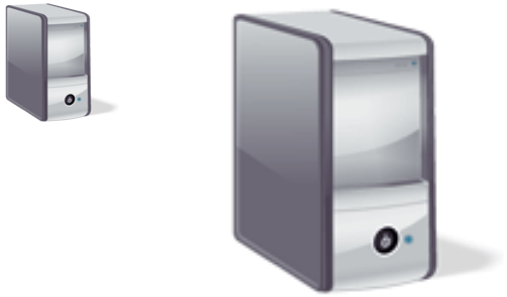
Grundlagen, Patterns und Algorithmen

- Skalierung
- Parallelität / Dependencies
- Patterns / Anti Patterns
- Algorithmen
- Ausflug Verteilte Systeme
  - Web Services
  - (Verteilte) Daten

Skalierung
Was in welchen Dimensionen

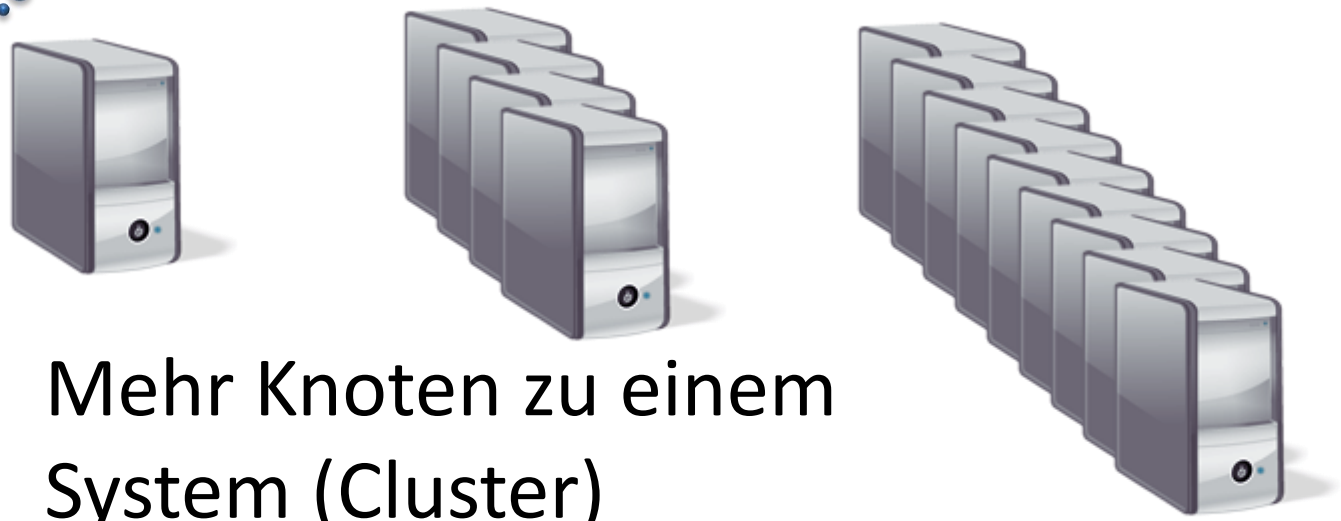
- CPU, GPU (Instructions/s)
- RAM (I/O, Größe)
- Storage (I/O, Größe)
- DB (Transaktionen/s, Größe)
- Netzwerk (Durchsatz, Delay, RTT)
- Algorithmen

Vertikale Skalierung (Scale Up)



- Mehr Ressourcen zu einem einzelnen Knoten
- Keine Änderung der SW!!!
- (physikalische) Grenzen

Horizontale Skalierung (Scale Out)



- Mehr Knoten zu einem System (Cluster)
- Skaliert (fast) ohne physikalische Grenzen
- Grenzen durch die Parallelisierbarkeit des Problems (Amdahl´s law)

=> Wir brauchen Beides

- Tradeoffs
  - Management Overhead
  - SW Komplexität
  - Parallelisierbarkeit des Problems
  - Bandbreite der Kommunikationskanäle
  - Preis

Parallelisierung

- Bei Scale Out sind immer zu klären:
  - Verteilung des Codes
  - Verteilung der Daten
  - Zugriff auf die Daten
- Verschiedene Granularitäten:
  - „CPU“ Instruktionen / Daten
    - SISD (Single Instruction Single Data) – 1 Kern
    - SIMD (Single Instruction Multiple Data) – GPU
    - MISD – eher exotisch
    - MIMD – Mehrkern CPUs
  - Tasks / Threads
  - Jobs, z.B: „Map Reduce“ Frameworks

Herausforderungen:

- **Verteilung von Code**
  … der natürlich auf Daten zugreift
- Verteilung von Daten
- Zugriff auf Daten & Code

Spielverderber der Parallelisierung

- Amdahls Law: wenn
  p = der O(n) parallelisierbare Teil eines Programmes
  s = der sequenzielle (nicht parallelisierbare) Teil des Programmes ist
  dann ist der Maximale Geschwindigkeitsgewinn durch:
  s+p / s+p/N bei N Cores/CPUs gegeben.
- Das gilt selbst wenn der Kommunikationsoverhead etc. vernachlässigbar ist
- Darin nicht berücksichtigt:
  - CPU Caches (wenn der limitierende Faktor nicht die Cycles sondern der Speicher ist),
  - Branch Prediction, etc.

Spielverderber der Parallelisierung

- Datenabhängigkeiten
  - Flow Dependency (read after write):
    a=1;
    b=a;
    c=b;
  - Anti Dependency (write after read):
    a=1;
    b=a+1;
    a=2;
  - Output Dependency (write after write)
    gleiches Beispiel
- Control Dependencies

Spielverderber der Parallelisierung

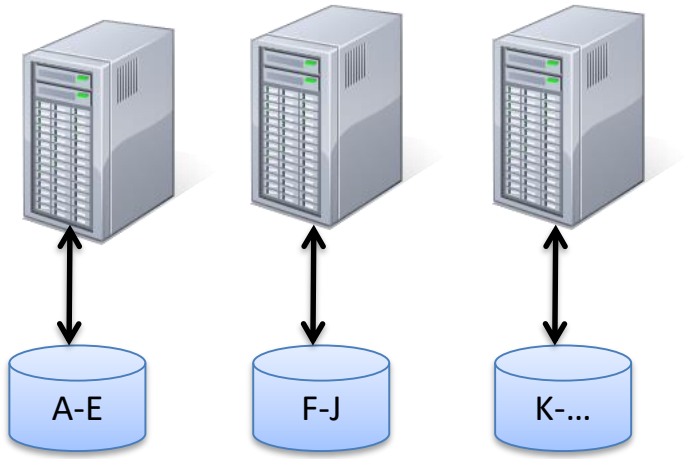
- Wir benötigen:
  - Locks
  - Synchronisation
  - Critical Sections
- Und haben (meist schwer zu findende):
  - Race Conditions
  - Deadlocks
  - Dirty Caches

Herausforderungen

- Verteilung von Code
- **Verteilung von Daten**
- Zugriff auf Daten & Code

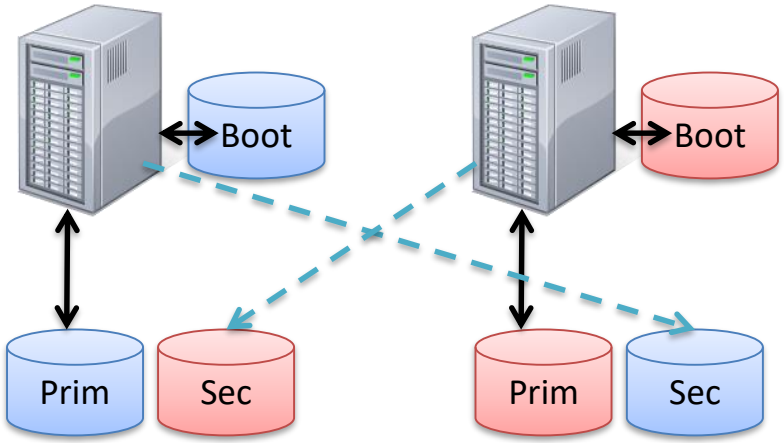
Pattern: Shared Nothing

- Daten sind partitioniert (Sharding) und lokal zum verarbeitenden Knoten
- (unendlich) skalierbar
- Gut bei hoher Lese/Schreibe Last
- Schlecht bei Transaktionen über das Cluster
- Schlecht bei Datenverknüpfungen
- Weitere Kapazität Ausbau „teuer“
- HA nur beschränkt möglich
- Schwer zu „balancen“



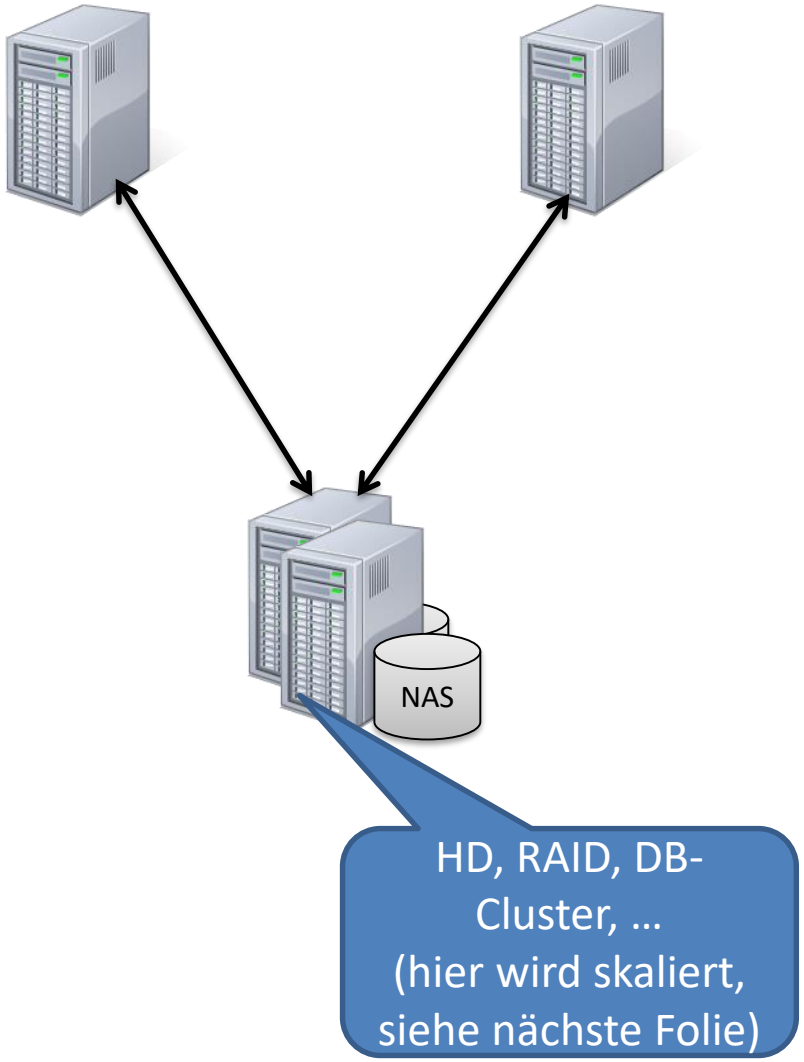
Pattern: Shared Nothing Setup (für HA/Failover)

- Z.B. DRBD



Pattern: Shared Disk Architecture

- Adaptiert schnell bei unterschiedlichen Last Szenarien (dynamisches Loadbalancing)
- Failover / HA ist einfach
- Schlecht bei hoher Schreibe Last (dirty Caches in den Knoten)
- Preis der Skalierung abhängig von der eingesetzten Technologie



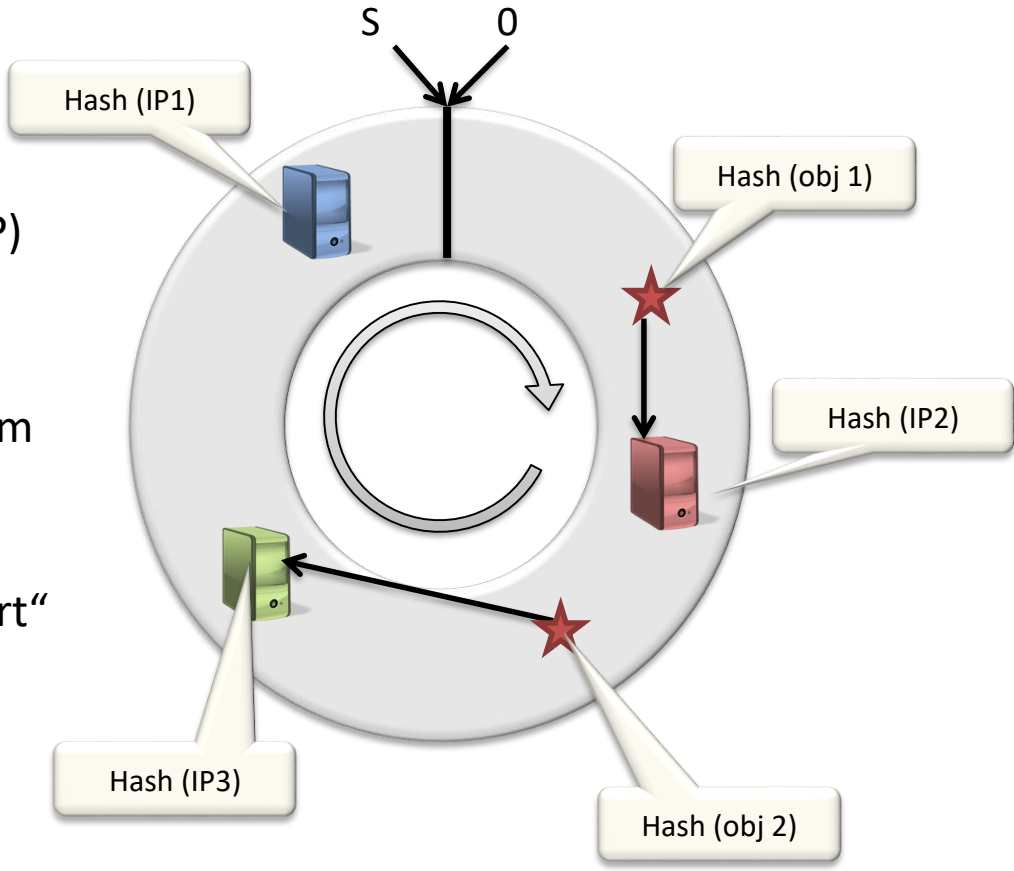
Algorithmus: Consistent Hashing

- Problem: Wie finde ich den Speicherort für ein Objekt in einem verteilten System mit n Knoten?
  - Z.B. bei Caches, (NoSQL) DBs, verteilte Filesysteme, …
- Naive/klassische Lösung:
  hash (Objekt) mod n
- …Funktioniert gut wenn n konstant ist
  wenn nicht (durch Ausfall oder Scale Out)…
- … Funktioniert gut wenn alle Server gleich dimensioniert sind

Publikumsfrage – Wie viele Hashes/Keys werden bei einem Ausfall ungültig?

Algorithmus: Consistent Hashing

- Knoten werden im Adressraum/Hashtabelle (0..S) angeordnet (z.B. durch Hashen der IP)
- Objekte werden mit der gleichen Hashfunktion gehashed
- Der Zuständige Knoten ist der mit dem nächsthöheren Hashwert (im Uhrzeigersinn der nächste)
- Knoten werden mehrfach „virtualisiert“ im Ring angeordnet (z.B durch Hash („IP1-1“), Hash („IP1-2“), …)
- Anzahl der virtuellen Punkte pro Knoten kann auch anhand der Leistungsstärke variiert werden
- Anordnung kann auch anhand von „Partitionen“ erfolgen



Algorithmus: Consistent Hashing

- Gleichmäßige Verteilung der Last wenn sich die Anzahl der Knoten ändert
- Dynamische Anpassung der Last durch die Anzahl „virtueller Punkte“ möglich => langsames Anfahren
- Redundanz / Replikation durch weitere Schreib Vorgänge beim „nächsten“, „übernächsten“ etc. Knoten.
- Dadurch auch bereits „gefüllter Cache“ möglich

Publikumsfrage – Wie viele Hashes/Keys werden bei einem Ausfall ungültig?

Spielverderber: CAP / Brewer's Theorem

- Ein Verteiltes System kann nur zwei der folgenden drei Eigenschaften erfüllen:
  - Konsistenz (C): Alle Knoten sehen zur selben Zeit die selben Daten
  - Verfügbarkeit (A): Alle! Anfragen an das System werden stets beantwortet
  - Partitionstoleranz (P): Das System setzt keine perfekte Kommunikationsinfrastruktur voraus. Nachrichten können verloren gehen das Netz kann Partitioniert werden.

ACID / BASE

- Die klassische DB Anforderung ist ACID:
  - Atomic: Alles oder nichts
  - Consistent: Vorsicht hier im Sinne von Integrität der Daten untereinander bei CAP im Sinne der Daten innerhalb verteilter Systeme!
  - Isolated: um Transaktionen zu „parallelisieren“
  - Durable: im Sinne von persistent
- Verwendet Dinge wie:
  - Locks
  - Commit / 2Phase Commit
  - Rollbacks
  - Transaktions-Logs
- D.H:
  - Systeme welche ACID unterstützen sind nicht Partitionstolerant und haben Einschränkungen bei der Availability

ACID / BASE

- Viele NoSQL DBs unterstützen BASE – verzichten auf strikte Consistency von CAP
  - Basically available
  - Soft State
  - Eventual Consistency!
- Der Übergang kann auch fließend zwischen A/C – Consistency vs Availability gewählt werden

Algorithmus: Multiversion Concurrency Control

- Löst das Problem des konkurrierenden Zugriffs ohne „Locks“
- Daten sind „Immutable“: Schreibzugriffe erzeugen neue Version.
- Lesen: funktioniert immer – eventuell bekomme ich eine „alte“ Version
- Schreiben: Eine Transaktion kennt die dafür gelesene Version des Objekts. Ist diese nicht mehr aktuell wird die Transaktion abgebrochen.

Publikumsfrage – welche Probleme machen Locks in Bezug auf Skalierung?

Arten der Konsistenz in verteilten Systemen

- Client Sicht:
  - Monotonic Read Consistency: System liefert niemals ältere Version bei Leseanfragen an den gleichen Schlüssel.
  - Monotonic Write Consistency: System garantiert die gleiche Schreib-Reihenfolge für alle Knoten (Replikas)
  - Read Your Writes Consistency: System liefert einem Client der eine Schreiboperation ausgeführt hat keine ältere Version des Datums.
  - Write Follows Reads Consistency: Das System garantiert, dass ein Schreibvorgang auf ein Datum in Version X auf anderen Knoten (Replikas) nur ausgeführt wird wenn dort das Datum auch in Version X vorliegt.
  - Strict Consistency: Leseoperation liefert immer den neusten Wert (Ergebnis der Letzten Schreiboperation)

Herausforderungen

- Verteilung von Code
- Verteilung von Daten
- **Zugriff auf Daten & Code**

Spielverderber: Zugriff auf Daten

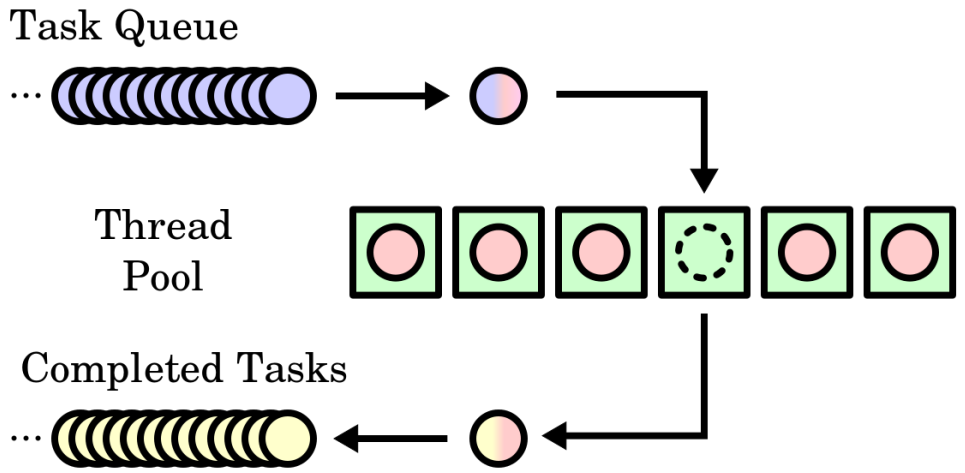
- Netzwerk, Netzwerk, Netzwerk (bzw. I/O)!
  - Ist nicht unendlich schnell
  - Die Latenz ist nicht 0
  - Die Latenz ist nicht konstant
  - Ist nicht immer Verfügbar
- Auswege:
  - Resilienz
  - Algorithmische Optimierung des Datenzugriffs!

Pattern: Resilienz /Stabilität

- Retrys / Timeouts – Beschränkt die Belegung von Ressourcen
- „Circuit Breaker“ – Verschwende keine Ressourcen
- Handshaking /Flow Control – passt die Load an die Ressourceverfügbarkeit an
  - Client-Throtteling, Server controlled scheduling
- 'Bulkheads' – Isoliert die Ressourcen um Seiteneffekte zu verringern
- Details:
  - http://www.javaworld.com/article/2824163/application-performance/stability-patterns-applied-in-a-restful-architecture.html

Pattern : Thread Pool Pattern

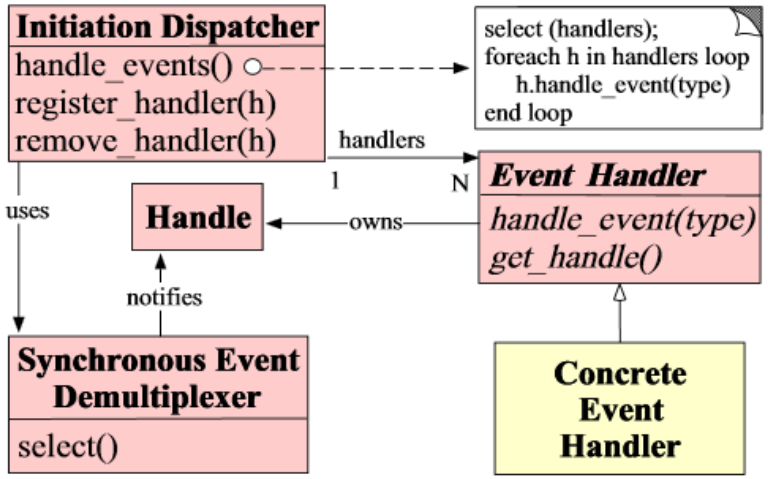
- Vorteile:
  - Optimiert den Zyklus:
    - Request
    - CreateThread
    - Process
    - DestroyThread
  - Systemunabhängige Programmierung
  - Systemabhängige Konfiguration (Größe)
  - **Isolation** !
- Nachteile:
  - (Speicher) Overhead
  - Synchronisation der Queues
  - Feeding slow Clients!
- Beispiel:
  - Apache prefork (default)



Quelle: Wikipedia - http://en.wikipedia.org/wiki/Thread_pool_pattern

Pattern : Reactor

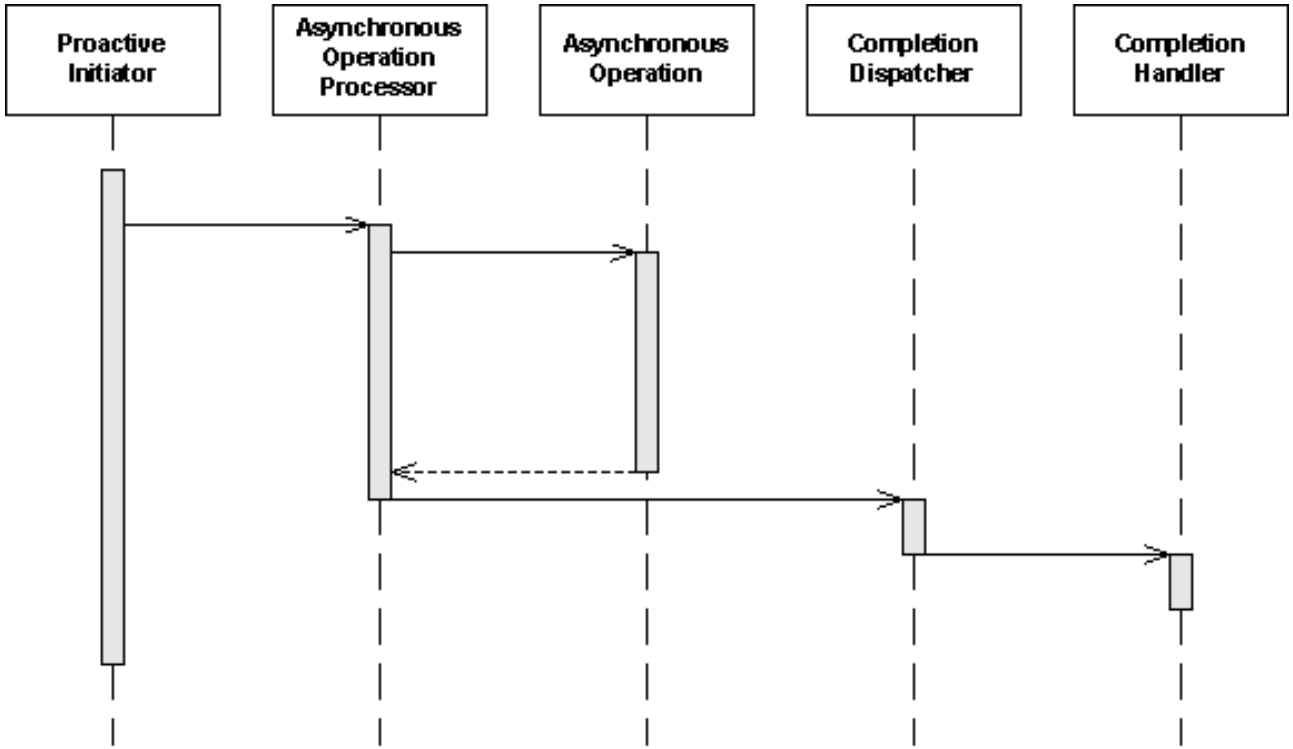
- Reactor:
  - Annahme und Dispatchen
    Asynchroner Requests
  - Serialisierte Abarbeitung
- Vorteile:
  - Belegt keine (wenige) Ressourcen bei langsamen Clients
  - Kosten für weitere Requests billig (keine Kontext Switche)
- Nachteile:
  - Non-Preemptive => kein Blocking I/O in den Event Handlern
  - Wenig Isolation
  - Schwer zu debuggen
- Beispiele:
  - Nginx , Gui- Event Loop / Game Loop



Quelle: http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf

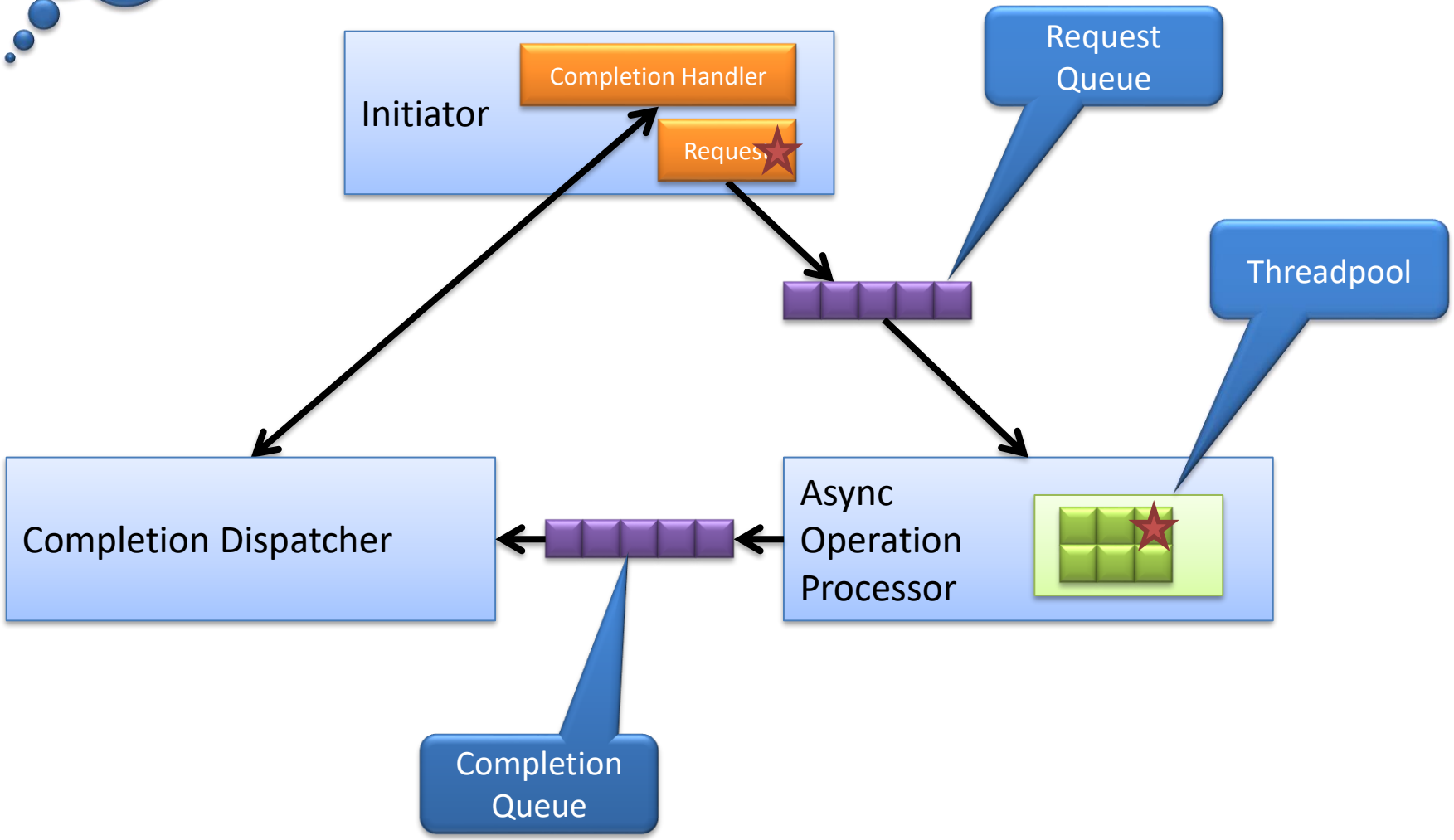
Pattern: Proactor Pattern

- Proactor:
  - Wie Reactor nur für lang laufende asynchrone Operationen
  - Completion Handler wird am Ende der Operation aufgerufen



Quelle: Wikipedia - http://de.wikipedia.org/wiki/Proactor

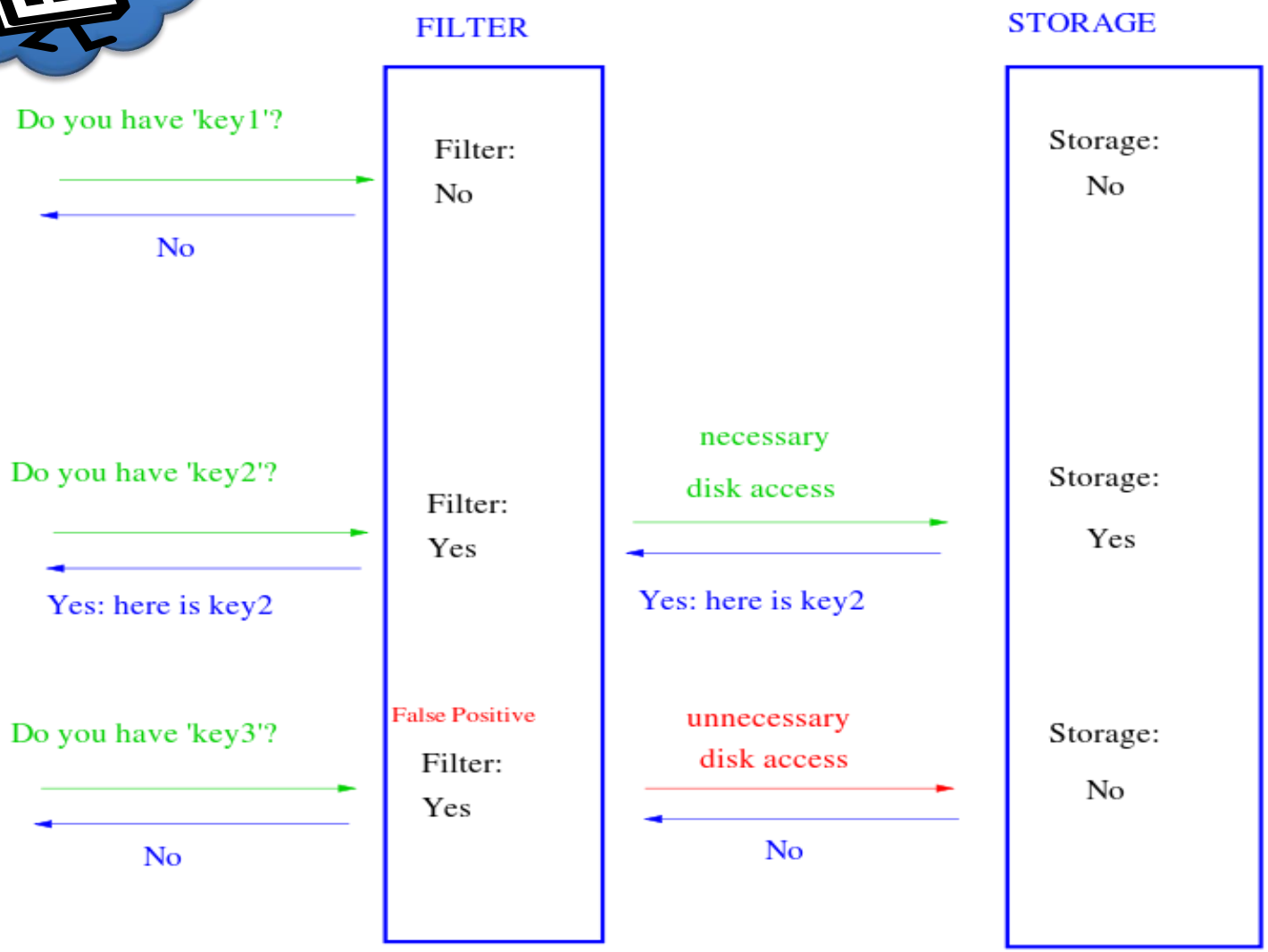
Pattern: Proactor



Bloom Filter

- Suche nach einem Schlüssel in:
  - Unsortierte Liste:
    - Schlüssel ist vorhanden – O(n/2)
    - Schlüssel ist nicht vorhanden – O(n)
  - Sortierte Strukturen, Bäume, etc. - O(log n)
  - Hash (ohne Kollisionen) O(1)
- Problem – Skalierung:
  - Hauptspeicher ist schnell aber endlich
  - HD groß aber langsam
- BloomFilter:
  - Schon 1970 von Burton H. Bloom für eine Rechtschreibprüfung
  - Verwendet (k) Hash Funktionen auf Teile des Schlüssels – O(k)
  - Akzeptiert eine Wahrscheinlichkeit für False Positives
  - Abhängig von der Wahrscheinlichkeit ist die Datenstruktur deutlich kleiner als eine Hashtabelle
  - Wird z.B.: verwendet in: Squid, Cassandra, Hadoop, etc

Bloom Filter



Quelle: Wikipedia - http://en.wikipedia.org/wiki/Bloom_filter

Bloom Filter
(besonders einfaches) Beispiel

Hashfunktion:
h(n) = n mod m; m = 7

n	n	n mod 7	0	1	2	3	4	5	6
e	5	5	0	0	0	0	0	1	0
i	9	2	0	0	1	0	0	0	0
s	19	5	0	0	0	0	0	1	0
∑			0	0	1	0	0	1	0

Test:

- sobald eine Position im Bitarray der k hashes 0 ist => Schlüssel nicht vorhanden
- Wenn alle Bits im Ergebnis 1 sind ist die Wahrscheinlichkeit p=Füllgrad ^ Zahl der Hashes dass der Schlüssel nicht vorhanden ist

Löschen nicht möglich (Lösung: counting bloom filter)

Herausforderungen

- Verteilung von Code
- Verteilung von Daten
- **Zugriff auf Daten & Code**
  - **Ausflug: Blockchains**

Blockchain: Motivation

- Wir wissen: (Daten) Konsistenz in verteilten Systemen ist problematisch (CAP)
- Die Lösungen basieren auf Zeitstempel, Mehrheitsentscheide, Tokensysteme, Transaktionsmanager, etc.
- Was wenn:
  - Knoten im System sich „unberechenbar“ verhalten
  - Ich nicht alle Knoten kenne/kontrolliere
  - Es auch unkooperative (bösartige) Knoten gibt

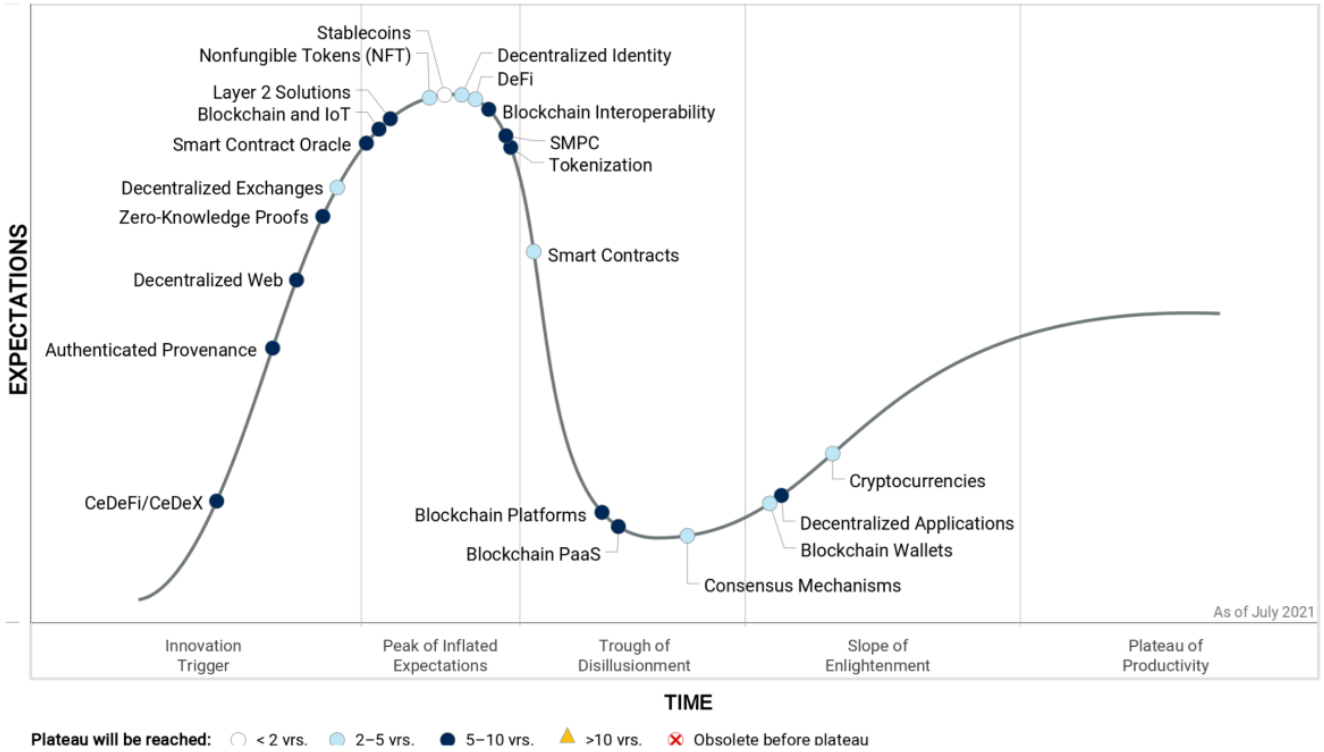
Blockchain: Motivation Beispiel

- Beweis von „Besitz“ / Transfer von „Besitz“:

Zutaten	Klassich (Bsp.)	Blockchain
Identitäten	Ausweis	Kryptografischer Schlüssel
Verwalter / Registrar	Banken / Notare	öffentliche, verteilte Liste aller Transaktionen (Distributed Ledger)
Regeln	Gesetze	Code
(Viel) Vertrauen	Stabilität (System)	?

Blockchain: Hype

Hype Cycle for Blockchain, 2021



Source: Gartner (July 2021)

747513

Blockchain Motivation Beispiel

- Beweis von “Besitz”
  => Kryptografisch - Private Key & Signatur & Verkettung
  => Kann jeder im Besitz der Blockchain validieren

- Transfer von „Besitz“
  => Benötigt Konsens in einem verteilten, dezentralen Netzwerk

Die Teilnehmer sind
- Nicht bekannt
- Haben eine unklare Verfügbarkeit
- Nicht per se Vertrauenswürdig

Blockchain
Transfer von Besitz

- Identität = Adresse => z.B. Public Key
- Transaktion von Quelle => Ziel, Wert, signiert mit private Key
- „Miner“ (parallel, verteilt) validieren die Transaktion und nehmen diese in einen neuen Block auf
- Block wird in die Chain „eingehängt“ und verteilt

BlockCHAIN (vereinfacht) Validierung

Header
- Hash (Transaktionen)
- Timestamp
- PoW / Nonce
- Hash Header-B(N – 1)

Body
#1 Transaction
#2 Transaction
#3 Transaction
#4 Transaction
#5 Transaction
#6 Transaction
#... Transaction

Header
- Hash (Transaktionen)
- Timestamp
- PoW / Nonce
- Hash Header-B(N – 1)

Body
#n+1 Transaction
#n+2 Transaction
#n+3 Transaction
#n+4 Transaction
#n+5 Transaction
#n+6 Transaction
#n+.. Transaction

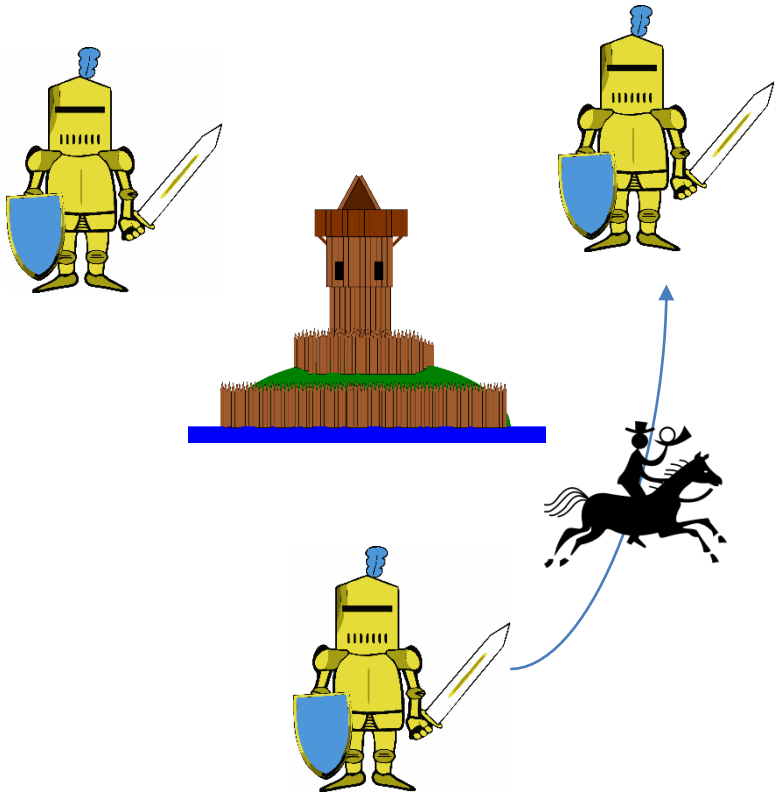
Header
- Hash (Transaktionen)
- Timestamp
- PoW / Nonce
- Hash Header-B(N – 1)

Body
#n+... Transaction
#n+... Transaction
#n+... Transaction
#n+... Transaction
#n+... Transaction
#n+... Transaction
#n+... Transaction

Und was haben Byzantinische Generäle damit zu tun?

- Müssen Konsens über Zeitpunkt eines Angriffs erreichen
- Nur ein gemeinsamer Angriff kann erfolgreich sein (alle Loyalen Generäle kommen zur selben Entscheidung)
- Nachrichten können nur per Boten ausgetauscht werden
- Die Kommunikation kann fehlerhaft oder kompromittiert, Verräter am Werk sein, etc.

- Lösungen (Protokolle) oft für n < 1/3 (n = Fehler, „Verräter“) mit verschiedenen Randbedingungen
- Einsatz in Safetyszenarien (e.g. Flightcontrol)
- Byzantinischer Fehler, Byzantinische Fehlertolleranz



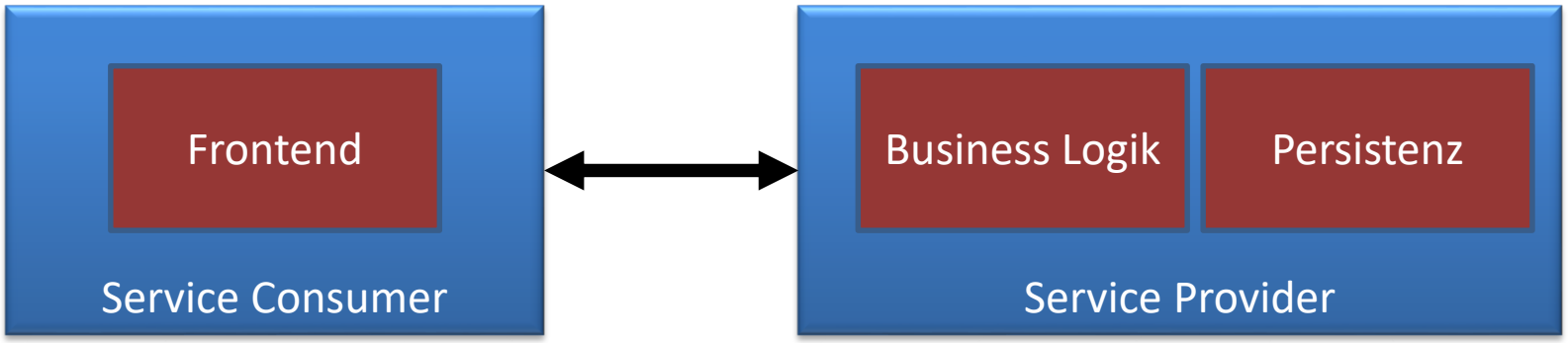
Proof of Work / Konsens in der Blockchain

- Löst das Problem der Byzantinischen Generäle für n (“Verräter“) < 50%
- „Motivation“ über Kosten und erwarteten Gewinn (Spieletheorie)
- Proof of Work über Krypto „Challenge“ => Kosten
- Belohnung der Miner nur dann wenn die Krypto Challenge gelöst wurde und der Block „überlebt“ (dauerhaft in die Blockchain aufgenommen wird, Bestandteil der „längsten Kette“ ist)

Herausforderungen

- Verteilung von Code
- Verteilung von Daten
- **Zugriff auf Daten & Code**
  - **Ausflug: Web Services**

SOA – Service Oriented Architecture



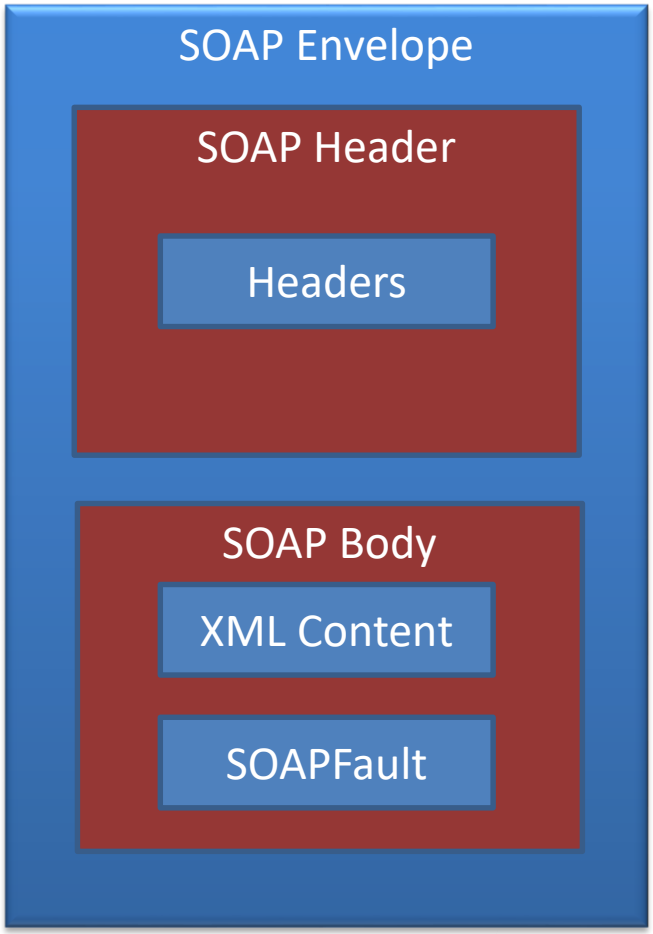
- Div. Design Prinzipien
  - Lose Kopplung
  - Abstraktion
  - Wiederverwendbarkeit
  - Zustandslosigkeit
  - …

≠ Web Services

SOAP

- Entstanden 1999
- XML basiert
- HTTP typisch als Transport, aber nicht erforderlich
- Struktur: Envelope, Header, Body
- WSDL als Beschreibungssprache
- (tot: UDDI)

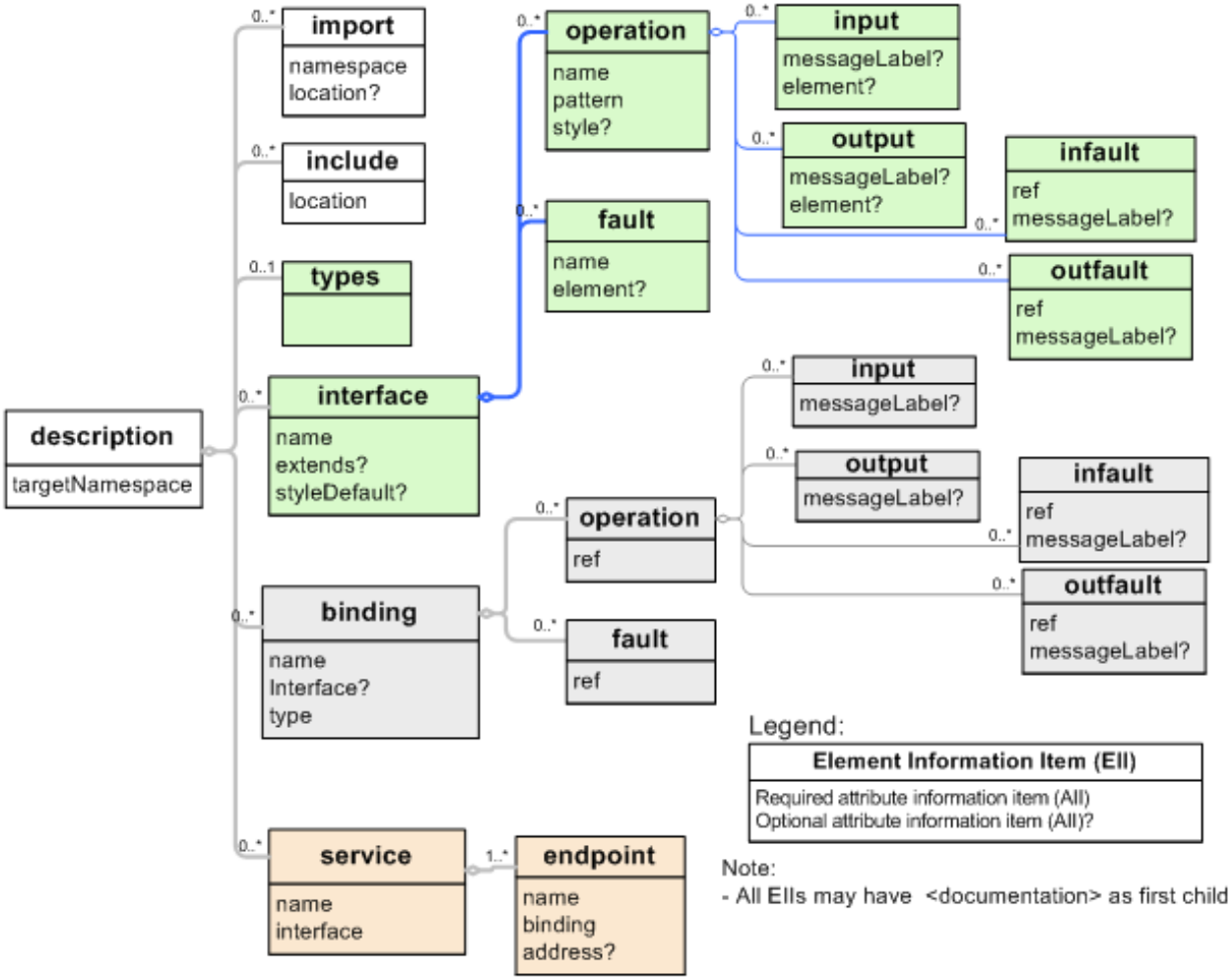
SOAP Nachrichtenaufbau




```
<?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.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

```
<?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.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

WSDL



Quelle: W3C

Beispiel: SoapUI

SoapUI 4.6.4

File Tools Desktop Help

Search Forum

Navigator

Projects
PaymentService
PaymentServiceSOAP
AddAdditionalPaymentDetailsToCustomer
Request 1
AddStandardPaymentDetailsToCustomer
AddWebCentAccountToCustomer
AssignPaymentDetailsToContract
AssignPaymentDetailsToOrder

Request Properties

Property	Value
Name	Request 1
Description	
Message Size	565
Encoding	UTF-8

Properties

Request 1

https://ac1www.private.ipayment.de/intern/service/payment/1.1/

XML Raw

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org
   <soapenv:Header/>
   <soapenv:Body>
      <pay:AddAdditionalPaymentDetailsToCustomerRequest>
         <customerId>?</customerId>
         <marketId>?</marketId>
         <ipaymentId>?</ipaymentId>
         <globalsPaymentTypeId>?</globalsPaymentTypeId>
         <!--Optional:-->
         <retryOnFailure>?</retryOnFailure>
      </pay:AddAdditionalPaymentDetailsToCustomerRequest>
   </soapenv:Body>
```

Au... Headers ... Attachments... WS... WS-... JMS Head... JMS Properties...

1:1

SoapUI log http log jetty log error log wsrm log memory log

REST

(REpresentational State Transfer)

- SOA mit REST -> ROA
- Erdacht 2000 von Roy Fielding als Doktorarbeit
- Stark an HTTP gebunden, URL adressierbar
- Response Encodings: XML, JSON, HTML, andere denkbar
- Zustandslos, CRUD

HTTP Methoden bei REST

Primitive	Bedeutung
GET	Ressource lesen, ohne Status zu verändern
POST	Ressource anlegen und sonstige Operationen
PUT	Ressource anlegen/ändern
PATCH	Teilweise Änderung einer Ressource
DELETE	Ressource löschen
HEAD	Metadaten zu einer Ressource erfragen
OPTIONS	Methoden zum Ressourcenzugriff abfragen

REST Beispiel

HttpPath	HttpMethod	TargetClass	TargetMethod	isDeprecated
/extern/PSS	POST	com.unitedinternet.demail.ident.backend.external.bss.PssResource	allocateResourceJson	false
/extern/PSS/Datatype	GET	com.unitedinternet.demail.ident.backend.external.bss.PssResource	retrieveDescriptionDatatypesHtml	false
/extern/PSS/Datatype /{typename}	GET	com.unitedinternet.demail.ident.backend.external.bss.PssResource	retrieveDescriptionSingleDatatypeHtml	false
/extern/PSS/Datatype /{typename}	GET	com.unitedinternet.demail.ident.backend.external.bss.PssResource	retrieveDescriptionSingleDatatypePlain	false
/extern /PSS/SystemConfiguration	GET	com.unitedinternet.demail.ident.backend.external.bss.PssResource	retrieveSupportedSettingsInfoHtml	false
/extern /PSS/SystemConfiguration /{settingname}	GET	com.unitedinternet.demail.ident.backend.external.bss.PssResource	retrieveDescriptionSingleSettingnameHtml	false
/extern /PSS/{provisioningId}	DELETE	com.unitedinternet.demail.ident.backend.external.bss.PssResource	cancel	false
/extern /PSS/{provisioningId} /Connection	GET	com.unitedinternet.demail.ident.backend.external.bss.PssResource	getResourceConnectionsHtml	false
/extern /PSS/{provisioningId} /Connection /{otherComponentType} /{otherProvisioningId}	DELETE	com.unitedinternet.demail.ident.backend.external.bss.PssResource	disconnectResource	false
/extern /PSS/{provisioningId} /Connection /{otherComponentType} /{otherProvisioningId}	PUT	com.unitedinternet.demail.ident.backend.external.bss.PssResource	connectResource	false
/extern /PSS/{provisioningId} /Creation	POST	com.unitedinternet.demail.ident.backend.external.bss.PssResource	createResourceLegacy	true

REST Encoding

- JSON, XML, YAML, HTML oder beliebig

```
{"id":23,"nachName":"Magschok","vorName":"Georg","alter":23}
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person id=„23">
        <alter>23</alter>
        <vorName>Georg</vorName>
        <nachName>Magschok</nachName>
</person>
```

- Chunking
- Compression
- Multipart

Beispiel: Enunciate

CUSTOMER

REST Data Model

Home > Data Model > ns0 > customer element

customer element

Type:	customer
Namespace:	(default namespace)
XML Schema:	ns0.xsd

this element contains customer information for the Ident support API including HATEOAS links

Example XML

```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
  <ident_lock>...</ident_lock>
  <ident_lock_at>...</ident_lock_at>
  <links>
    <href>...</href>
    <method>...</method>
    <rel>...</rel>
  </links>
  <links>
    <!--...-->
  </links>
```

Example JSON

```
{
  "ident_lock" : false,
  "ident_lock_at" : ...,
  "links" : [ {
    "href" : "...",
    "method" : "GET",
    "rel" : "list-customer-ident-processes"
  }, ... ]
}
```

REST in Java: JAX-RS

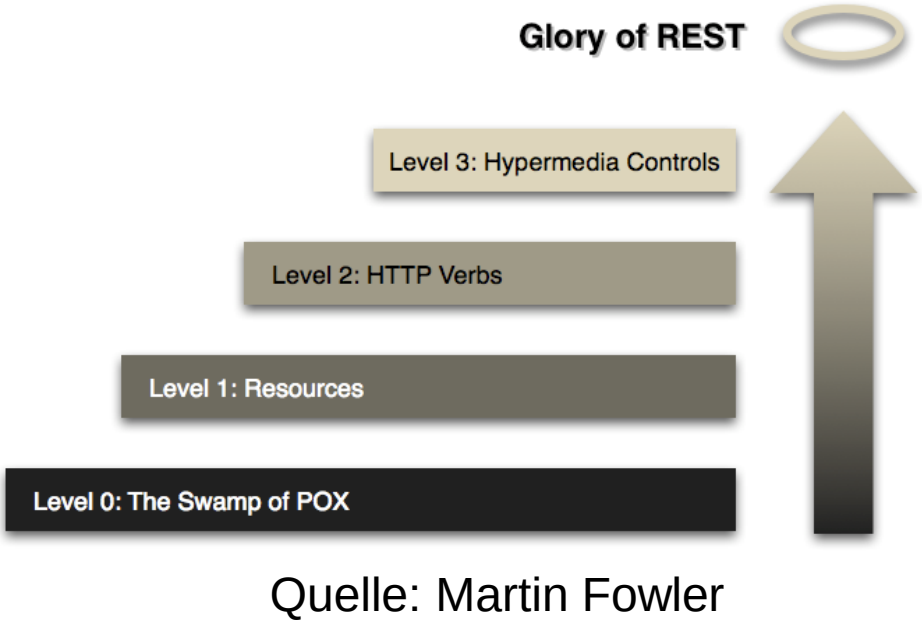
```
@Path("/greeting")
@Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
@Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })

public class GreetingService {
 @GET public Response message() {
         return new Response("Hi REST!");
 }
 @POST public Response lowerCase(final Request message) {
         return new Response(message.getValue().toLowerCase());
 }
}
```

REST: WADL

```
<?xml version="1.0" encoding="UTF-8"?>
<wadl:application xmlns:wadl="http://wadl.dev.java.net/2009/02" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <wadl:doc xmlns:enunciate="http://enunciate.codehaus.org/" enunciate:generatedBy="Enunciate-1.26"/>
  <wadl:grammars>
    <wadl:include href="ns0.xsd"/>
  </wadl:grammars>
  <wadl:resources base="http://localhost:8080/de-mail-entities-rest">
    <wadl:resource path="/rest/idents/{customerId}">
      <wadl:param name="customerId" style="template">
        <wadl:doc>
          <![CDATA[customer id]]>
        </wadl:doc>
      </wadl:param>
      <wadl:method name="GET">
        <wadl:doc>
          <![CDATA[get the customer including HATEOAS links]]>
        </wadl:doc>
        <wadl:request/>
        <wadl:response>
          <wadl:doc>
            <![CDATA[CustomerWithAdditionalInformation with HATEOAS links]]>
          </wadl:doc>
          <wadl:representation mediaType="application/json"/>
        </wadl:response>
      </wadl:method>
      <wadl:method name="DELETE">
        <wadl:doc>
          <![CDATA[delete customer by customer id]]>
        </wadl:doc>
```

HATEOAS & RMM

- Rest Maturity Model (Leonard Richardson)



Quelle: Martin Fowler

- HATEOAS (Hypermedia as the Engine of Application State): Dynamische Ressource Links, weg von der statischen Interfacedefinition

Andere Mechanismen zur verteilten Kommunikation

- RPC
- DCOM
- CORBA
- RMI
- XML-RPC
- DCE
- RMI-IIOP
- Thrift
- Protocol Buffers
- GraphQL
- …

Out of Scope dieser Vorlesung