QAWARE SOFTWARE ENGINEERING

Cloud Computing

Kapitel 8: Cloud-fähige Softwarearchitekturen

Dr. Josef Adersberger

Softwarearchitektur =
(Komponenten + Schnittstellen + [De-]Komposition) x Sichten

Zweck:
- **Beherrschbarkeit & Parallelisierbarkeit**: Divide & Conquer
- **Isolation**: Problem- / Complexity-Hiding

➔ Die Softwarearchitektur gilt entlang aller Phasen.

Entwurf

Ausleihe

Programmierung

Modul: Ausleihe
`pom.xml`
`de.qaware.bib.ausleihe.IAusleihe`
`de.qaware.bib.ausleihe.impl.Ausleihe`

Betrieb

?

De-Kompositionseinheiten
- Datenhoheit
- Hohe Kohäsion
- Lose Kopplung

- Entwicklungseinheiten
- Planungseinheiten

- Release-Einheiten
- Deployment-Einheiten
- Laufzeit-Einheiten (Risikogemeinschaften, Einheiten der Isolation)
- Skalierungs-Einheiten

Regel 1 für den Betrieb in der Cloud.

"Everything fails all the time."

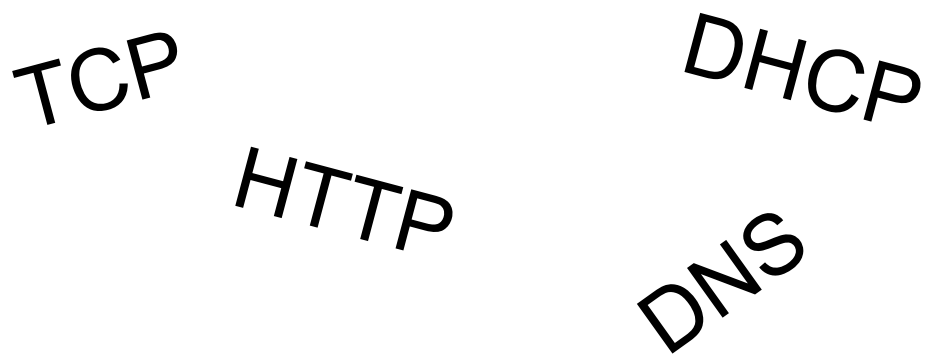
— Werner Vogels, CTO of Amazon

Regel 2 für den Betrieb in der Cloud.

Soll nur der Himmel die Grenze sein, dann funktioniert nur horizontale Skalierung.

Regel 3 für den Betrieb in der Cloud.

Wer in die Cloud will, der sollte Cloud sprechen.



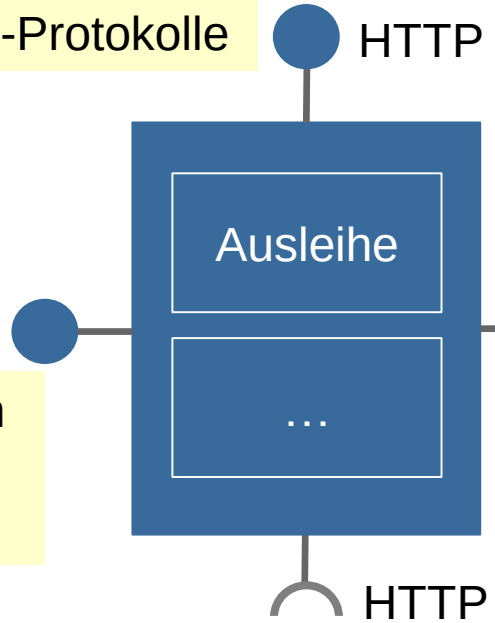
Regel 4 für den Betrieb in der Cloud: Mache es Dir so leicht wie möglich.

Quelle: http://www.vocativ.com/culture/society/the-hipster-effect

Betriebskomponenten, die für die Cloud ausgelegt sind, werden aktuell oft als Micro Services bezeichnet.

- Betriebseinheiten sollten auch der fachlichen De-Komposition folgen (Service-Orientierung)
- Die Betriebseinheiten sind:
  - Release-Einheiten
  - Deployment-Einheiten
  - Laufzeit-Einheiten
  - Skalierungs-Einheiten

Kommunikation über Internet-Protokolle



Diagnosestecker

Besitzt Diagnoseschnittstellen um ein fehlerhaftes Verhalten erkennen zu können

main()

Eigenständig und isoliert lauffähig

Cloud-Architektur aus Sicht der Softwarearchitektur: Design for Failure.

1. Jede Komponente läuft eigenständig und isoliert ➔ *Micro Service*
2. Die Micro Services kommunizieren untereinander über Internet-Protokolle ➔ *Micro Service System*
3. Jeder Micro Service kann in mehreren Instanzen laufen und bietet damit Redundanz. Es gibt keinen „Common Point of Failure“.
4. Jeder Micro Service besitzt Diagnoseschnittstellen um ein fehlerhaftes Verhalten erkennen zu können
5. Jeder Micro Service kann zu jeder Zeit neu gestartet und auf einem anderen Knoten in Betrieb genommen werden. Er besitzt keinen eigenen Zustand.
6. Die Implementierung hinter einem jeden Micro Service kann ausgetauscht werden, ohne dass die Nutzer davon etwas bemerken.

Die De-Kompositionsebene der „Micro“ Services muss klug gewählt werden.

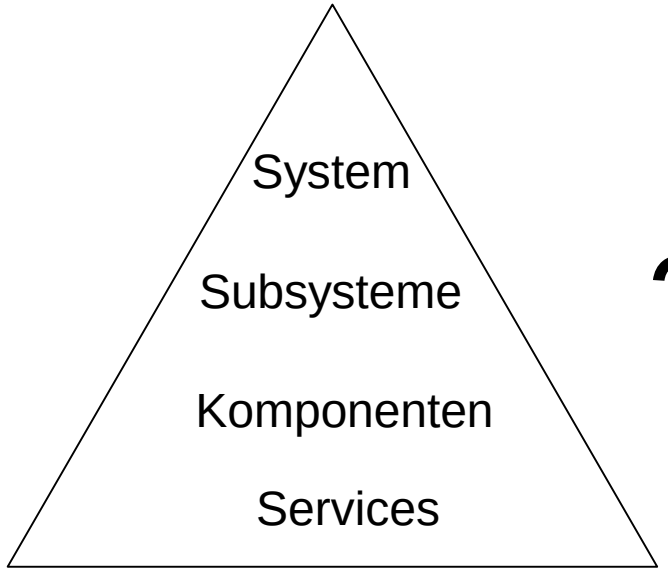
- Der Tradeoff:

+

- Flexible Skalierungsmöglichkeiten
- Isolation zur Laufzeit
- Komponentenorientierung bis in den Betrieb
- Unabhängige Release und Deployments möglich

-

- Komplexere Betriebsumgebung
- „Verteilungs-Schulden“
- Schwierigere Diagnostizierbarkeit
- Komplexere Skalierbarkeit
- Komplexere Integration



?

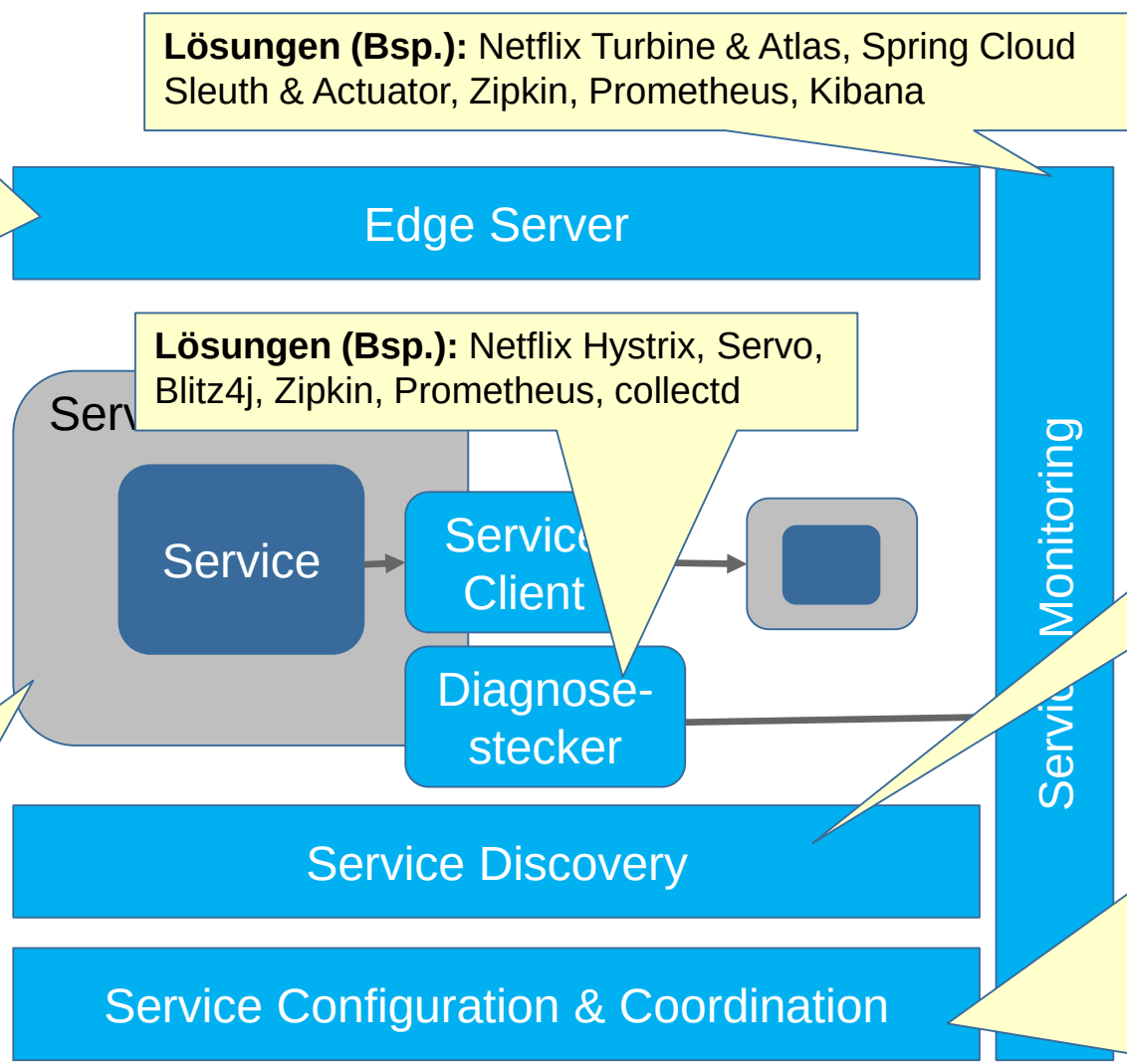
Die Technische Infrastruktur rund um die „Micro Services“.

Typische Aufgaben:
- Authentifizierung
- Load Shedding
- Load Balancing
- Failover
- Rate Limiting
- Request Monitoring
- Request Validierung
- Caching
- Logging

Lösungen (Bsp.):
- Netflix Zuul
- Vulcan.d
- Kong

Typische Aufgaben:
- HTTP Handling
- Konfiguration
- Diagnoseschnittstelle
- Lebenszyklus steuern
- APIs bereitstellen

Lösungen (Bsp.):
- Spring Boot
- Netflix Karyon
- Dropwizard



Typische Aufgaben:
- Service Registration
- Service Lookup
- Service Description
- Membership Detection
- Failure Detection

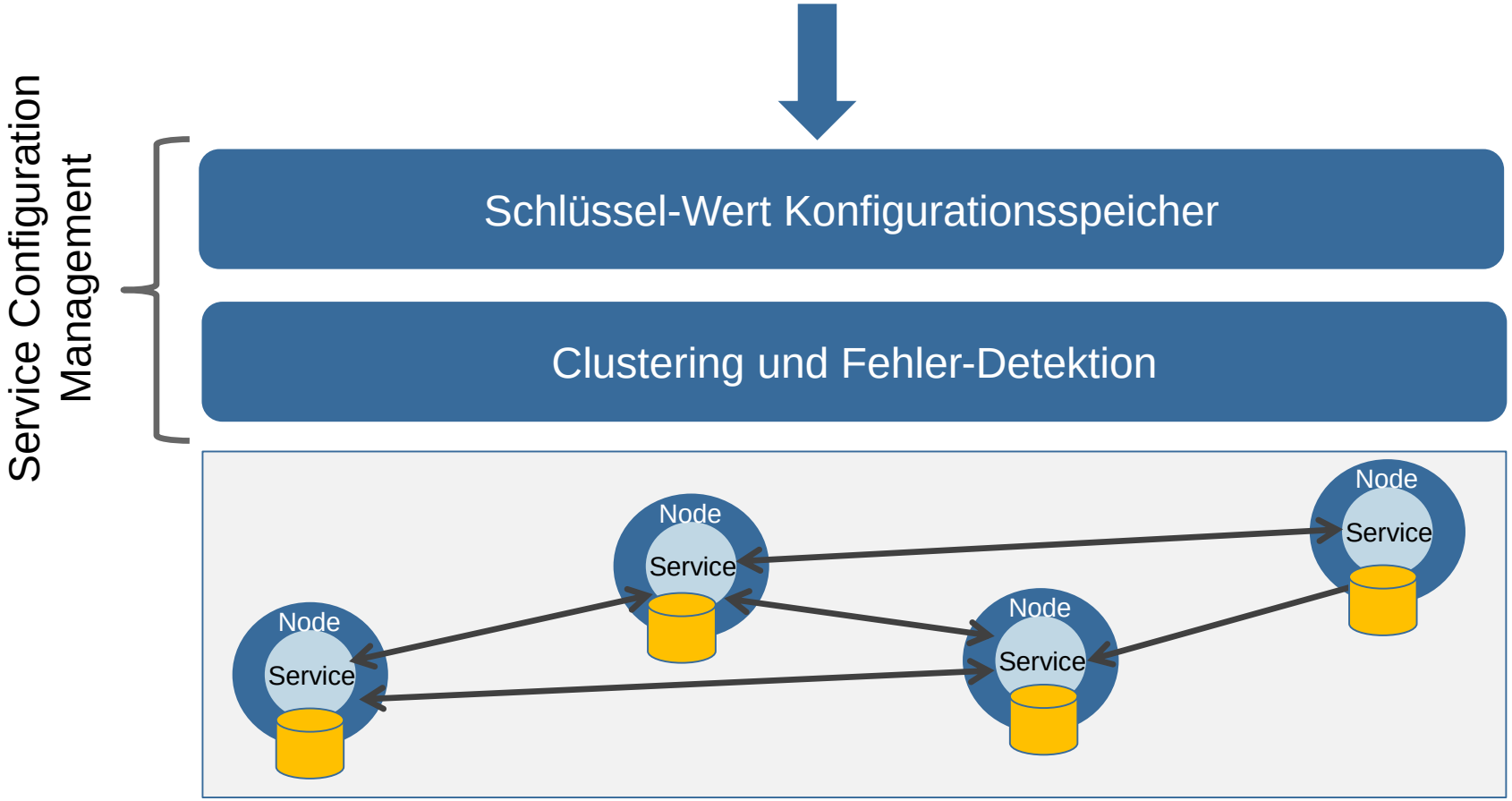
Lösungen (Bsp.):
- SkyDNS
- Netflix Eureka
- Consul
- Hyperbahn

Typische Aufgaben:
- Key-Value-Store (oft in Baumstruktur. Teilw. mit Ephemeral Nodes)
- Sync von Konfigurationsdateien
- Watches, Notifications, Hooks, Events
- Koordination mit Locks, Leader Election und Messaging
- Konsens im Cluster herstellen

Lösungen (Bsp.):
- Zookeeper
- etcd
- Consul
- Spring Cloud Config / Bus / Cluster

Service Configuration: Verteilter Zustand

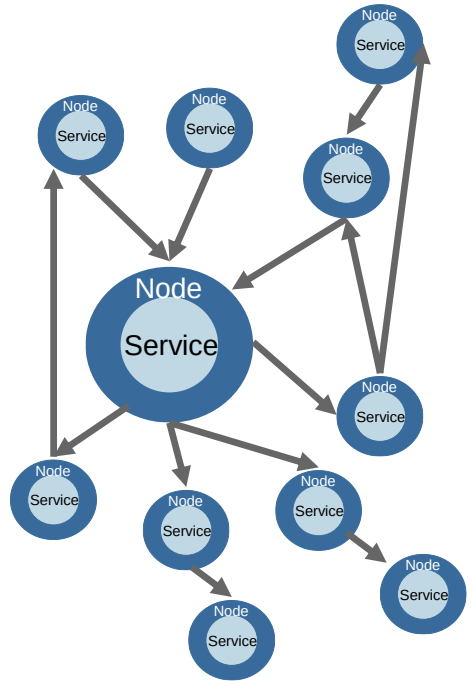
Ein verteilter Konfigurationsspeicher.



Wie wird der Zustand des Konfigurationsspeichers im Cluster synchronisiert?

Gossip Protokolle: Inspiriert von der Verbreitung von Tratsch in sozialen Netzwerken.

- Grundlage: Ein Netzwerk an Agenten mit eigenem Zustand
- Agenten verteilen einen Gossip-Strom
  - Nachricht: Quelle, Inhalt / Zustand, Zeitstempel
  - Nachrichten werden in einem festen Takt periodisch versendet an eine bestimmte Anzahl anderer Knoten (Fanout)
- Virale Verbreitung des Gossip-Stroms
  - Knoten, die mit mir in einer Gruppe sind, bekommen auf jeden Fall eine Nachricht
  - Die Top x% an Knoten, die mir Nachrichten schicken bekommen eine Nachricht
- Nachrichten, denen vertraut wird, werden in den lokalen Zustand übernommen
  - Die gleiche Nachricht wurde von mehreren Seiten gehört
  - Die Nachricht stammt von Knoten, denen der Agent vertraut
  - Es ist keine aktuellere Nachricht mit gleichem Inhalt vorhanden



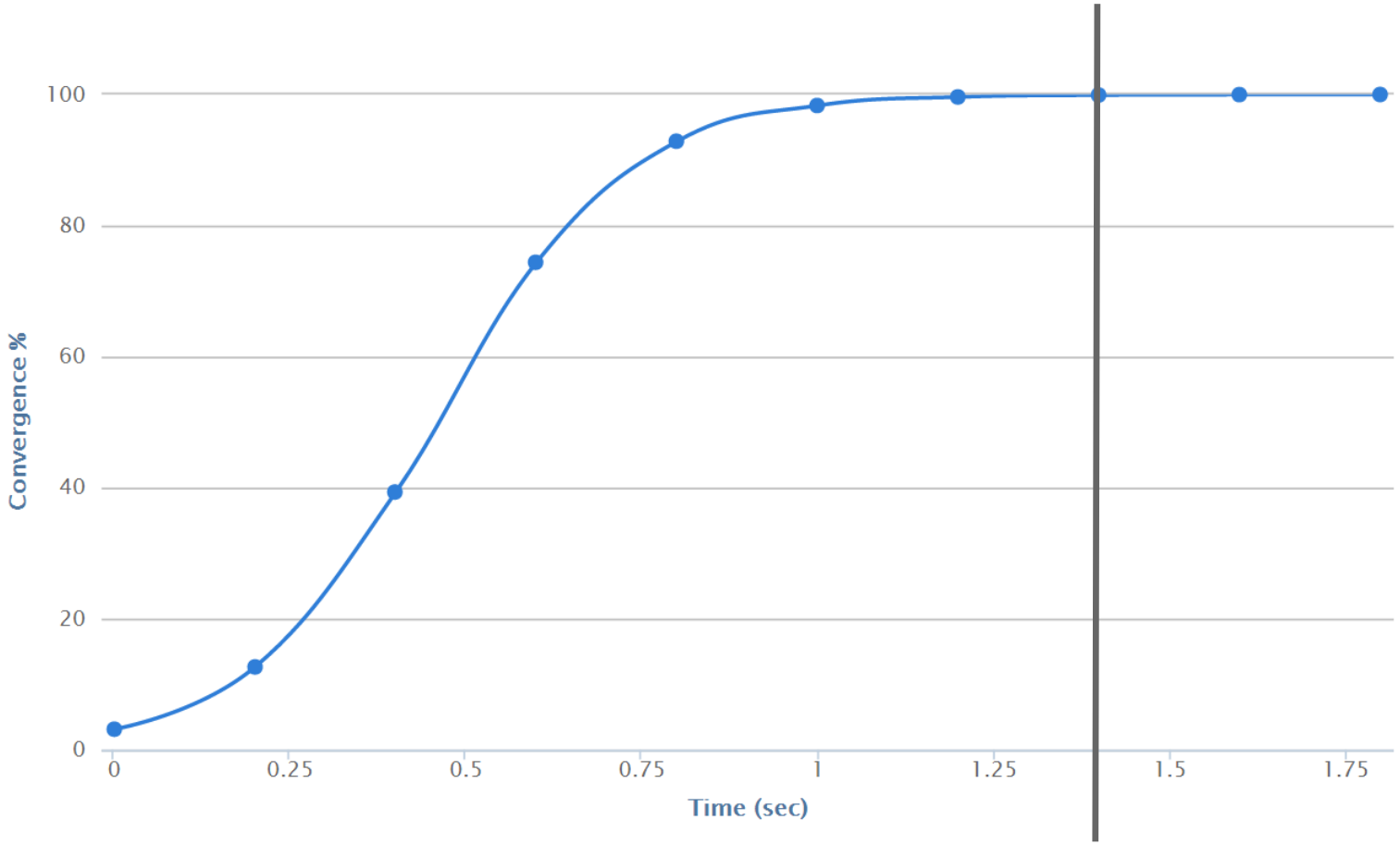
Vorteile:

- Keine zentralen Einheiten notwendig.
- Fehlerhafte Partitionen im Netzwerk werden umschifft. Die Kommunikation muss nicht verlässlich sein.

Nachteile:

- Der Zustand ist potenziell inkonsistent verteilt (konvergiert aber mit der Zeit)
- Overhead durch redundante Nachrichten.

Die Konvergenz der Daten und damit der Zeitpunkt der vollständigen Konsistenz ist berechenbar.



Vollständige Konsistenz

Estimated max bandwidth: 328.1 kbps/node

Gossip Interval

The gossip interval controls how often messages are gossiped to other nodes

0.2 seconds

Gossip Fanout

The gossip fanout controls how many nodes we gossip with

3 nodes

Nodes

This controls how many simulated nodes are in the cluster

30

Packet Loss

This controls the amount of simulated packet loss [0, 100)

0 % lost packets

Node failures

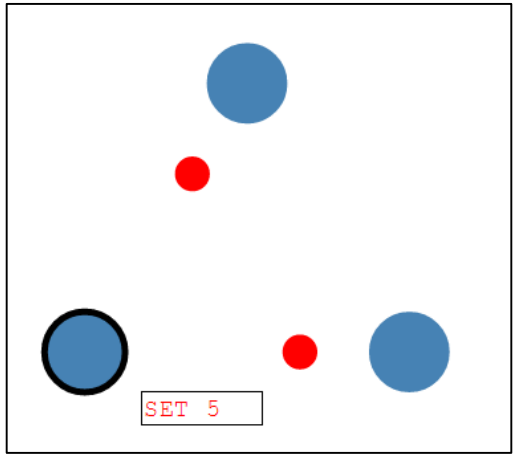
This controls what percent of simulated nodes are failed

0 % failed

https://www.serfdom.io/docs/internals/simulator.html

Gossip Protokolle sind nicht streng konsistent. Ist dies wichtig, so helfen Protokolle für verteilten Konsens.

- Grundlage: Netzwerk an Agenten
- Prinzip: Es reicht, wenn der Zustand auf einer einfachen Mehrheit der Knoten konsistent ist und die restlichen Knoten ihre Inkonsistenz erkennen.
- Verfahren:
  - Das Netzwerk einigt sich per einfacher Mehrheit auf einen Leader-Agenten – initial und falls der Leader-Agent nicht erreichbar ist.
  - Alle Änderungen laufen über den Leader-Agenten. Dieser verteilt per Multicast Änderungsnachrichten periodisch im festen Takt an alle weiteren Agenten.
  - Quittiert die einfache Mehrheit an Agenten die Änderungsnachricht, so wird die Änderung im Leader und (per Nachricht) auch in den Agenten aktiv, die quittiert haben.
- Konkrete Konsens-Protokolle: Raft, Paxos



Vorteile:

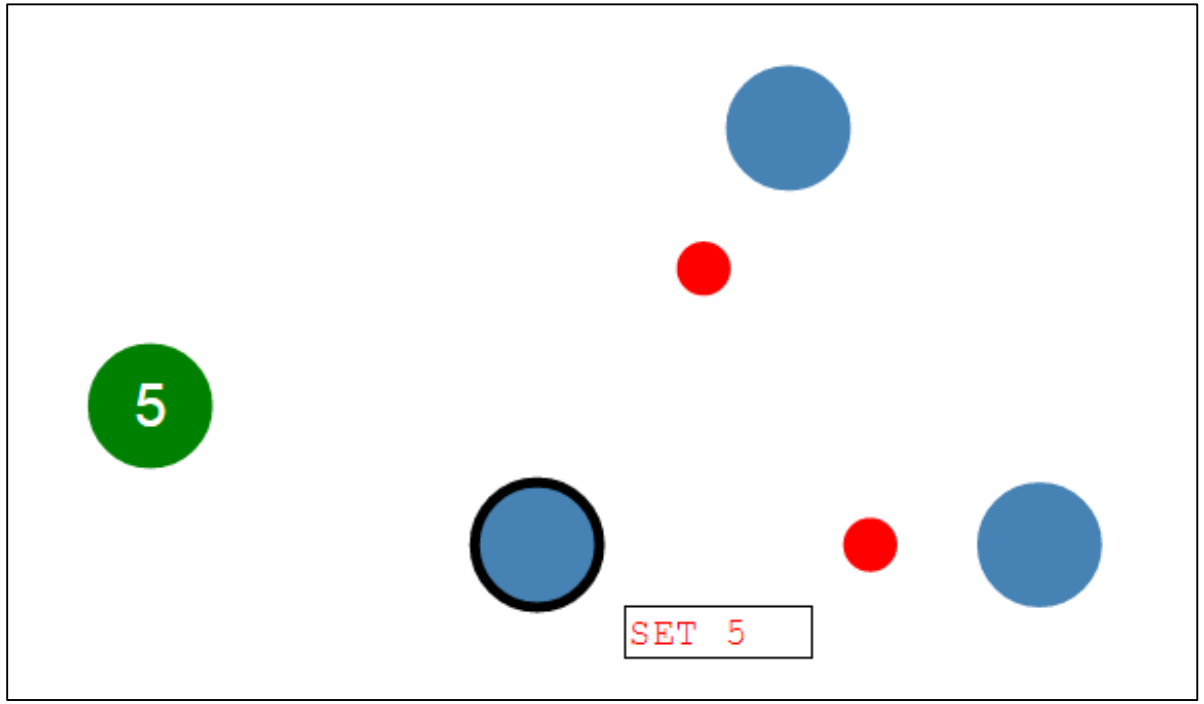
- Fehlerhafte Partitionen im Netzwerk werden toleriert und nach Behebung des Fehlers wieder automatisch konsistent.
- Streng konsistente Daten.

Nachteile:

- Der zentrale Leader-Agent limitiert den Durchsatz an Änderungen.
- Nicht hoch-verfügbar: Bei einer Netzwerk-Partition kann die kleinere Partition nicht weiterarbeiten.

Das Raft Konsens-Protokoll

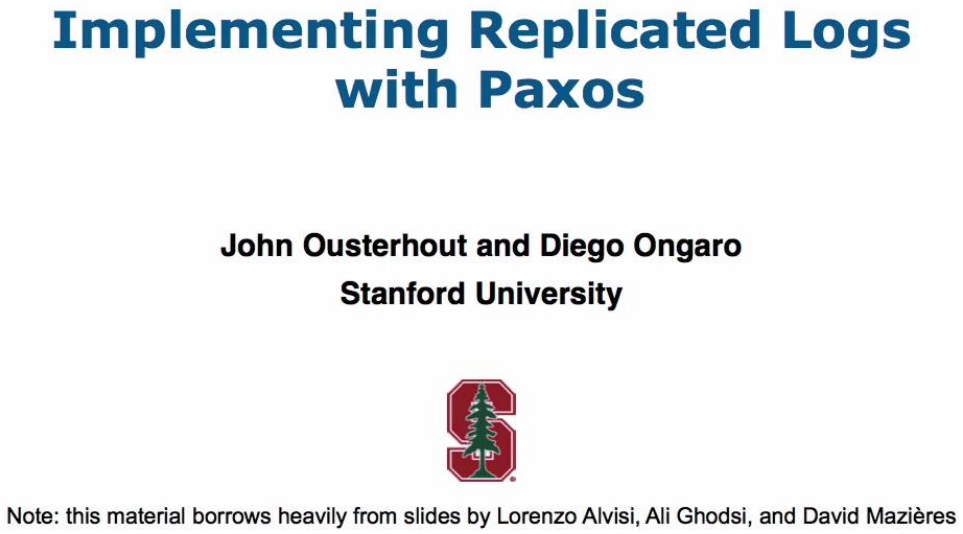
- Ongaro, Diego; Ousterhout, John (2013). "In Search of an Understandable Consensus Algorithm".



http://thesecretlivesofdata.com/raft

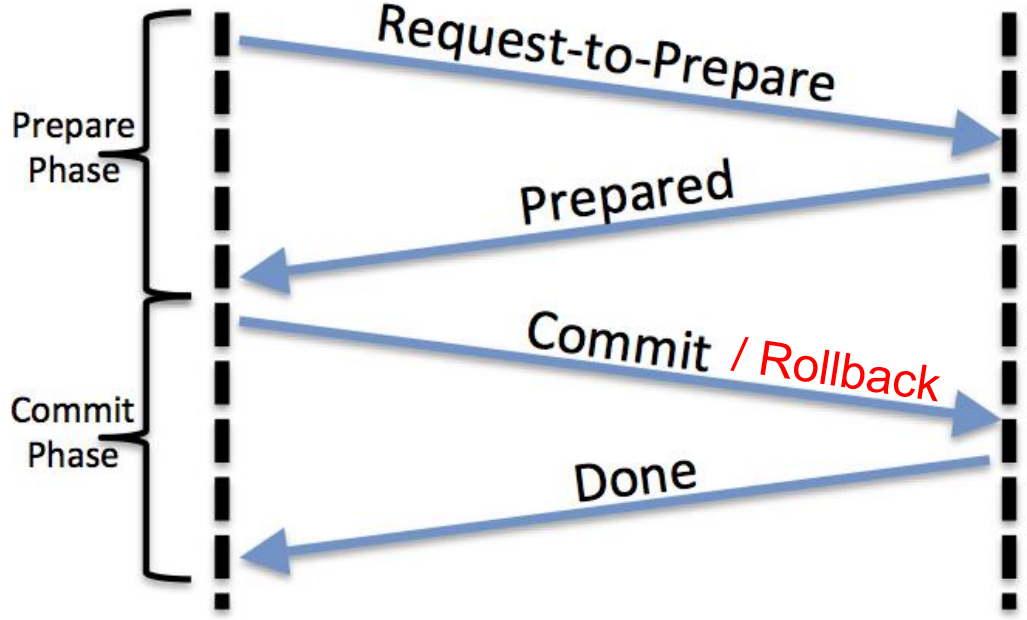
Paxos als Alternative zu Raft

- Erstmalig 1989 beschrieben von Leslie Lamport zur Replikation von Zustandsautomaten in verteilten Systemen auf Basis (s)einer logischen Uhr.
- Unterschied zu Raft: Komplexer.
  Aber funktional gleichwertig zu Raft.
- Quellen:
  - http://the-paper-trail.org/blog/consensus-protocols-paxos/
  - https://www.youtube.com/watch?v=JEpsBg0AO6o
  - http://www.uio.no/studier/emner/matnat/ifi/INF5040/h08/group/The_Paxos_Algorithm.pdf
  - http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf



Ist strenge Konsistenz über alle Knoten notwendig, so verbleibt das 2-Phase-Commit Protokoll (2PC)

- Ein Transaktionskoordinator verteilt die Änderungen und aktiviert diese erst bei Zustimmung aller. Ansonsten werden die Änderungen rückgängig gemacht.



Vorteil:

- Alle Knoten sind konsistent zueinander.

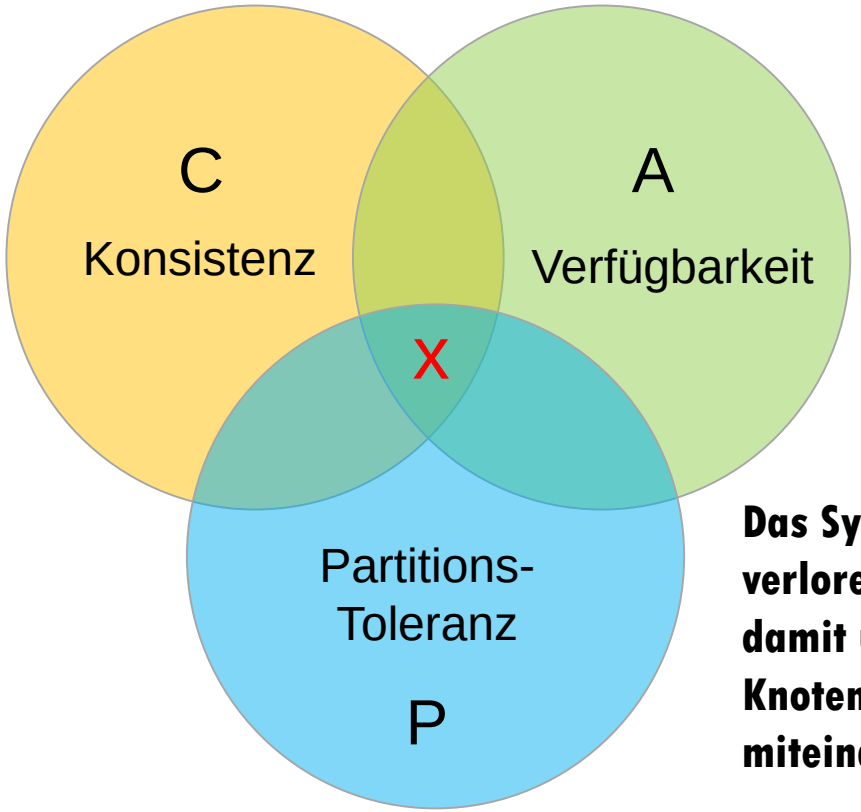
Nachteile:

- Zeitintensiv, da stets alle Knoten zustimmen müssen.
- Das System funktioniert nicht mehr, sobald das Netzwerk partitioniert ist.

Das CAP Theorem

- Theorem von Brewer für Eigenschaften von zustandsbehafteten verteilten Systemen – mittlerweile auch formal bewiesen.
  Brewer, Eric A. "Towards robust distributed systems." *PODC*. 2000.
- Es gibt drei wesentliche Eigenschaften, von denen ein verteiltes System nur zwei gleichzeitig haben kann:

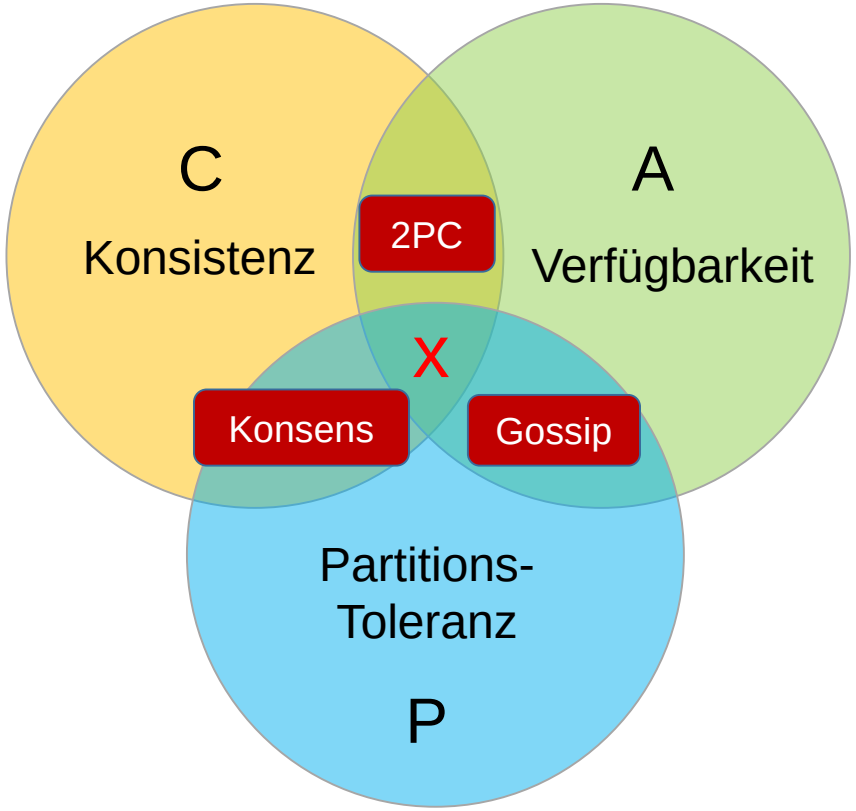
Alle Knoten sehen die selben Daten zur selben Zeit. Alle Kopien sind stets gleich.



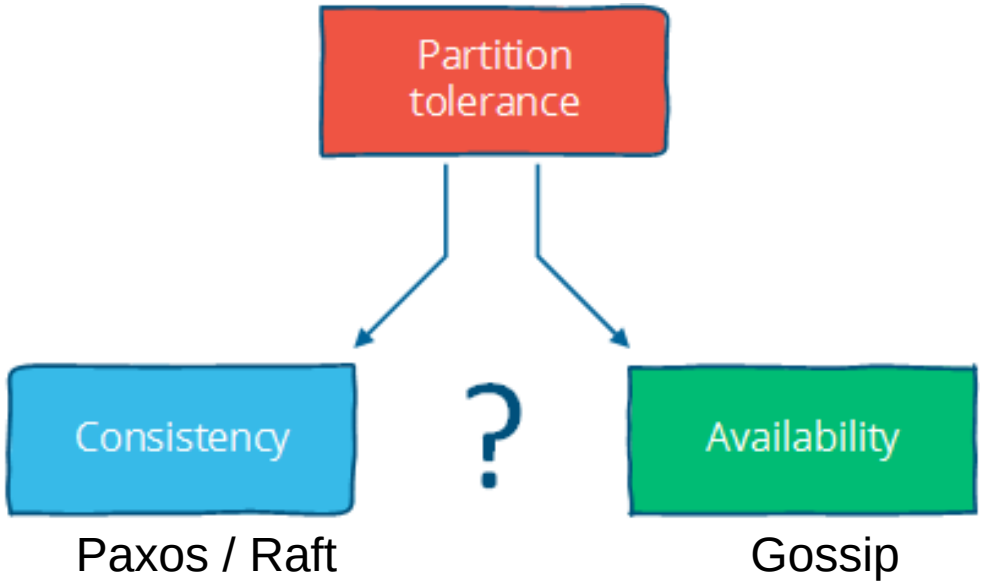
Das System läuft auch, wenn einzelne Knoten ausfallen. Ausfälle von Knoten und Kanälen halten die überlebenden Knoten nicht von ihrer Funktion ab.

Das System funktioniert auch im Fall von verlorenen Nachrichten. Das System kann dabei damit umgehen, dass sich das Netzwerk an Knoten in mehrere Partitionen aufteilt, die nicht miteinander kommunizieren.

Die vorgestellten Protokolle und das CAP Theorem.



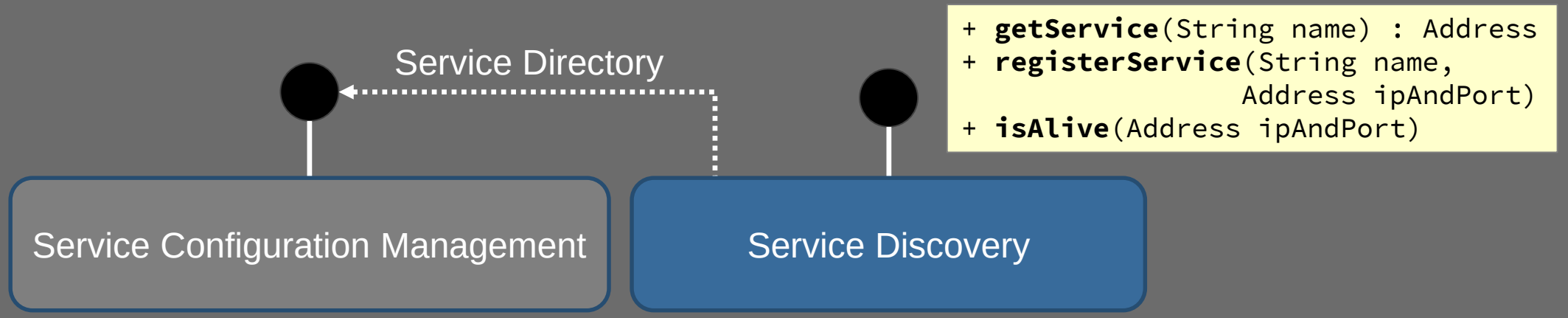
In der Cloud müssen Partitionen angenommen werden. Damit ist die Entscheidung binär zwischen Konsistenz und Verfügbarkeit.



Lösungen

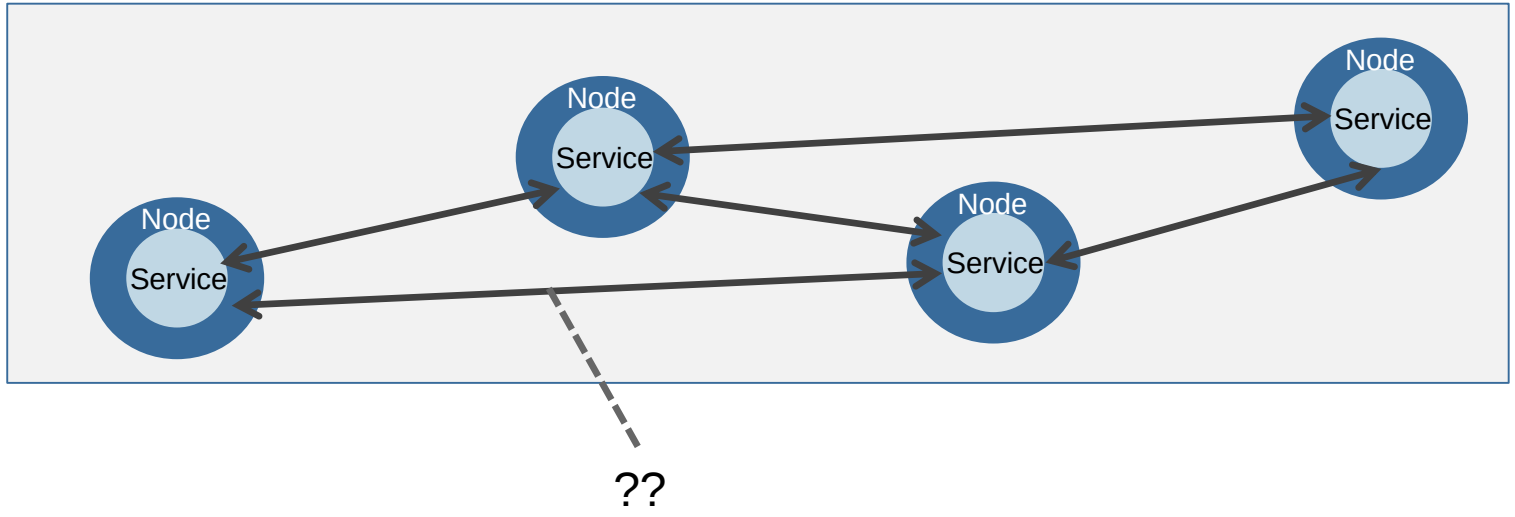
Komponente	Synchronisations-Konzepte	Lösungen
Service Configuration Management	Konsens	Raft-basiert: ▪ etcd (https://github.com/coreos/etcd) ▪ Consul (http://www.consul.io) Paxos-basiert: ▪ Doozerd (https://github.com/ha/doozerd) ▪ Zookeeper (https://zookeeper.apache.org)
	Gossip	▪ Serf (https://www.serfdom.io) ▪ Copycat (https://github.com/kuujo/copycat)
	2PC	▪ Klassische Datenbanken

Service Discovery



Flexibler Zugriff auf Services

Die Probleme einer klassischen Verknüpfung von Services in der Cloud.



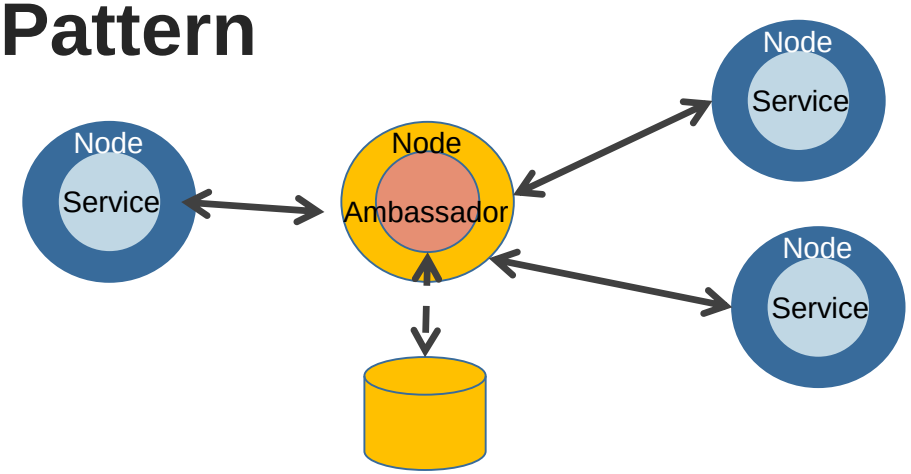
Probleme:

- Mangelnde Redundanz: Jeder Service wird direkt genutzt. Er kann nicht unmittelbar in mehreren Instanzen laufen, die Redundanz schaffen.
- Mangelnde Flexibilität: Die Services können nicht ohne Seiteneffekt neu gestartet oder auf einem anderen Knoten in Betrieb genommen werden – oder sogar durch eine andere Service-Implementierung ausgetauscht werden.

Lösungen:

- Dynamischer DNS
- Ambassaor
- Dynamischer Konfigurationsdateien und Umgebungsvariablen

Das Ambassador Pattern



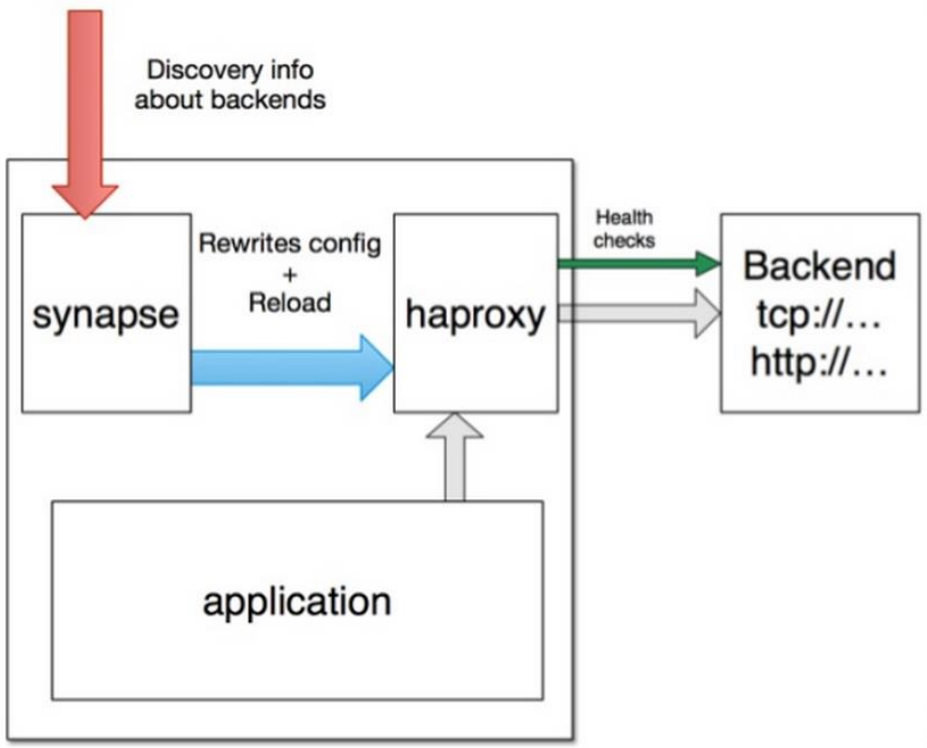
Ein Ambassador-Knoten für jede Knoten-Art (z.B. NGINX-Webserver)

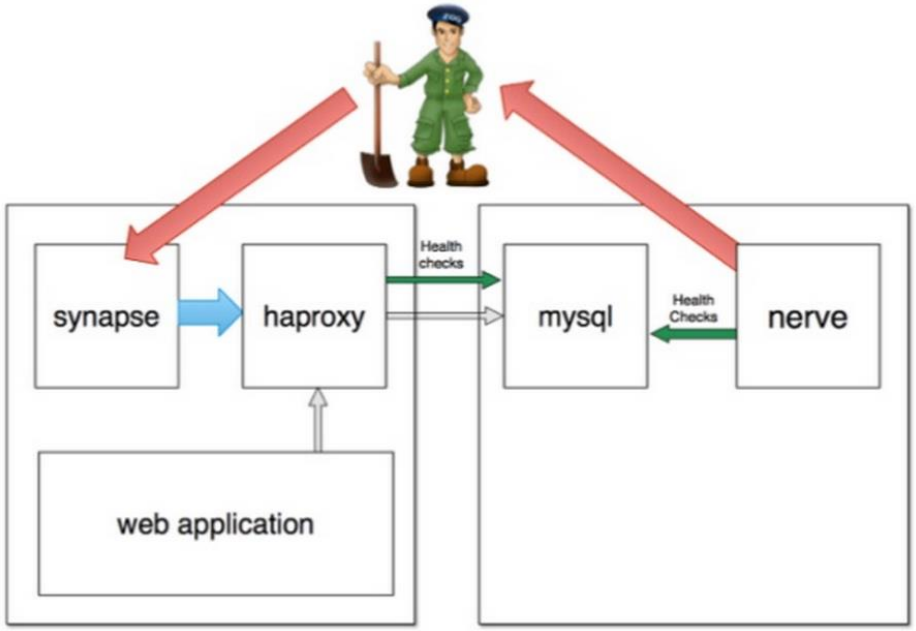
- **Service Registration**:
  - Beobachtet das Cluster und erkennt neue und kranke/tote Knoten in seiner Gruppe.
  - Hinterlegt die aktuell aktiven Knoten im Konfigurationsspeicher.
- **Service Discovery**: Der Client kommuniziert mit dem Ambassador-Knoten, der die Anfragen aber möglichst effizient an einen Knoten der Gruppe weiterreicht.

Der Ambassador-Knoten kann dabei eine Reihe an Zusatzdienste erweisen bei der Verbindung zum Service (**Service Binding**):

- Load Balancing inklusive Failover
- Service Monitoring
- Circuit Breaker Pattern
- Throttling

Service Discovery am Beispiel Smartstack





Lösungen

Konzept	Teil-Konzepte	Lösungen
Service Discovery	• Dynamische Konfigurationsdateien / Umgebungsvariablen • Dynamischer DNS / Namensdienst • Ambassador / LB / Service-Router • API und Registratur • Failure Detection (Heartbeats, Gossip-Protokolle, stehende Verbindungen) • Membership Detection (Gossip-Protokolle, Broadcasts, Discovery-Protokolle)	• Consul (http://www.consul.io) • Serf (https://www.serfdom.io) • Skydock (https://github.com/crosbymichael/skydock) • Vulcand (http://de.slideshare.net/sttts/beyond-static-configuration) • SkyDNS (https://github.com/skynetservices/skydns) • Libswarm (https://github.com/docker/libswarm) • Eureka (https://github.com/Netflix/eureka) • Smartstack (http://nerds.airbnb.com/smartstack-service-discovery-cloud)

Quellen

Internet-Quellen

- Grundlagen von veteilten Systemen:
  - http://book.mixu.net/distsys/ebook.html
- Service Configuration Management:
  - http://de.slideshare.net/sttts/beyond-static-configuration
  - Das Gossip-Protokoll: https://www.serfdom.io/docs/internals/gossip.html
  - Das Raft-Protokoll: http://raftconsensus.github.io
  - Serf: http://iankent.uk/2014/02/27/getting-started-with-hashicorp-serf
- Service Discovery
  - http://de.slideshare.net/teemow1/container-orchestration
  - Das Ambassador Pattern: https://www.digitalocean.com/community/tutorials/how-to-use-the-ambassador-pattern-to-dynamically-configure-services-on-coreos
  - Service Discovery mit HAproxy und Serf: http://www.centurylinklabs.com/auto-loadbalancing-with-fig-haproxy-and-serf/?hvid=4kkCCH
  - Service Discovery auf Basis von Docker: http://jasonwilder.com/blog/2014/07/15/docker-service-discovery/
  - Service Discovery mit Smartstack: http://nerds.airbnb.com/smartstack-service-discovery-cloud/
- Resource Management (mit Mesos)
  - http://de.slideshare.net/tomasbart/introduction-to-apache-mesos
  - http://iankent.co.uk/2014/02/26/a-quick-introduction-to-apache-mesos
  - http://de.slideshare.net/pacoid/chug-datacenter-computing-with-apache-mesos