

Lernportfolio

Joel Maximilian Mai

Matrikelnummer 11118561

25. April 2019

TH KOELN

WEB DEVELOPMENT



Inhalt

[Workshop 04.04.19 3](#_Toc7116694)

[Aktueller Lernstand: 3](#_Toc7116695)

[Im Workshop gelernt: 3](#_Toc7116696)

[Neue Lernziele: 3](#_Toc7116697)

[Proof of Concepts: 3](#_Toc7116698)

[Anforderungen: 3](#_Toc7116699)

[Versprechungen vom Dienst: 3](#_Toc7116700)

[Selbststudium mit dem Buch „Reactive Design Patterns“ 4](#_Toc7116701)

[Kapitel 1 bis 2 4](#_Toc7116702)

[Sharding-Patterns: 4](#_Toc7116703)

[Responsives System entwickeln: 4](#_Toc7116704)

[Integrating nonreactive Components: 5](#_Toc7116705)

[Reacting to Users: 5](#_Toc7116706)

[Understanding the traditional approach: 5](#_Toc7116707)

[Analyzing latency with shared resource: 5](#_Toc7116708)

[Limiting maximum latency with a queue: 5](#_Toc7116709)

[Exploiting parallelism: 6](#_Toc7116710)

[The limits of parallel execution: 6](#_Toc7116711)

[Reacting to failure 6](#_Toc7116712)

[Compartmentalization and bulkheading 6](#_Toc7116713)

[Using circuit breaker 6](#_Toc7116714)

[Losing strong consistency 6](#_Toc7116715)

[BASE 7](#_Toc7116716)

[Accepting Updates 7](#_Toc7116717)

[The need for Reactive Design Patterns 7](#_Toc7116718)

[Workshop 11.04.19 8](#_Toc7116719)

[Aktueller Lernstand: 8](#_Toc7116720)

[Domaindiagramm: 8](#_Toc7116721)

[Erster Aufbau des Services: 9](#_Toc7116722)

[Im Workshop gelernt: 9](#_Toc7116723)

[Neue Lernziele: 9](#_Toc7116724)

[Lernziele am 25.4: 9](#_Toc7116725)

[Workshop 25.04.19 9](#_Toc7116726)

[Aktueller Lernstand: 9](#_Toc7116727)

[Im Workshop gelernt: 9](#_Toc7116728)

[Neue Lernziele: 9](#_Toc7116729)

[Selbststudium mit dem Buch „Reactive Design Patterns“ 10](#_Toc7116730)

[Kapitel 4, 5 & 6 10](#_Toc7116731)

[Message Passing 10](#_Toc7116732)

[Messages: 10](#_Toc7116733)

[Vertical scalability: 10](#_Toc7116734)

[Event-based vs. message-based: 10](#_Toc7116735)

[Synchronous vs. asynchronous: 10](#_Toc7116736)

[Flow control: 11](#_Toc7116737)

[Delivery guarantees: 11](#_Toc7116738)

[Events as messages: 11](#_Toc7116739)

[Synchronous message passing: 11](#_Toc7116740)

[Location transparency 11](#_Toc7116741)

[What is location transparency?: 11](#_Toc7116742)

[The fallacy of transparent remoting: 12](#_Toc7116743)

[Explicit message passing to the rescue: 12](#_Toc7116744)

[Optimization of local message passing: 12](#_Toc7116745)

[Message loss: 12](#_Toc7116746)

[Horizontal scalability: 12](#_Toc7116747)

[Location transparency makes testing simpler: 12](#_Toc7116748)

[Dynamic composition: 12](#_Toc7116749)

[Divide and conquer 13](#_Toc7116750)

[Hierarchical problem decomposition: 13](#_Toc7116751)

[Defining the hierarchy: 13](#_Toc7116752)

[Dependencies vs. descendant modules: 13](#_Toc7116753)

[Building your own big corporation: 13](#_Toc7116754)

[Advantages of specification and testing: 13](#_Toc7116755)

[Horizontal and vertical scalability: 13](#_Toc7116756)

# Workshop 04.04.19

## Aktueller Lernstand:

* Basics in Javascript aus WBA1 bekannt
* Modellierung von Software aus ST1 bekannt
* Zielsetzung von Projektarbeit aus MCI bekannt

## Im Workshop gelernt:

* Asynchrone Kommunikation via Rabbit MQ und Node JS

## Neue Lernziele:

* Node JS im Selbststudium
* Reactive Design Patterns lesen
* How To Domainmodell

Nach dem wir uns das Verhalten des Systems vorstellen konnten, stellten wir uns allerdings die Frage wie wir weiter vorgehen sollen. Natürlich könnten wir weiter Modelle bauen, aber das würde unsere nicht vorhandenen Kenntnisse über XML/XMPP nicht besser machen.

Wir haben vor die Tutorials zu Javascript und XML zu erarbeiten:

* <https://www.w3schools.com/Js/js_intro.asp>
* <https://www.w3schools.com/xml/default.asp>
* <https://www.w3schools.com/nodejs/nodejs_intro.asp>
* GDW Inhalte aufarbeiten (da noch nicht teilgenommen)
* Reactive Design Patterns lesen

### Proof of Concepts:

* JavaScript/Java -> Programmiersprache?
* XMPP/XML -> Chat austausch und Veranstaltung

### Anforderungen:

* Eventbasiert – zumindest weitgehend
  + Asynchrone Kommunikation zwischen Veranstaltung und User
* Open Data Portale
  + Synchrone Datenabfrage jedoch asynchrone Weiterleitung

### Versprechungen vom Dienst:

* Nachrichtenaustausch über eine Gruppe der Veranstaltung
* Nachrichten direkt von der Veranstaltung
* Statusupdates der Verkehrs- und Wetterlage oder anderen Events im Umkreis
* Nachträgliche Modifikationen an den Veranstaltungsdetails

# Selbststudium mit dem Buch „Reactive Design Patterns“

## Kapitel 1 bis 2

* Responsive = Reaktion auf den User
* Resilient = Reaktion auf Versagen des Systems und das überkommen und aktiv bleiben dessens
* Elastic = Flexibel und Funktionstüchtig bei der Menge des Workloads
* Message-Driven = Reaktion auf verschiedene Inputs

### Sharding-Patterns:

* Kopieren eines Systems um Verfügbarkeit zu garantieren -> Replicate
* Active-Passive Replication:
  + Festlegen welches Replikat Updates akzeptiert
  + Dieser leitet dann das Update weiter an die Anderen
  + Erst wenn dieses ausfällt übernimmt ein anderes Replikat
* Consensus-based Multiple-master Replication:
  + Jedes Update wird von allen Replikas bestätigt und übernommen
  + Damit alle konsistent (Consistent) sind aber dadurch sind die anfällig für Verzögerungen (Latency) oder nicht Verfügbar(Availability) auf Grund des höheren Workloads
* Optemistic Replication (with conflict detection and resolution):
  + Mehrere aktive Repliken verbreiten Updates und führen Transaktionen zurück, wenn Konflikte auftreten oder verwerfliche Updates verworfen werden, die während einer Netzwerkpartition durchgeführt wurden
* Conflict Free Replication:
  + Dieser Ansatz schreibt Zusammenführungsstrategien so vor, dass Konflikte nicht definitionsgemäß entstehen können, jedoch nur die eventuelle Konsistenz erfordert und besondere Sorgfalt bei der Erstellung des Datenmodells erforderlich ist

### Responsives System entwickeln:

* Applikation läuft lokal und speichert Emails und bla lokal
* Synchronisiert mit Server/-n
  + Erzeugt Spannung zwischen dem Wunsch das System zu verteilen und responsiv zu bleiben
* Circuit Breaker Patterns
  + Beobachtet das Anfrageverhalten des Nutzers und schaltet sich zwischen, wenn zu viele Anfragen gemacht werden
  + Zusätzliche Features:
    - Flow Control
    - Backlogs
  + Anwendungsbereich von Circuit Breaker Patterns
    - Zwischen Anwendungsebene des Users und Webserver
    - Zwischen Webserver und Backend Services
* Wichtig ist auch die Einstufung des Features: Essential / Nonessential
* Avoiding the ball of mud:
  + Klassische Visualisierung eines Systems ist Frontend Backend
  + Jedoch durch die Erweiterung/Verteilung des Backends auf verschiedene Services
  + Kann ein Big ball of mud entstehen wenn die Kommunikationswege
  + Nicht frei sind
* Spezielle Designform: Message-Flow / Message-Driven

### Integrating nonreactive Components:

* Device Drivers
* API (Eher Synchron und daher können sie die Funktionalität der App blockieren)
* Daher wichtig: Resource-Management Patterns
  + Ständiger retrofit des Systems
  + Extra Threads, Prozesse oder Maschinen wenn nötig
  + Festlegen was bei einem Overload passieren soll
  + Auskapseln von APIS und sie auslagern so gut es geht um crashes zu vermeiden
  + IPC = Inter Process Communication
    - Pipes
    - Sockets
    - Shared Memory
* Verzögerungen beobachten und im Falle dieser:
  + Temporäre Fehler Meldungen
  + Warnungen in der Antwort
  + Nutzer benachrichtigen, dass
    - Später erneut versuchen
    - Oder über die Verzögerung
* Reactive Manifesto

### Reacting to Users:

* Nicht nur einzelne Komponenten betrachten sondern auch das Innere dieser
* Zusätzlich das Ziel haben sie so responsive wie nötig zu halten

### Understanding the traditional approach:

* Wenn Sie die Leistung eines Systems wie dieses optimieren, ist einer der Schlüsselparameter das Verhältnis von Anforderungsthreads zu Einträgen des Verbindungspools. Es macht nicht viel Sinn, den Verbindungspool größer als den Request-Thread-Pool zu machen.
* Verbindungspool zu klein -> Bottleneck
* Die beste Antwort für eine gegebene Last liegt irgendwo zwischen den Extremen. Der nächste Abschnitt befasst sich mit dem Finden eines Gleichgewichts.

### Analyzing latency with shared resource:

* Bei Anfragen die darauf warten prozeduiert zu werden, benötigt man immer eine Art von Queue Date Structure
* Lösungsansätze:
  + Controller
  + TCP Buffer
  + Cache
  + Fallback
* WICHTIG: Immer auf Anfragen Overload vorbereiten und System beschützen

### Limiting maximum latency with a queue:

* Erster einfacher Schritt: If no database connaction available -> Return null
* Einführung einer Explicit Queue
  + Statt direkt einen Error zu senden können hier noch Anfragen zwischen geparkt werden
  + Erst wenn diese Queue voll ist wird abgewiesen

### Exploiting parallelism:

* Meisten Programme erwarten dass ihre Funktionen eine Anwort liefern, tun sie dies nicht funktioniert das ganze Programm nicht
* Lösung:
  + Subtasks einzeln betrachten und parallel schalten -> Die Aufgabe dauert nur so lange wie die Längste Subtask
  + Und das Ergebnis nicht als zwingend passend betrachten -> Try Catch
  + Fehlerwerfende Threads suspendieren um den Platz für neue Threads freizugeben

### The limits of parallel execution:

* Um von diesem Wachstum zu profitieren, müssen Sie die Berechnungen sogar auf einer einzigen Maschine verteilen.
* Ein herkömmlicher Ansatz ist wenn ein gemeinsamer State verwendet wird, der auf gegenseitigem Ausschluss durch Sperren basiert, werden die Kosten für die Koordinierung zwischen Kernen sehr CPU-bedeutend.

### Reacting to failure

* Software will fail
* Hardware will fail
* Humans will fail
* Timeout is failure
* Resillience:
  + Die Fähigkeit einer Substanz oder Objekt wieder zurück in eine vergange Form sich selbst zu bringen
  + The Kraft sich schnell von Schwierigkeiten oder Anstrengungen zu erholen
* Wichtige Teilaufgaben eines Systems zu deligieren
* Distribute and compartmentalize
* Wichtige Daten in kopierter Form zu sichern
* Wichtige Kopien von Systemzentren auf verschiedene Stromnetze oder gar Länder oder Kontinente zu verteilen

### Compartmentalization and bulkheading

* Falls die Compartments nicht hundert prozentig von einander getrennt sind, kann ein Systemfehler in einem von ihnen das gesamte System zum crashen bringen.

### Using circuit breaker

* Wenn die Antwort einer Anfrage bereits nicht mehr benötigt wird, ist es sinnlos sie überhaupt noch zu bearbeiten
* Falls der Circuit Breaker eine Verzögerung im Systemablauf feststellt kann er Anfragen von akzeptiere-alle auf wartet-hier-und-kommt-nach-der-wartezeit-dran oder gar ich-nehme-nichts-mehr-an umstellen um dem System Zeit zu geben sich zu erholen und die eigenen Warteschleifen zu leeren
* Wie viele Reconnection-Attempts machen Sinn?
* Wie lange sollte man zwischen ihnen warten?

### Losing strong consistency

* Nimm zwei von drei:
  + Konsistenz – Alle Kopien beeinhalten identischen Inhalt
  + Hohe Verfügbarkeit der Daten für Updates
  + Toleranz zu Netzwerkpartitionen

### BASE

* Verteilte Systeme sind meist auf verschiedenen Prinzipien gebaut
* Eins davon heißt BASE
  + Basically available
  + Soft State (Ein Zustand der aktiv versucht erreicht zu werden statt von einem Default Zustand auszugehen)
  + Eventually Consistent
* Der Letzte Punkt bedeutet, dass zwar zu manchen Zeitpunkten ein inkonsestenter Zustand herrschen kann, dieser jedoch mit der Zeit wieder konsestent wird.
* BASE setzt sich zusammen aus
  + Associative
    - Jede Aktion kann in Stapeln erfolgen (Assoziativ)
  + Commutative
    - Jede Aktion kann in jeder Reihenfolge erfolgen (Kommutativ)
  + Idempotent
    - Jede Aktion kann beliebig oft durchgeführt werden
  + Distributed

### Accepting Updates

* Wichtig hierfür sind CRDTs (Conflict-Free-Replicated-Data Types)
* Wie man zwei Dokumente wieder konsistent bekommt
  + Die beiden Änderungsketten nicht simultan bearbeiten sondern über einander legen
  + Erst die eine Kette anwenden und dann die Andere
  + Bis beide Partitionen „geheilt“ sind
* Wenn das System offline oder einfach unerreichbar ist, ist die Lösung dem Nutzer immernoch manche Feautures anzubieten.
* Man könnte auch das ganze so sehen, dass das System auf einen ungefähren Zustand wechselt bis eine Verbindung wieder hergestellt werden konnte

### The need for Reactive Design Patterns

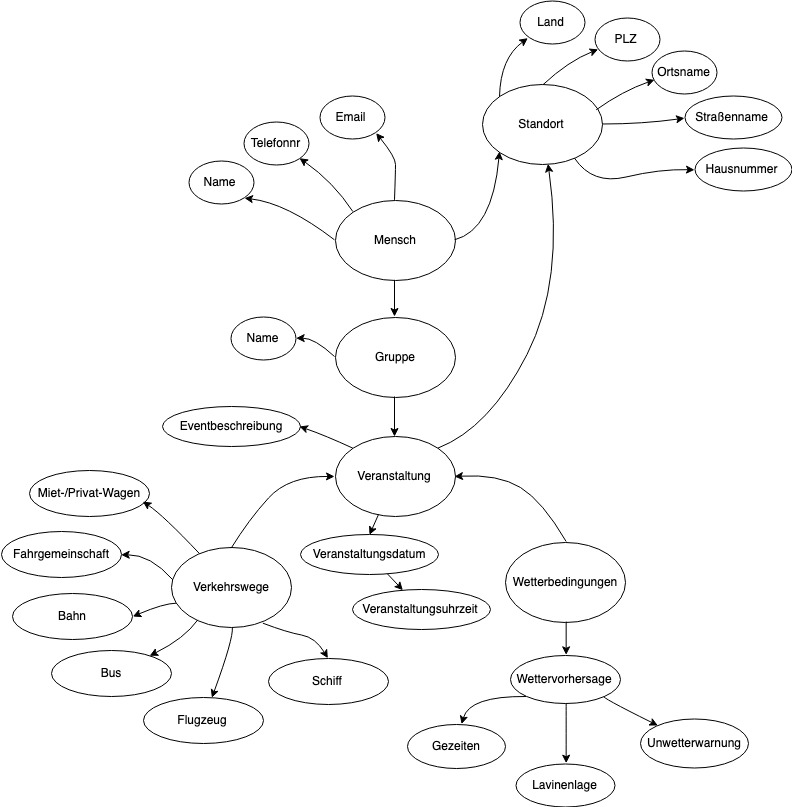
* SOA = Service-Oriented Architecture
* Managing Complexity
  + Essential Complexity = Ausgelöst durch ein Problem
  + Incidential Complexity = Ausgelöst durch die Lösung
  + Message oriented Development führt genauso zu einem konsistenten Service wie die Synchronen Entwicklungen und weil sie deutlich leichter sind zu implementieren werden sie immer öfter benutzt

# Workshop 11.04.19

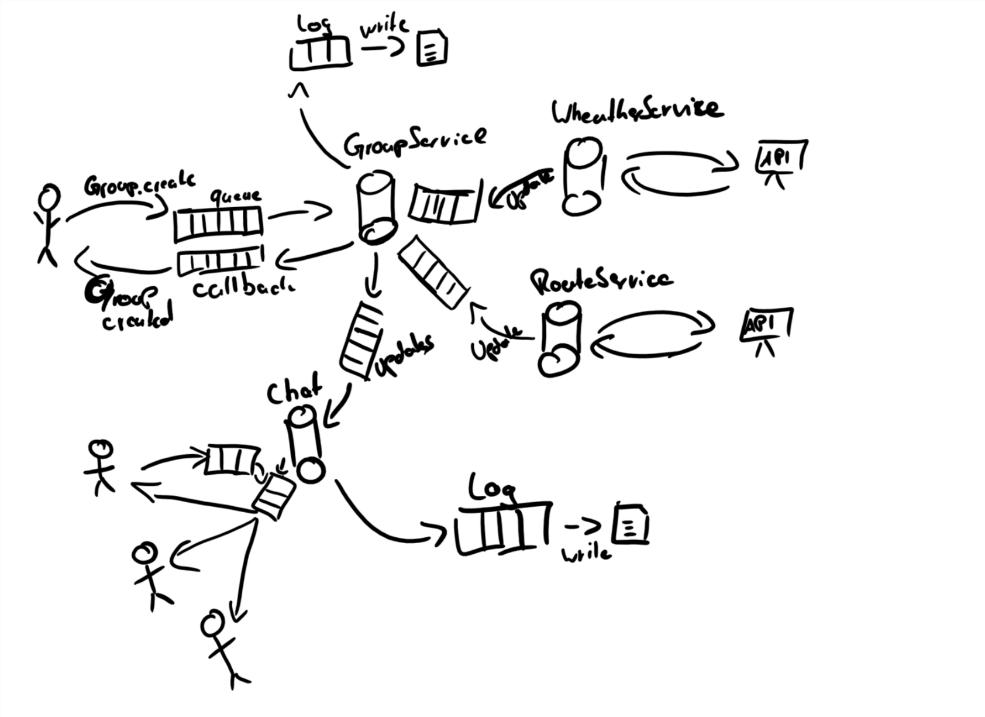
## Aktueller Lernstand:

* Node JS Basics
* Rabbit MQ und Readline
* Reactive Design Patterns zusammengefasst

### Domaindiagramm:



### Erster Aufbau des Services:



## Im Workshop gelernt:

* Readline Basics

## Neue Lernziele:

* Node JS + RabbitMQ zum laufen bringen
  + Nodejs lernen und alles nötige installieren
  + Client aufstellen (proto)
    - <https://www.rabbitmq.com/tutorials/tutorial-one-javascript.html>
  + Servicemodule aufstellen (proto)
    - <https://www.rabbitmq.com/tutorials/tutorial-five-javascript.html>
  + Beschränkte API-Einbindung erreichen (issues)
* Gedanken machen wie wir den Service aufbauen
* Domainmodell fertigstellen
* Kapitel 4, 5 & 6

### Lernziele am 25.4:

* Servicemodule und Client komplexieren
* Alle auf denselben Stand bringen
* Lernportfolio fertigstellen

# Workshop 25.04.19

## Aktueller Lernstand:

* Server Client verstanden und angewandt
* API Einbindung erfolgreich
* GitHub Issues angewandt

## Im Workshop gelernt:

* Telegram API Einbindung

## Neue Lernziele:

* Erweiterung des Skripts durch tatsächliche Feature

# Selbststudium mit dem Buch „Reactive Design Patterns“

## Kapitel 4, 5 & 6

* Event =
* Producer =
* Consume =
* Message =

## Message Passing

### Messages:

* Immutability (Unveränderlichkeit) hat oberste Priorität
* Messages werden in Form von Message Queues versendet, um bearbeitet werden zu können
* Anders als bei Shared-Memory Systemen gibt es keinen gemeinsamen Speicher
* Messages werden versendet:
* An andere Computer
* An andere Prozesse
* Innerhalb des eigenen Prozesses

### Vertical scalability:

* Message Passing (Nachrichtenaustausch) entkoppelt Sender und Empfänger
  + Dadurch muss der Sender nicht wissen, wie der Empfänger mit der Message (Nachricht) umgeht und sie verarbeitet oder ob der Empfänger die Message parallel verarbeiten kann oder nicht -> Muss also keine „Rücksicht nehmen“
  + So kann beispielsweise die Rechenkapazität des Empfängers unabhängig vom Sender beliebig ausgebaut werden und die Verarbeitung auf mehrere Einheiten verteilt werden, um effizienter zu sein.

### Event-based vs. message-based:

* Event-Driven
  + Ein Event ist ein ausgelöstes Signal sobald ein Ereignis eintritt.
  + In einem event-basierten System gibt es sog. Listener die aufgerufen werden sobald ein Ereignis eintritt auftritt.
* Message-Driven
  + Eine Message (Nachricht) sind Daten, die an einen bestimmten Empfänger gesendet werden
  + Ein message-basiertes System wartet auf die Ankunft von Nachrichten, um darauf zu reagieren.

### Synchronous vs. asynchronous:

* Synchrone Kommunikation
  + Beide Parteien müssen bereits sein, miteinander zu kommunizieren
    - Sender stellt Anfrage und wartet auf eine Antwort
* Blockiert den Prozess, bis die Kommunikation abgeschlossen ist
* Asynchrone Kommunikation
  + Sender kann senden ohne, dass der Empfänger dafür bereit ist
    - Sender stellt Anfrage, wartet aber nicht auf eine Antwort
      * Blockiert den Prozess nicht, wodurch die nächste Aufgabe durchgeführt werden kann

### Flow control:

Die Flow Control (Flusskontrolle) gibt dem Empfänger von Daten die Möglichkeit, den Sender über eine Überlastsituation (Overflow) zu informieren und darauf zu reagieren, beispielsweise die Übertragungsrate zu verringern oder das Senden einzustellen bis das Problem behoben ist, sodass keine Daten verloren gehen.

### Delivery guarantees:

* Man muss immer von der Möglichkeit ausgehen, dass Nachrichten verloren gehen können
  + Auch in synchronen Systemen
* Prinzipiell gibt es die folgenden Möglichkeiten zum Garantieren von Nachrichtenaustausch
  + At-Most-Once Delivery
    - Jeder Request wird ein Mal gesendet, wenn er verloren geht oder der Empfänger dabei scheitert ihn zu verarbeiten gibt es keine Möglichkeit zur Wiederherstellung
  + At-Least-Once Delivery
    - Der Versuch, eine Verarbeitung eines Requests zu garantieren
    - Benötigt eine Bestätigung des Empfängers
    - Der Sender muss den Request behalten, um ihn erneut zu senden, falls keine Bestätigung des Empfängers eintrifft
    - Empfänger können einen Request mehrmals erhalten, wenn das Senden der Bestätigung fehlschlägt und der Sender ihn anschließend erneut sendet
  + Exactly-Once Delivery
    - Ein Request muss und darf nur genau einmal verarbeitet werden
    - Der Empfänger muss Informationen darüber haben, welche Requests er bereits bearbeitet hat

### Events as messages:

* Eine Nachricht die versendet wird und ankommt kann mal als Event sehen.
* Da sie weitergeleitet werden können, spricht man auch von einem Event-driven System - da die Events die einzelnen Komponenten verbinden.
* Da viele PC-Komponenten sowieso schon mit Messages/Events arbeiten ist es so gesehen die 'natürlichste' Form der Kommunikation unabhängiger Komponenten.

### Synchronous message passing:

Solange keine asynchrone Kommunikation erforderlich ist, ist die asynchrone Übergabe von Nachrichten an entkoppelte Systeme unnötig und führt zu einem größeren Aufwand, als durch eine synchrone Weiterleitung erfolgen würde.

## Location transparency

* Warum sich darauf beschränken nur Threads zu trennen?
  + Systeme auch trennen
  + Dies eröffnet neue Blickwinkel für Performance der Systeme

### What is location transparency?:

* Source-Code zum senden einer Nachricht sieht gleich aus, ohne Rücksicht darauf ob der Empfänger es verarbeiten kann.
* Die Knoten können beschränkungslos sich Nachrichten schicken da Diese über die gleiche Art verschickt werden. Lediglich der Weg und Inhalt ist eigen.

### The fallacy of transparent remoting:

* Schon lange ist das Ziel von Computer Netzen, die Nachrichten so anzulegen dass sie sowohl lokal als auch remote(also aus der ferne) gleichermaßen verschickt und verarbeitet werden können 🡪 Einheitlichkeit
  + Transparent Remoting
* Das Problem ist, dass ein Funktionsaufruf lokal nur zu exception oder Resultat führen kann, jedoch über ein Netzwerk deutlich mehr schief gehen kann.
  + Nicht richtig aufgerufen
  + Verloren gegangen
  + Oder fehlerhaft versendet
* Regelfalllösung: Timeout Exception erhöhen
* Performanceproblem ist: Der Aufruf muss erst versendet werden und dann auf die Antwort gewartet werden
* Performanceproblem: Dateigröße die übers Netz gesendet werden kann ist kleiner als die Lokale

### Explicit message passing to the rescue:

* Location Transparency’s Ziel ist es den Lokalen und Globalen Nachrichtenaustausch zu vereinheitlichen und zu abstrahieren
* Mit location transparency ist jede Nachricht potentiell eine übers Netz geschickte
  + Aber da man auf eine Antwort nicht wartet ist hier alles erledigt
* Mit transparent remoting hofft man bei jedem Aufruf auf eine erfolgreiche Antwort
* Mit location transparency kann man die software überall laufen lassen ohne probleme mit der verbindung zu haben zu den anderen teilservices

### Optimization of local message passing:

* Um Verzögerungen zu verringern kann man den Nachrichtenaustausch hier auch über Referenzen vollziehen
* Falls Lokal implementiert, ist es meist sinnvoller sich and den oben genannten Punkt zu orientieren und nicht mit Nachrichtenaustausch zu arbeiten – da man hier die sende und empfangszeit sparen würde

### Message loss:

* Nachrichten die über Netzwerke versendet werden, können über viel mehr Wege verloren gehen
* Falls nicht alle Antworten einer Anfrage wieder ankommen 🡪 Überlegen was getan werden soll
  + Wiederstand des Systems gesteigert

### Horizontal scalability:

* Mehr Server einzurichten die das selbe tun kann die Performance erhöhen
* Da Location Transparency 🡪 Unabhängig vom Standort

### Location transparency makes testing simpler:

* Da man mehrere Server hat die das selbe tun kann man sich einen rauspicken und tests durchführen ohne das system vom netz zu nehmen

### Dynamic composition:

* Parallelisierung von Abfragen durch Location Transparency
* Und Fallbacks durch zusätzliche Horizontal Scalability

## Divide and conquer

### Hierarchical problem decomposition:

* Bei dieser Applikation werden Probleme in kleinere aufgeteilt.
* Nicht immer Ideal da die resultierende Anzahl der Probleme überwältigend werden kann.

### Defining the hierarchy:

* Aufteilung der Hierarchy
* Am wichtigsten sind die logischen Funktionen welche Implementiert werden
* Die mit der niedrigsten Priorität sind die kleinsten „Details“ welche keinen starken Einfluss auf das System haben.
* Die Beziehung zwischen einem Modul und dessen Nachfolger ist mehr als nur eine Notwenigkeit
  + Erlaubt die Bearbeitung eines spezifischen Problems ohne Gefahren
* Höherer Rang = wahrscheinlicher für spezifischen Use-Case

### Dependencies vs. descendant modules:

* „Ownership“ ist ein wichtiger Punkt
  + Es sollte ein Modul vorhanden sein mehreren Implementierungen \*Andere Module welche auf Funktionen des ersten Moduls zugreifen wollen sind Abhängig/ haben Abhängigkeiten

### Building your own big corporation:

* Metapher
* Ohne ordentliche Trennung der Segmente hinsichtlich ihrer Verantwortung und Aufgabe entstehen Konflikte zwischen ihnen
* Aufgaben werden von höherem Rang übernommen wenn der untere Ausfällt
  + Eigene Zuweisung der Zusammenhänge

### Advantages of specification and testing:

* Jede Aufgabe eines Moduls muss eindeutig sein
* Tests sind ein Notwendiger Bestandteil
  + Von test driven development auf testability driven design wechseln
  + Dadurch wird besseres Design erzielt
  + Divide et regna sollte Module hervorrufen welche einfach zu testen sind

### Horizontal and vertical scalability:

* Vorhandenen Protokolle kommunizieren via message passing
* Frei wählbare Größe der Latenz um die Suchanfrage zu optimieren
  + Wichtig weil eine Instanz pro Nutzer ist nicht wirtschaftlich
    - Mehrere Nutzer auf einer Leitung, größe der Latenz sollte nach Minimalprinzip gewählt werden