Aktivieren des Row-Versioning (MS-SQL + Oracle)

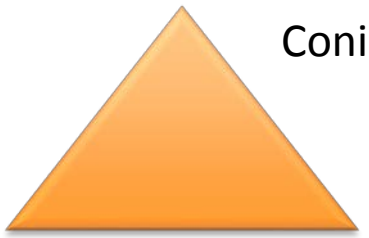
Da 1 mal 1 des Transaktionshandling - Transaktionsprinzipien

Sind die Anforderungen **weicher** hinsichtlich der notwendigen Transaktionalität, können die Daten nach den BASE-Prinzipien verarbeitet werden:

BASE – Prinzip bei No-SQL-Datenbank

- Basically Available, Soft State, Eventual consistency.
- Das System ist nach Ablauf einer gewissen (möglichst kurzen) Zeitspanne der Inkonsistenz wieder in einem konsistenten Zustand.
- Hier wird die Konsistenz der Daten (zumindest temporär) zugunsten einer konstanten Verfügbarkeit der Daten eingeschränkt.

Da 1 mal 1 des Transaktionshandling – Das CAP – Theorem

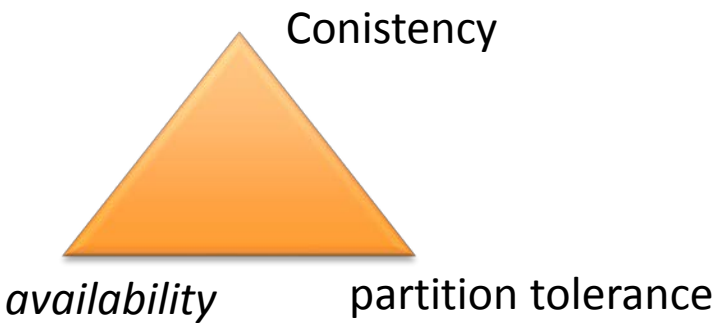


Consistency: Die Konsistenz der gespeicherten Daten. In verteilten Systemen mit replizierten Daten muss sichergestellt sein, dass nach Abschluss einer Transaktion auch alle Replikate des manipulierten Datensatzes aktualisiert werden.

Availablility: Die Verfügbarkeit im Sinne akzeptabler Antwortzeiten. Alle Anfragen an das System werden stets beantwortet.

Partition tolerance: Die Ausfalltoleranz der Rechner-/Servernetze. Das System arbeitet auch bei Verlust von Nachrichten, einzelner Netzknoten oder Partition des Netzes weiter.

Da 1 mal 1 des Transaktionshandling – Das CAP – Theorem



CAP-Theorem (nach Brewer)

In einem Netzwerksystem sind zu einem gegebenen Zeitpunkt nur zwei der drei Eigenschaften Konsistenz (C), Verfügbarkeit (A) und Partitionstoleranz (P) gleichzeitig realisierbar.

Da 1 mal 1 des Transaktionshandling – Das CAP – Theorem

AP: Verfügbar und Partitionstolerant

- Domain-Name-System (DNS): Das DNS fällt in die Kategorie **AP**. Die Verfügbarkeit ist extrem hoch, ebenso Toleranz gegenüber dem Ausfall einzelner DNS-Server. Allerdings ist die Konsistenz nicht immer sofort gegeben: es kann mitunter Tage dauern, bis ein geänderter DNS-Eintrag an die gesamte DNS-Hierarchie propagiert ist und damit von allen Clients gesehen wird.

Da 1 mal 1 des Transaktionshandling – Das CAP – Theorem

CA: Konsistent und Verfügbar

- Ein RDBMS-Cluster fällt meistens in die Kategorie **CA**. Sie streben vor allem Verfügbarkeit und Konsistenz aller Knoten an. Da sie meistens mit hochverfügbaren Netzwerken und Servern betrieben werden, müssen sie nicht unbedingt gut mit einer Partitionierung umgehen können. Wie schon erwähnt, sind C, A und P eben graduelle Größen.

Da 1 mal 1 des Transaktionshandling – Das CAP – Theorem

CP: Verfügbar und Partitionstolerant

- Banking-Anwendungen: Für verteilte Finanzanwendung wie z. B. Geldautomaten ist die Konsistenz der Daten oberstes Gebot: Ein abgehobener Geldbetrag muss zuverlässig auch auf der Kontenseite abgebucht werden, eingezahltes Geld muss auf dem Konto erscheinen. Diese Anforderung muss auch bei Störungen im Datenverkehr sichergestellt werden (Partitionstoleranz).

- Gegenüber der Konsistenz und der Partitionstoleranz ist die Verfügbarkeit zweitrangig (**CP**): bei Netzwerkstörungen soll ein Geldautomat oder ein anderer Service eher nicht verfügbar sein als nicht korrekte Transaktionen abwickeln.

Maßnahmen zur Verbesserung der Skalierbarkeit

Strategien zur Skalierbarkeit der Applikationslogik / Refactoring der Applikationslogik

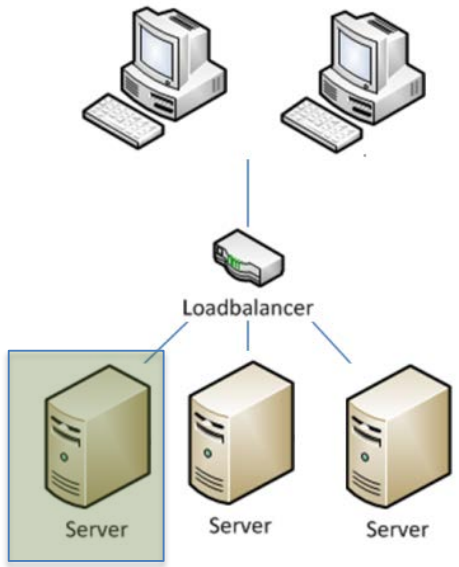
- Zustandslosigkeit fördern
- Asynchrone Kommunikation vor synchroner Kommunikation
- Code und Ressourcen trennen
- Minimieren von Ressourcenkonflikte
- Trennung von transaktionale Aufrufe und nicht-transaktionalen Aufrufen
- Schichten- und Servicebildung

Maßnahmen zur Verbesserung der Skalierbarkeit: Zustandslosigkeit fördern

Zustand: Bindung von Daten an die Ressourcen des verarbeitend End Servers

Problem:

- Verarbeitung ist nur auf dem Server möglich an welchen die zugehörigen Daten gebunden sind. (z.B. Session-Information, Login-Information, Warenkorb, …)
- Ressourcen wie Threads, Speicher, Netzwerkverbindung werden unter Umständen unnötig lange belegt
- Eine Verteilung eingehender Anfragen auf verschiedene Rechner (durch den Load-Balancer) ist nicht möglich



Verarbeitung nur auf explizitem Server möglich

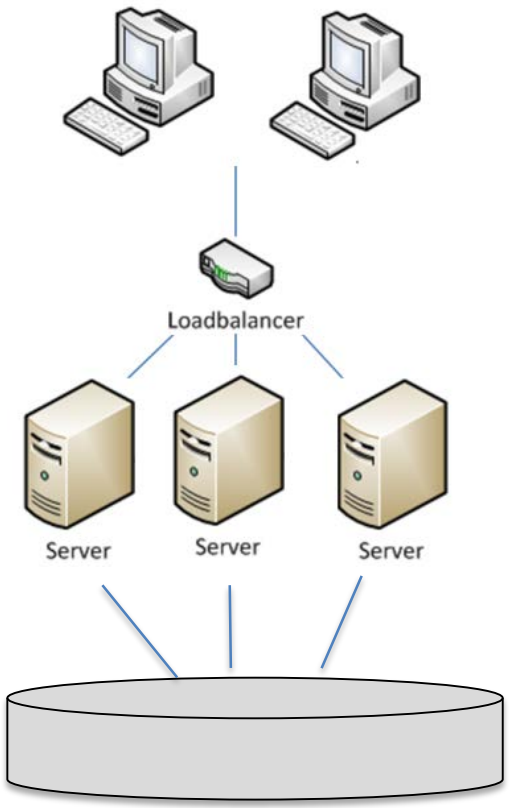
Maßnahmen zur Verbesserung der Skalierbarkeit: Zustandslosigkeit fördern

Zustand: Bindung von Daten an die Ressourcen des verarbeitend End Servers

Lösung:

- Daten/Information welche auf unterschiedlichen Systemen verfügbar sein müssen werden in einen Backend-Store abgelegt
- Daten/Information welche auf unterschiedlichen Systemen verfügbar sein müssen werden zwischen den Systemen aktiv repliziert

Verarbeitung auf allen Server möglich. Abgleich über Backend-Store



Maßnahmen zur Verbesserung der Skalierbarkeit: Asynchrone Kommunikation

- **Problem bei synchroner Kommunikation:**
  - Der Client/Aufrufer blockiert bis das Ergebnis des Aufrufes eintrifft.

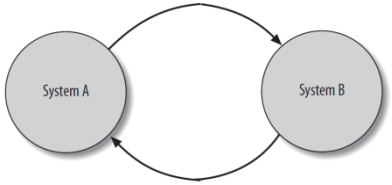
- **Lösung: Asynchrone Kommunikation:**
  - Der Aufrufende Dienst/Client muss nicht auf das Ende eines Auftrages warten.
  - Auftrag wird von Empfänger quittiert (Empfangsbestätigung)
  - Kein warten bis der eigentlich Auftrag verarbeitet wurde
  - Er wird evtl. am Ende von dem erfolgreichen Abschluss informiert.

- **Vorteil:**
  - Der Aufrufende Dienst kann weiter arbeiten und blockiert nicht

- **Achtung:**
  - Die Use-Cases müssen die lose Kopplung auf ebene der Kommunikation erlauben

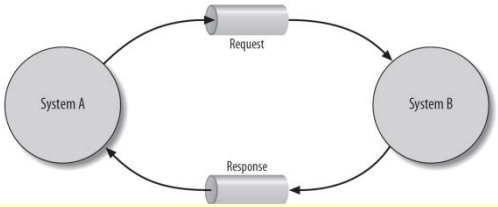
Maßnahmen zur Verbesserung der Skalierbarkeit: Asynchrone Kommunikation

Kommunikation zwischen Systemen erfolgt durch Austausch von one-way Nachrichten. Dieses Grundprinzip kann verwendet werden um komplexe Kommunikationsszenarien zu realisieren



Request/Response (Synchron)

- Request und Response werden über gleichen Kanal ausgetauscht (z.B. TCP/IP, HTTP)
- System A und B müssen gleichzeitig aktiv sein



Request/Response (Asynchron)

- Nachrichten werden indirekt über eine Queue verschickt
- System A und B müssen nicht gleichzeitig aktiv sein
- Korrelation notwendig
- Z.B. Websockets, AJAX

Hoch — **Kopplung** — **Niedrig**

Maßnahmen zur Verbesserung der Skalierbarkeit: Asynchrone Kommunikation

Vorteile der losen Kopplung

- Hohe Stabilität da die Systeme nicht gleichzeitig online sein müssen
- Service Consumer wird nicht durch aktives Warten blockiert
- Höhere Skalierbarkeit

Nachteile der losen Kopplung

- Programmlogik wird komplizierter
- Nachrichten können in unterschiedlichen Reihenfolge eintreffen
- Antwort und Anfrage müssen korreliert werden (Asynchrone Events)
- Antwortzeiten können nicht garantiert werden

Skalierbarkeit des Systems zum Preis einer höheren Komplexität.

Maßnahmen zur Verbesserung der Skalierbarkeit: Asynchrone Kommunikation

Problem bei Kommunikation:

- Fehlerfälle in der Netzwerkkommunikation müssen durch den Client behandelt werden
- Ändernde Aufrufe können aufgrund von Fehlern zu inkonsistenten Datenständen führen

- **Lösung: Asynchrone Kommunikation:**
  - Aufrufe werden durch den Client wiederholt durchgeführt
  - Um Duplikate/Doppelbuchungen zu vermeiden
    - Die Aufrufe müssen idempotent realisiert werden.
    - Idempotente Dienste liefern selbst bei wiederholter Ausführung einer Aktion dasselbe Ergebnis.
    - Idempotente Dienste können Schreiboperationen sehr stabil horizontal skalieren

Definition Idempotenz:

Als idempotent bezeichnet man Funktionsaufrufe, die immer zu den gleichen Ergebnissen führen, unabhängig davon, wie oft sie mit den gleichen Daten wiederholt werden. Idempotente Arbeitsgänge können zufällig oder absichtlich wiederholt werden, ohne dass sie nachteilige Auswirkungen auf den Computer haben.

Maßnahmen zur Verbesserung der Skalierbarkeit: Minimieren von Ressourcenkonflikte

Problem: Synchronisation bei Ressourcenkonflikten

- Ressourcenkonflikte existieren wenn gleichzeitig mehrere Dienste/Aufrufe auf die gleiche Ressource zugreifen
- In Konsequenz muss immer eine Zwangsserialisierung über Rechnergrenzen hinweg erfolgen welche die mögliche Parallelität begrenzt und unabhängige Komponenten in einen Gleichtakt zwingt.

Lösung:

- Reduzieren von Synchronisation auf ein notwendiges Minimum
- Eleminieren von Konflikten
- Je weniger Ressourcenkonflikte existieren um so höher die mögliche Parallelität

    - Prüfen der Anforderung, überdenken der vorhandenen Implementierung

Maßnahmen zur Verbesserung der Skalierbarkeit: Code und Ressourcen trennen

Problem: Skalierbarkeit von externen, abhängigen/benutzen Ressourcen fördern

- Im Quellcode werden die Zugriffe auf externe Ressourcen festgelegt (z.B. durch IP-Adressen)
- Dieser Zugriff muss so implementiert sein, dass die externen Ressourcen ebenfalls skalieren können
- Beispiele: Zugriff auf Datenbank, Mail-Server, Logistic-System, ..

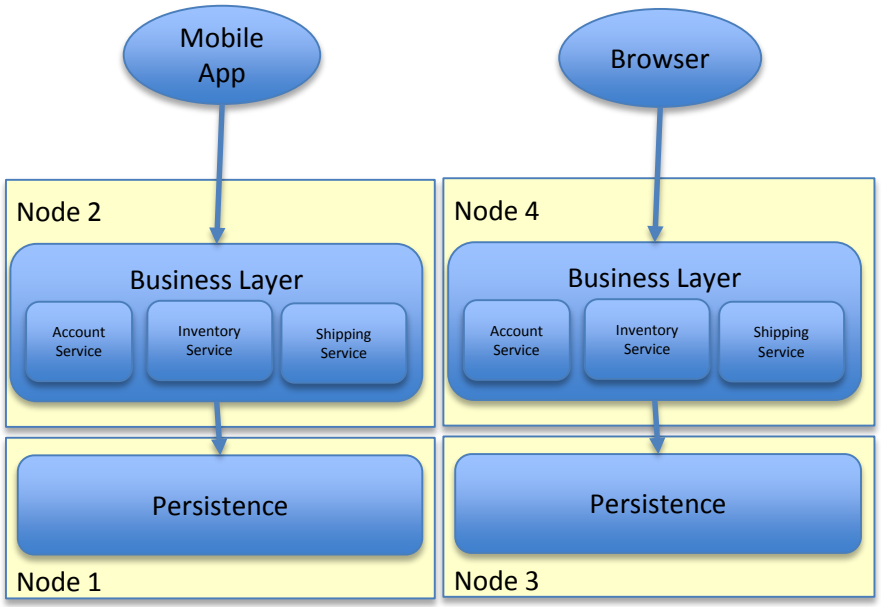
Lösung:

- Um auf Applikationsebene die Parallelisierung von Aktivitäten zu fördern, ist die Trennung von Code und den verwendeten Ressourcen in dem Sinne voranzutreiben, dass bei der Erstellung des Codes keine Limitierungen hinsichtlich der verwendeten Ressourcen implizit oder explizit erfolgt – nur so ist es möglich, das Applikationen alle zu Verfügung gestellten Ressourcen (z. B. im Cloud Computing oder in virtualisierten Umgebungen) nutzen können.
- Die Trennung von Code und Ressourcen bedeutet, dass das eigentliche Ressourcenmanagement nach Möglichkeit dem umgebenden Container (Applikationsserver) bzw. dedizierter Managementsoftware (z. B. Cloud-Management-Software) überlassen wird.

Maßnahmen zur Verbesserung der Skalierbarkeit: Schichten und Servicebildung

Problem: Parallelisierung von Aktivitäten innerhalb der Applikationslogik

- **Lösung: Klassisches Verfahren : Bildung von horizontalen Schichten**
  - Schichten können unabhängig voneinander entwickelt und verteilt betrieben werden. Diese Verteilung erlaubt es, eine Schicht unabhängig von den darüber liegenden Schichten zu skalieren.
  - Aber: ab einer gewissen Größe des Systems kann sich die Dekomposition allerdings als schwierig hinsichtlich der Skalierbarkeit herausstellen, da dann innerhalb einer Schicht oft die funktionale Kohäsion sinkt und die Komplexität steigt – mit negativem Einfluss auf die Skalierbarkeit

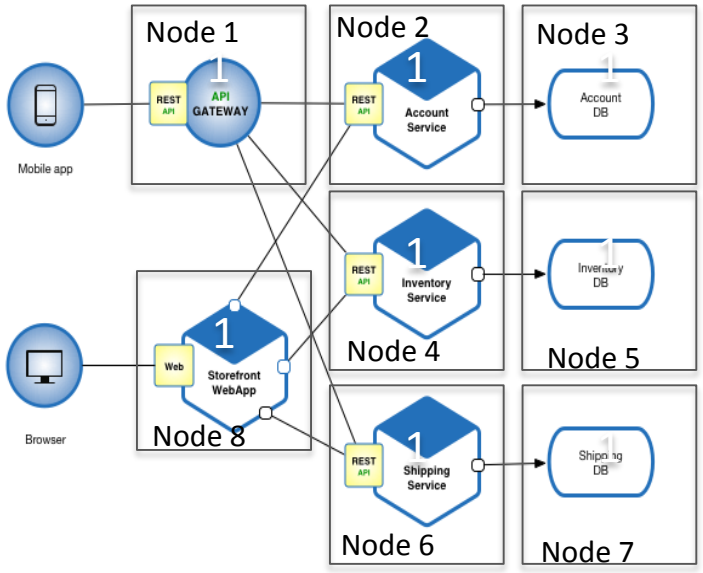


Klassische Architektur – Verteilung von Schichten

Maßnahmen zur Verbesserung der Skalierbarkeit: Schichten und Servicebildung

Problem: Parallelisierung von Aktivitäten innerhalb der Applikationslogik

- **Alternativ: Definition des Systems als Micro-Services / Service oriented Architecture**
  - Verteiltes System von (unabhängigen) Diensten/Komponenten
  - Komponenten implementieren Fachliche Funktionalität mit hoher Kohäsion
  - Kopplung der Dienste auf Basis von asynchroner/synchroner Kommunikation durch z.B. Broker/Mediatoren



Literaturverzeichnis

https://jaxenter.de/prinzipien-skalierbarer-architektur-848

https://www.informatik-aktuell.de/entwicklung/methoden/skalierbare-anwendungsarchitektur.html

https://www.it-agile.de/wissen/agiles-engineering/skalierbare-software-architekturen/

https://blog.codecentric.de/2011/08/grundlagen-cloud-computing-cap-theorem/

http://www.wikipedia.de