Rechnernetze Grundlage für VS, Webtechnologien Teilgebiet der VS

Warum?

Zuverlässigkeit, Verfügbarkeit, Redundanz

Nachteile:

Kosten (Geld, Latenz, Schwankung), Erhöhte Komplexität (Debugging, Timeouts)

Verteilungstransparenz -> System wirkt so als wäre alles an einem Ort

**Nebenläufigkeit:**

Quasiparallelität auf Einprozessorsystemen

Echte Parallelität auf Mehrkernprozessoren

Nebenläufigkeit in VS:  
Client: Asynchrones warten auf Serverantwort

Server: Parallele Abarbeitung von Anfragen

**Client:**

Synchron -> blockingReceive()

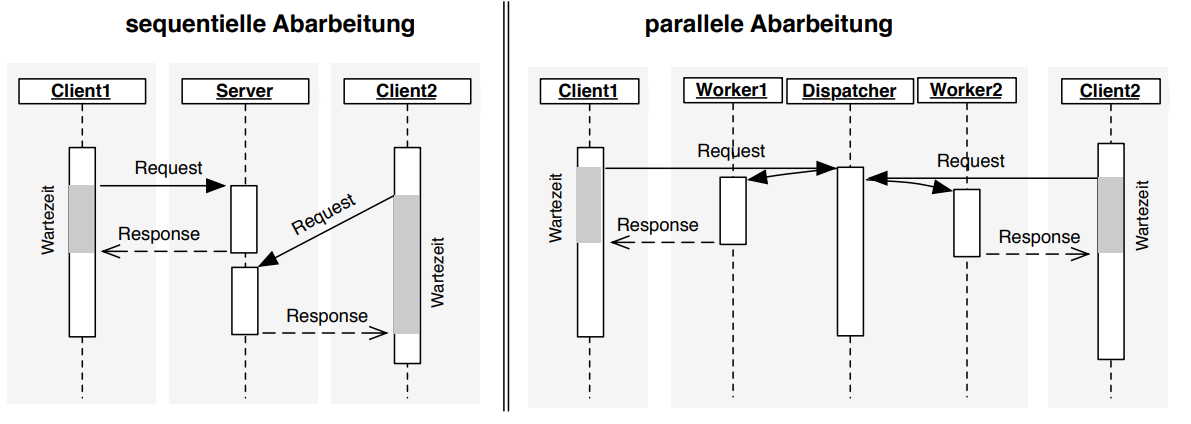
Asynchron, polling -> nonBlockingReceive() in Loop, break loop when getting response

Asynchron, nebenläufig -> Send Request, start Thread2, continue. Thread2 -> wait for response

**Server:**

Seriell: Wait for Request, answer it, wait for next Request

Parallel: Dispatcher waits for Request, give it to worker to answer it, wait for next Request



Threadpool:  
Statt pro request einen Thread zu starten und diesen anschließend zu verwerfen wird zu Beginn ein Pool erstellt. Alle Threads dieses Pools „bedienen“ sich aus einer Queue in der der Dispatcher die requests ablegt.

Dynamisch wachsender Threadpool:

Wie Threadpool, aber wenn z.B. die Queue zu lang wird werden neue Threads gestartet

POOL\_SIZE = Kerne / CPU-Intensität (der Tasks)

* Wenn die Bearbeitung der Requests zu 50% aus File-IO besteht können mehr Threads erstellt werden um die CPU voll auszulasten
* Beenden des Servers:

Flag -> while(!done) {request = blockingReceive()}

Nachteil: Wenn nach setzen der Flag keine Requests mehr kommen wartet der server ewig im blockingReceive()

Poison Pill:

Shutdown Aufforderung als Request

Nachteil (bösartiger) Client kann Poison Pill versehentlich (absichtlich) verschicken

**Verteilte Architekturen:**

Logische auf physikalische Komponenten abbilden, Verantwortlichkeit auf Netzkomponenten aufteilen

Client-Server-Architektur

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| Einfach | Einschränkend, kein dynamischer Rollentausch |
| Gut skalierbar bzgl. Administrativer Verteilung   * Wenige gleichbleibende Server | Zentralistisch |
|  | Single Point of Failure |
|  | Schlecht skalierbar bzgl. Größe   * Überlastung bei zu vielen Clients |

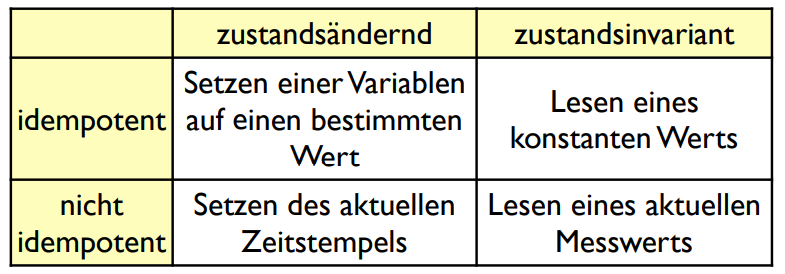
Bezogen auf einen Bestimmten Dienst ist ein Rechner entweder Client **oder** Server

Aber: Server kann einen anderen Dienst als Client in Anspruch nehmen

Verteilte Server:

Statisch konfiguriert -> Unterschiedliche Server für unterschiedliche Dienste

Dynamische Zuordnung -> Unterschiedliche Server für denselben Dienst um mehr Clients versorgen zu können

Anfragearten

zustandsändernd vs zustandsvariant

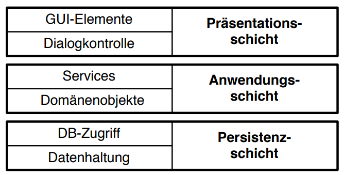
idempotent vs nicht idempotent

idempotent oder zustandsvariant ? -> Bei Fehlschlag nochmals Versenden, selbes Ergebnis

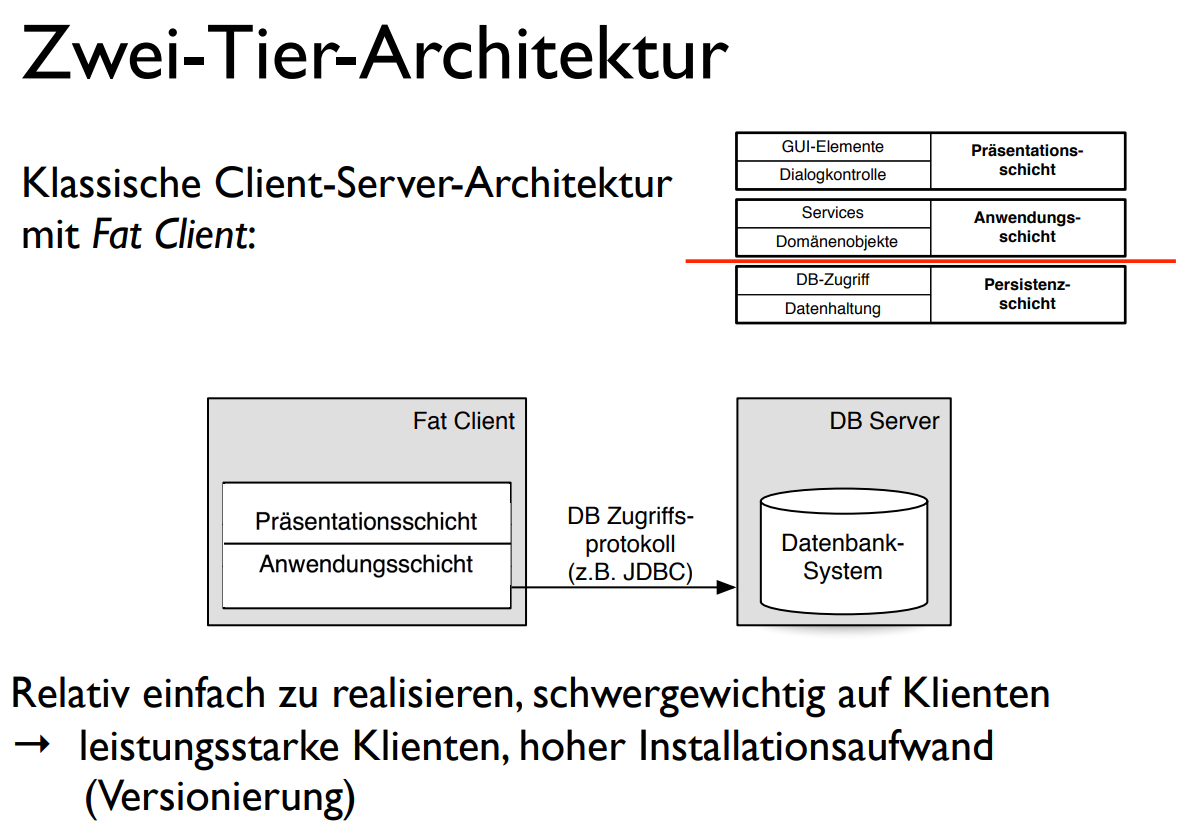
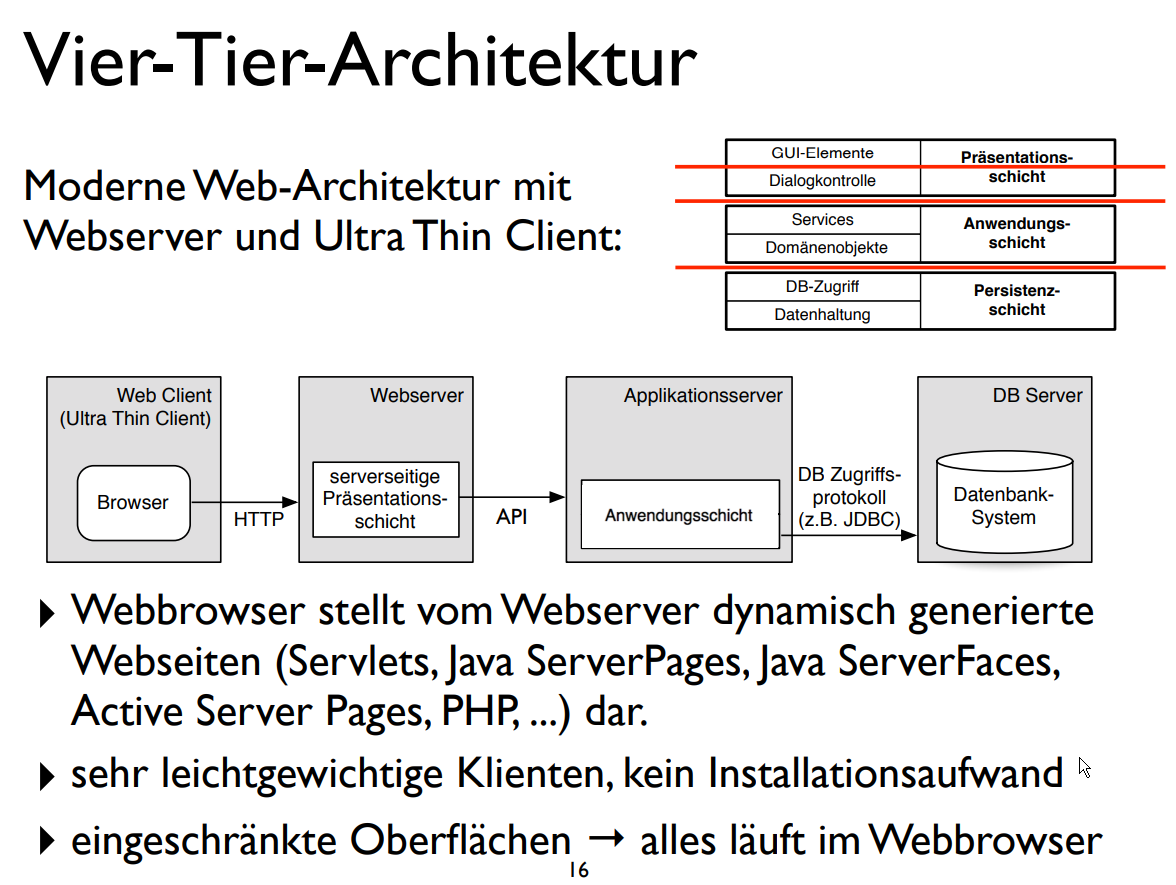
Zustandslose Anfrage: Server weniger belastet, robust gegen Server/ClientCrash

Zustandsbehaftete Anfrage: Schlankere Anfrage, Skalierbarkeitsproblem bei vielen parallelen Sessions

Wiederholungsanfrage kann durch Sequenznummer erkannt werden

Multi Tier Architekturen:

Dreischichten-Architektur verteilen



Peer-to-Peer-System

Jeder Knoten Server und Client

Konkrete Rolle ändert sich pro Kommunikation

Verteiltes System ohne zentralen Server

Neuer Peer bringt Last und Ressourcen -> gut skalierbar bzgl. Größe

Computer Cluster

Vernetzung räumlich konzentrierter Einzelrechner -> sehr schnelle (kabelgebundene) Verbindung

Netztopologie sehr wichtig, entscheidet über Skalierbarkeit, Routingkomplexität etc.

Hypercube Routing TODO

Cloud Computing

Daten und Anwendungen werden bei zentralem Cloud Anbieter gespeichert/ ausgeführt

* Internet als Vermittlungsinstanz

**Synchronisation:**

Nach einer Zeit t geht eine Uhr mit Driftrate p maximal p \* t vor/nach

Bsp: t = 20sec, p = 10^5 0 -> 20 \* 10^5 = 0,0002sec = 0,2msec

Synchronisation mittels UTC Empfänger -> sehrt genau

Wenn kein UTC Empfänger Synchronisationsalgorithmen (NTP Network Time Protocol)

Manchmal reicht logische Zeit ( was passierte vor was, also Reihenfolge)

**Algorithmen zum gegenseitigen Ausschluss:**

Zentraler Algo: Request an Kooordinator – Koordinator reiht anfragen in Queue und vergibt Token an Nr1, Nr1 betritt kritische Sektion, verlässt sie, gibt Token an Koordinator zurück.

Verteilter Algo: Request an alle Prozesse, alle müssen OK geben. Wenn gerade in kritischer Sektion OK verzögern bis verlassen, dann OK geben. Wenn 2 Prozesse zeitgleich wollen gibt der mit jüngerer Request OK.

Token Ring: Token wandert der Reihe nach durch alle Prozesse, wenn nicht benötigt sofort weitergeben, sonnst kritische Sektion betreten, verlassen, Token weitergeben.

**Replikation und Konsistenzmodelle**

Replikation: Mehrere Kopien halten für bessere Verfügbarkeit und bessere Effizienz (Verteilung) beim lesen

Konsistenzmodelle:

Strikt: „Göttlicher Blick“, letzter Write wird gelesen, Reihenfolge bekannt

Sequenziell: Read Reihenfolge in allen Prozessen gleich

Kausal: Reihenfolge nur gleich wenn kausaler Zusammenhang besteht. Zusammenhang besteht wenn Write in einem Prozess oder wenn Read vor Write (im selben prozess) stattfindet

FIFO: Zusammenhang nur bei Write innerhalb eines Prozesses, Read vor Write wird ignoriert.