# QZusammenfassun Vss - Grundlagen TEIL SLM

# Programm

Definierter Start und Ende, nur Offline Update möglich, Deterministisch (Liefert nur reproduzierte Werte).

## VSS (Gruppe von Systemen und Programmen)

Wird möglicherweise nie angehalten, Akzeptiert die ganze Zeit Inputs, Konfigurierbar im laufenden Betrieb, Möglicherweise nicht Deterministisch, Event-Gesteuert

**Vorteile**: Verteilte Standorte, Parallele Verarbeitungen möglich, Skalierbarkeit, Fehlertoleranz, Verfügbarkeit

**Nachteile**: Erhöhte Komplexität, Performanzprobleme wegen Netzwerkverzögerungen je nach Design möglich, Zuverlässigkeit -> jedes Teilsystem kann einen Ausfall verursachen, Transaktionssicherheit

## Middleware

Für die Vereinfachte Implementierung von Systemen, Transparenz, Events, Security, Kommunikation, Abstraktion, Beispiel: Java RMI, Web Services

## ACID (Atomicity Consistency Isolation Durability)

Atomare Operation, alles oder nichts ausführen – Garantie, Hinterlässt immer konsistente Daten, Durch Abgrenzung stören Transaktionen einander nicht. Dauerhaft garantiert nach einem Abschluss einer Transaktion, dass die Daten dauerhaft gespeichert sind, auch bei Auftreten von Fehlern.

## CORBA (Common Object, Request Broker Architecture)

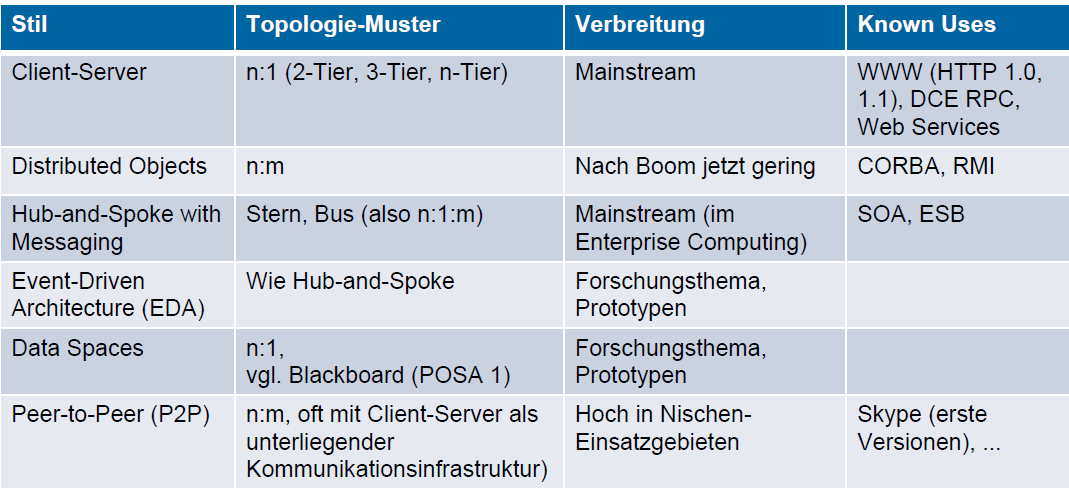
Spezifikation für eine Middleware für die Verbindung von verschiedene Softwareprogrammen zu einem System. CORBA-Middleware kann anstatt von Web Services eingesetzt werden. Ist Programmiersprachen unabhängig. Orbix ist eine Software die CORBA Standard implementiert. Mittels IDL (Interface Definition Language) wird eine Schnittstelle spezifiziert und später dann auch direkt Code daraus generiert. CORBA Einsatz heutzutage gering.

## Idempotent

Idempotente Methoden/Service usw. sind Methoden die beliebig oft aufgerufen werden können ohne einen Einfluss auf den Datenbestand im Server zu haben. Beispiel: «GET user» Request -> Idempotent, «POST new user» ist nicht idempotent.

# Enterprise and integration Patterns

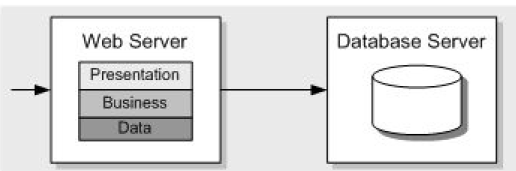
## Architekturstile (Architecure Patterns)



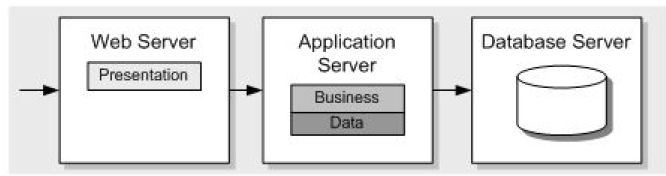
**Hub-And-Spoke:** Mainstream im Enterprise Computing. Benutzt einen Message Broker für das Verteilen von Nachrichten. Client bzw. Server verbinden zum Hub/Message Broker und müssen somit nicht für jeden Client/Server eine Verbindung aufbauen. Der Message Broker verwaltet die Verbindungen zwischen den Parteien.

## Deployment Patterns

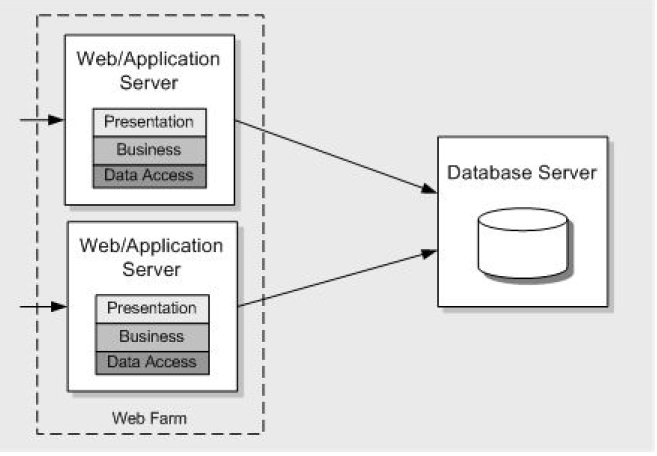
Non-Disributed Deployment

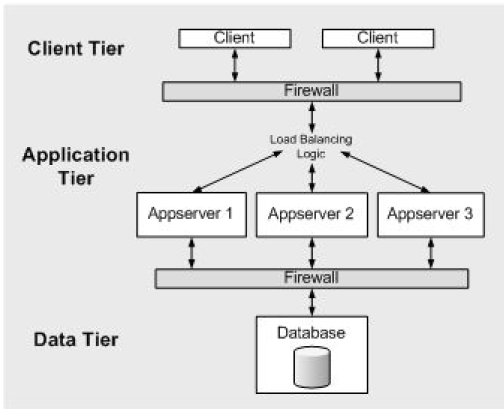


Distributed Deployment -> Vorteile: mehr Skalierbarkeit (Flexibilität), in der Regel bessere Performance, Kommunikationsaufwand darf aber nicht unterschätzt werden.

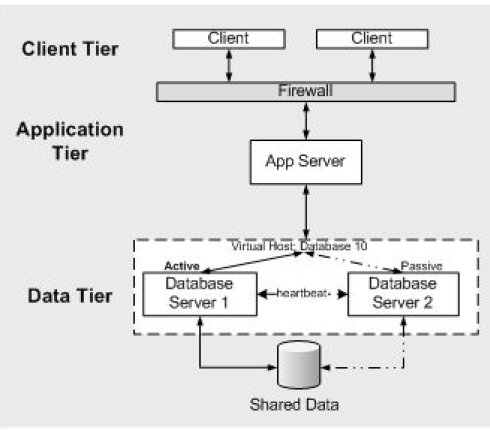


Webfarm



Load-balancing Cluster 

Failover Cluster



## Messaging Patterns

**Document Messages**: Transferieren von Daten zwischen Applikationen mittels Messages.

**Command Messages:** Führt Befehle anhand von Messages aus, sind grundsätzlich keine Spezielle Nachrichten, sie führen im Zielsystem einfach einen Befehl aus.

**Message Expiration:** Setzt ein Timeout auf einer Message um nicht mehr gültige Messages, welche zu lange Zustell-Zeiten haben, als ungültig zu erklären.

**Request-and-Reply:** Einfacher Einsatz eines Message Patterns, der Sender sendet eine Message und der Empfänger antwortet auch mit einer Message. 1<->1 Kommunikation.

**Push-pull:** Die Implementation von Publish und Subscribe Channel

**Fan-Out:** Verteilen von Messages auf unterschiedlichen Empfängern parallel. Der Sender wartet nicht auf eine Antwort.

## Messaging Channels

**Publish-Subscribe:** Anmelden von Pushmeldungen mit einem gewissen Topic oder gewissen Attributwerten um Messages zu erhalten. Typisch für mehrere Empfänger.

**Routing:** Die Auswahl des Consumers anhand eines Attributes.

**Point-to-Point Msg. Pattern** nur ein Empfänger, wichtig wenn Reihenfolge entscheidend ist. Sobald Parallelisierung ist die Reihenfolge nicht mehr gewährleistet.

## Messanging End Points

**Competing Consumer**: Mehrere Empfänger arbeiten Messages ab, eine Message wird nur einmal abarbeitet (Parallelisierung der Consumers).

**Selective Consumer:** Ein Consumer verarbeitet nur Messages welche seiner Filterkriterien entsprechen.

## System Managment Patterns

Für die Überwachung von Queue basierten Systemen

**Wire Tap**: Punkt zu Punkt versenden von Messages inkl. Kopie auf einem Inspektionskanal.

**Detour:** Umleiten von Messages für die Analyse, Testen und für Debugging.

**Test Message:** Sicherstellen vom Zustand von Message Systemen.

**Smart Proxy:** Adressen werden von Ausgehenden Nachrichten überschrieben und bei einer Antwort wieder rekonstruiert. Das Zielsystem kennt nicht die Ursprüngliche Adresse.

**Message Store**: Speichern von allen Nachrichten in einem zentralen Ort.

**Message History**: Speichert den Fluss von Nachrichten

**Channel Purger:** Entfernt alte Messages um Spuren von alten Tests zu bereinigen.

## Technologien und Ihre Messanging Patterns

**WCF**: Datagram (nur einseitig), Request-Reply, Duplex

**ZeroMQ:** Request-Reply (RPC, mehrere Clients auf mehreren Servern), Publish-subscribe (mehrere Clients auf mehreren Servern für Datenaustausch), Push-pull (verschiedene Nodes mit möglicher Parallelisierung, oder auch mehrere Empfänger für eine Message (Fan-Out Pattern)) , Exclusive pair (1 Client, 1 Server)

# Low-Level Remoting TCP/IP Socket

## Socket

Ein Socket ist ein Verbindungsendpunkt der vom Programm wie eine gewöhnliche Datei beschreiben und gelesen wird. Es wird beim Server ein fester Port gewählt um den Socket anzusprechen. Der Client kann ein dynamischer Port wählen, Der Client sendet nach connect() einen write() oder send() und wartet mit read() oder recv() **Nachteile von Sockets**: Keine Objekte, daher keine Typisierung. Message Exchange Pattern (MEP) muss selber implementiert werden.

**Verbindung verweigern:** Hört kein Prozess auf dem Ziel Host wird bei TCP ein ACK RST zurückgegeben, bei UDP ein ICMP Paket welches aber nicht an die Java API weitergereicht wird. **Timeouts:** Der Server benötigt keine Timeouts da beliebig auf Anfragen gewartet wird. **Teilen von Datenübertragungen:** Kann Sinn machen da UDP sehr schlank ist und die Steuerung sollte persistent sein-> TCP. Abbruch einer UDP Verbindung wird vom Server nicht sofort erkannt.

## MPI (Berkeley Sockets and Msg Passing Interface)

Gleiches Protokoll für alle Plattformen, Standard für den Nachrichtenaustausch, Sammlung von Operationen, kein konkretes Protokoll, Für Datenaustausch innerhalb von Systemen zwischen Prozessen aber auch Austausch zwischen verteilten Systemen.

## Netcat

Tool um UDP und TCP Protokolle zu testen. UDP und TCP Socket. Kann auch gut mit Java Socket erstellt werden.

## Java Socket API

**TCP** Erstellen von client Socket Socket(host, port); empfangen von Bytestream getImputStream(); (Blockierend) Server Socket der auf Requests wartet. ServerSocket.accept().

**UDP:** DatagramSocket(port) den Server Socket erstellen und der InetSocketAddress(host, port) für den Client. Der Client musst zuerst mal Daten Senden damit der Server den Port des Clients kennt. Darum socket.send(new DataPackat(bytesObj, bytesObj.lenght, InetAddress.getByName(‘localhost’), port)); danach mit while(true){ socket.recive(pkt); auf Lieferung von Server warten (Blockierend!). Warten auf mehrere Clients kann mit einer while(true){…} Schlaufe und jeweils starten eines neuen Threads Serverseitig erfolgen.

## WebSockets

neue Entwicklung von Sockets auf basis von HTML 5 -> Realtime Web Applikationen, hat nichts mit den 'alten' Sockets zu tun. HTTP Verbindungen werden vom Client geöffnet, danach kann Server an den Client Daten schicken ohne dass dieser eine Anfrage sendet. (Gegensatz zu reiner HTTP Verbindung)

## Service Activator

Der Service Activator kann eine Drittsoftware sein. Macht nicht anders als eine Message entgegennehmen und gibt sie weiter, Möglich als One Way oder auch als Two Way

# Asynchronous Messaging

Message Descriptor: Identifyer und Steuerinfos

Message Data: Inhalt der Meldung in der Struktur des Zielprogrammes. Queue-basiertes Messaging gestattet die flexible und lose Koppelung unterschiedlicher Systeme.

## MQ Manager Vorteil Persistent vs non Persistent

Persistent ist Teurer als non Persistent und muss eine Log nebenbei schreiben. Bei Persistenten Nachrichten sind nach Stromausfall/Absturz, keine Daten verloren gegangen.

## MQ von Microsoft

* Message Queue
* Vermehrt nur in der Windows Welt benutzt

## Aufruf von Queue

Consuming -> nach dem Lesen löschen

Non-Consuming -> nach dem Lesen noch verfügbar

## Queue von Java: Java Message Service (JMS)

Dokumentation notwendig um JMS anzuwenden

JMS ist erst später zur Projektorientierten Programmierung "hinzugefügt" worden. **Vorteil** Es können von Java <-> Java ganze Objekte ausgetauscht werden

**JMS Message Reliability Levels: Best effort nonpersistent**: Messages werden einfach verworfen, wenn ein System oder Netzwerk abstürzt oder überlastet ist. **Express nonpersistent**: Messages werden einfach verworfen, wenn ein System oder Netzwerk abstürzt. **Reliable nonpersistent**: Messages werden einfach verworfen, wenn ein System abstürzt. **Reliable persistent:** Messages könnten verworfen werden, wenn ein System abstürzt.

**Assured persistent:** Messages werden nicht verworfen.

## RabbitMQ

* OpenSoruce MQ
* Basiert auf AMQP (Standard protocol for the interoperability between all messaging middleware)

Work queues

Publish/Subscribe: Senden von Nachrichten zu ver. Empfängern gleichzeitig

Routing: Erhalten von Nachrichten nach Kriterien

Topics: Erhalten von Nachrichten anhand von Muster; Beispiel:

## RabbitHutch

Hypothetisches System zur Steuerung von Gebäudetechnik, welches für den Austausch von Daten RabbitMQ verwendet

# RMI & Web Services

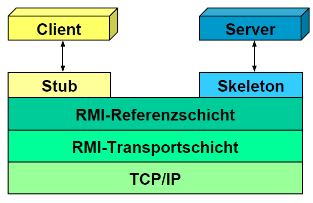
RMI ist RPC in Java, RMI kann ganze Java Objekte übermitteln. Interfaces und Klassen können vom Server zu Client und vom Client zum Server transportiert werden. Eine Aktualisierung einer Klasse kann nur auf dem Server stadtfinden. Der Client kann per RMI die neuste Definition automatisch holen, es muss somit kein Code auf dem Client aktualisiert werden. Die Kommunikation ist für den Entwickler verborgen.

## RPC (Remote Procedure Calls)

Aufruf einer Prozedur auf einem anderen System. Dabei wird der Prozedurname bzw. eine ID und die Parameter übermittelt. Es sind keine Spezifischen Messages, es sind einfach Messages mit Kommandos. Vereinfacht die Client/Server Programmierung. **RPC Aufruf:** 1. Client ruft eine Prozedur auf inkl. Parameter 2. Der *Client Stub* packt die Parameter in eine Message (marshalling) 3. Betreibsystem verschickt die Nachricht 4. Das Betriebssystem auf dem Server leitet die einkommende Messages an den richtigen *server stub* weiter 5. Unmarshalling auf dem Server 6. Stub ruft die Serverprozedur auf und verpackt die Antwort gleich. Während dieser Prozedur bleibt der Client Code meistens stehen und wartet auf die Antwort vom Server.

Asynchronous RPC: versenden/empfangen und verarbeiten von Nachrichten zu jeder Zeit.

## RMI Begriffe

Ein **Stub** Ist ein Stellvertreterobjekt (Remote Proxy), das Clientaufruf an Server weiterreicht. Ein **Skeleton** nimmt aufrufe des Stubs entgegen und leitet sie an Serverobjekt weiter. **Referenzschicht** = Namensdienst (Registry). **Transportschicht** = Verbindungen verwalten.

## Implementation von RMI

Serverseitig:

Public interface hello extends **Remote**{

String sayHello () throws **RemoteException**;

}

Public class HelloImpl extends **UnicastRemoteObject** implemnts Hello {

Public HelloImpl() threows Remote Exception{ super(); }

Public String sayHello() throws RemoteException{

Return "hello VSS";

} //create sever

Public class **HelloServer**{

Public static void main(String args[]){

**HelloImpl obj = new HelloImpl();**

**Naming.rebind("rmi://[hn]/remoteHello", obj);**

} //try and catch auch notwendig!

Clientseitig (Interface analog von Server):

Public class RmiClient {

Public static void main(String[] args){

**Hello obj** = (Hello)Naming.lookup("rmi://[hn]/remoteHello");

String message = **obj.sayHello();**

Sysout(message);

} //exeption handling mit try & catch notwenidg!

## Details von RMI

Wenn ein Objekt eine Instance von Remote ist wird die Referenz übertragen sonst der Wert (serialisiert). Server kann remote Wert ändern, Zugriff ist teuer.

**Wichtig im RMI ist** Timeout Management, gleiche JVM Version notwendig, Firewalls blockieren im Standard die Ports. **Blockierender Stub** hat den Vorteil, dass es einfacher implementiert ist, jedoch ist asynchrone Programmierung nicht mehr möglich.

## IDL (Interface Definition Language)

IDL ist eine Spezifikationssprache welche Interfaces zwischen verschiedene Plattformen, Programmiersprachen und Betreibsystemen beschreiben. Mit IDL werden RPCs Programmiersprachen unabhängig beschrieben und dienen den stubs für die Erstellung des Codes.

**Beispiele** CORBA.

## WSDL

RPC Style mit WSDL wird mit wie folgt definiert:

<soap:envelope><soap:body>

<methodenname>

<param1>value</param1>

<param2>value</param2></…></…></..>

Unterschied RPC Style und Dokument Style (Standard für Internetkommunikation): Applikation macht das Mapping der XML Werte, Wie die WSDL Datei aussieht entscheidet die Applikation. Die Applikation muss *marshalling* und *unmarshalling* handeln. (Marshalling => Umwandeln der Daten in Brauchbare Objekten)

# Operationale Modelle, Design for Performance

## Nicht funktionale Anforderungen (NFRs)

Verfügbarkeit, Compliance, Wartbarkeit, Performance (Reaktionszeit, Latenz, Durchsatz, Service Zeit), Privatsphäre, Wiederherstellung, Elastizität, Ausfalltoleranz, Robustheit, Skalierbarkeit, Usability

## Performance

Ein System ist nie schnell genug, es ist immer ein Gleichgewicht zwischen Kosten, Funktionalität und Performance.

## Zusammenhängende Aspekte von Performance

**Response Time** Steigt ab ca. 60% Auslastung exponentiell -> eingehende Anfragen müssen zuerst warten

**Throughput:** Druchsatz damit alle Konten einer Bank über Nacht abgeschlossen werden können. **Capacity**: Richtige Komponentengrösse um Auslastung zu Tragen. **Contingency**: Kann auch mit Ausnahmesituation noch umgehen -> Beispielsweise auch Jahresende Abschluss möglich.

# Design for Scalability and Availability

**Skalierbare Grösse:** Schnellere und mehr Komponenten (CPU, RAM, HDD, Switches usw.)

**Skalierung der Generation**: Technologie Evaluation

**Vertikale Skalierung**: Grössere/Schnellere Server

**Horizontale Skalierung**: Anzahl Server erhöhen.

## Load balancing

**IP Spraying**: eine IP Steht für die Clients zur Verfügung und der HTTP Verkehr wird vom Balancer einfach an verschiedene Webserver weitergeleitet. Das Verteilen von Anfragen erfolgt nach 3 Prinzipen: **Round Robin**, **Least-recently-used**, **Workload-based** (nur mit Monitoring)

## HTTP Load Balancing

Behalten von Session trotz Load Balancing:

* Speichern der Session Daten auf dem Server
* IP soruce affinity: Anhand der Soruce IP Adresse wird wieder gleicher Server zugewiesen: Problem, was wenn der User IP Adresse wechselt?
* Application layer presistence: Balancer benutzt den Session Cookie um den User zu identifizieren

Identifizierung einer Session:

**Affinity**: Verwendet Layer unterhalb von Appl. Layer

**Presistence**: verwenden von Applikation Layer

**Session affinity:** verwendet Cookies

## Availability

System verfügbar und beantwortet Anfragen korrekt.

**MTTF**: durchschnittliche Zeit bis ein System ausfällt

**MTTR**: durchschnittliche Zeit bis ein System nach einem Ausfall wieder online ist.

**MTBF**: Zeit zwischen 2 Ausfällen

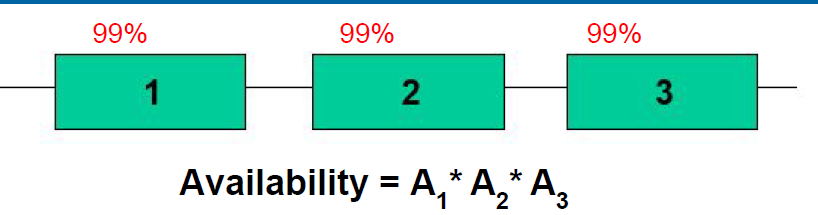
**RPO**: Maximale Zeit für Datenintoleranz nach Ausfall.

**RTO**: die Maximale Zeit bis ein System wieder läuft nach einem Ausfall

Non Functionl Requiment (NFR): RTO + RPO

Kosten für die Verfügbarkeit steigen Exponentiell. Der Kritische Punkt ist bei ca. 95% Verfügbarkeit.

## Verfügbarkeit berechnen





**Verfügbarkeit verbessern**: Vermeiden von Single Point of Fail (SPoF)

**Cold Standbay:** Das Backupsystem läuft nicht und hat keine aktuellen Daten. Wird bei einem Ausfall von den primären Komponenten hochgefahren und der Backup der Daten wird restored. Ausfallzeit von ein paar Stunden.

**Warm Standby:** ein Watchdog überwacht das produktive System, antwortet dieses nicht, werden die Dienste auf dem Backupserver aktiviert. Verbessert MTTR. Bei Überlastung vom Produktiven Server kanns passieren, dass der Backupserver fälschlicherweise hochgefahren wird, Konflikte können die Folge sein.

**Hot Standby**: min 2 Nodes laufen immer. Daten werden immer wieder auf dem Backupserver kopiert. Recovery innerhalb von Sekunden.

**Active-Active (Load Balancing):** Ausfälle sollten nicht bemerkbar sein.

**Fail Fast Principle (Circuit Breaker Pattern):** Anfragen auf einem Service werden überwacht, wenn es Ausfälle/Fehler gibt, wird der Service gestoppt um nicht Folgefehler zu verursachen. Oft kann der Fehler auch automatisch behoben werden und dementsprechend den Dienst wieder starten.

## Scalability

**Size Scalability:** Bessere und mehr Komponenten **Generation Scalability**: Technolgy Evolution, **Vertical Scalability, Scale Up**: Grössere Server/Bandbreite/Storage **Horizontal Scalability Scale Out:** Anzahl Server/Nodes erhöhen.

# VSS Management

Richtige Verwaltung von Systemen ist ein grosser Bestandteil für die Verfügbarkeit von Systemen.

Monitoring, Trend Analyse, Configuration Mgm. Usage, Performance Monitoring, Security (überwachen von Login)

## ITIL

**Service Strategy:** Financial, Service Portfolio, Strategy

**Service Design:** SLM, Availability, Capacity, Service Catalog, Information Security, Requirements Engineering

**Service Transition:** CM, CMDB, Release, KM, Validation and Testing

**Service Operation:** IM, PM, RF, Access, Event

**Continual Service Improvement**: Service Improvement, Service Measurement, Service Reporting

## JMX (Java Management Extensions)

Für die Verwaltung und Überwachung von Java (JVM, Java Virtual Machine). MBeans ist die API der JMX. JMX wird in 3 Ebenen unterteilt **Distributed Servcie Level:** Schnittstelle für die Kommunikation gegen aussen (ausserhalb der JVM). **Agent Level**: Agent die die Kommunikation mit den Ressourcen übernehmen**. Instrumentation Level**: Auf diesem Level befinden sich die Ressourcen, die überwacht und konfiguriert werden.

JMX kann auch verwendet werden um Konfiguration in einer laufenden Software zu ändern. Achtung das Ändern der Konfiguration kann zu inkonsistenten Zuständen führen: **Complexity** wird erhöht, **Availability** kann erhöht werden, da kein Neustart der Applikation notwendig ist um neue Konfiguration zu laden, **Stability**: kann negative Auswirkungen haben (erhöhte Komplexität).

## Logging

Easy to use, Zentralisiert, kleiner Effekt auf die Performance, verwenden von Loglevels, Standard Formate und Tools, Vermeiden von W Fragen in Logs (Wer, Wann, Wo..usw) **Verschiedene Zielgruppen: Entwickler:** Technische Details, welche Methode, Variablenwerte usw. **Systemadministrator:** Netzwerkauslastungen, Start/Stop Protokoll, Konfigurationsfehler. **Supporter**: Vorgehen von User nachvollziehen.

**Loganforderungen für VSS:** Lognachrichten müssen an Instanzen gebunden werden können. Bei Netzwerkproblemen dürfen keine Nachrichten verloren gehen. Verteilte Aufrufe müssen ersichtlich sein.

# CFIA (Component Failure Impact Analysis)

Erkennen von kritische Verfügbar- und Performancerisiken, wie auch spezifizieren von Messgrössen. Prozess der beim Design eines Systems durchgeführt werden muss. Massnahmen welche getroffen werden können um Ausfälle zu vermeiden oder Ausfälle zu verkürzen. Phase 1: Verbessern was sofort gemacht werden kann (innerhalb von Wochen) Phase 2. Hoch Priorisierte Tasks, Phase 3. Tiefer Priorisierte Massnahmen oder grosse Änderungen in der Architektur.

**Expanded CFIA**: für jeden CI folgende Fragen beantworten: Was ist es? Was kann schiefgehen? Wie kann ich es wiederherstellen? Was passiert, wenn dieses CI ausfällt? Wer ist zuständig? Usw.

**Risk Rating Tabelle:** Beantworten folgenden Fragen mit Zahlen von 1-10 = 1 sehr gut, kleines Risiko/ 10 sehr schlecht, grosses Risiko: Businessimpact, Häufigkeit des Ausfalls, Wie einfach ist es den Fehler zu erkennen, vorhandenes Wissen für die Wiederherstellung, Wiederstellungsmassnahmen, Automation im Wiederherstellungsprozess, Sicherheitsschutz (wie exponiert), Patchintervall/-status

**Allgemeine Massnahmen:** Eliminieren von SPOF, aktuelle und ausreichende Dokumentation, Monitoring, Automatisierte Wiederherstellung von Systemen, Angemessene Backups, eliminieren von «Key Personen» in kritischen Systemen.

## CFIA Matrix

|  |  |  |
| --- | --- | --- |
| **CI** | **Service 1** | **Service n** |
| **Pc1** | X | A |
| **App1** | M |  |

**X**-> S1 nicht verfügbar ohne PC1; **A** -> Redundanter PC1 verfügbar, kein Ausfall wenn nur PC1 ausfällt **M**-> Wie A nur manuell eingreifen nötig. Leer App1 hat keinen Einfluss auf Service n. CIs mit den meisten X sind am Kritischen.

# Verschiedenes

## CloudSum

Javaanwendung die als Client oder Server gestartet werden kann. Der Server ist nicht idempotent. Setzt für die Kommunikation ZeroMQ ein.

## ZeroMQ

Kommunikations-Library für die Kommunikation zwischen Prozessen und Threads, Neben TCP Kommunikation wird auch **inproc** als Kommunikationsprotokoll innerhalb eines Prozesses angewendet.

**Intraprozesskommunikation:** Kommunikation innerhalb eines Prozesses beispielsweise mit **inproc.** Kein Verbindungsaufbau nötig. Vorteil von Intrapozesskomm. In vergleich mit gemeinsamem Speicherzugriff: keine Race conditions, Synchronisation nicht notwendig.

**Interprozesskommunikation:** Datenaustausch über bsp. TCP. Je grösser die Nachricht desto grösser der Kommunikationsaufwand verglichen mit Intraprozesskomm.

## Weitere Tools

**Redis**: Persistent Datastore, Frei erhältliche KeyValue Data Store der auch als Verteilter Cache benutzt werden kann. Redis verwendet für die Kommunikation **RESP** (Redis Serialization Protocol). RESP kann verschiedene Datentypen serialisieren, wie Integers, Strings, Arrays. Befehle und Argumente werden als StringArray am Server versendet.

**Chirp:** Twitter ähnliche Applikation,Verwendet Redis, Chirp verwendet Request/Replay als Message Exchange Pattern, Aufrufe sind Synchron,

**Gatling:** Stress Tool für Balckbox Tests

**Slf4j:** Logging für API Logging, eingesetzt von Chirp.

**Logback**: Loggingframework von Chirp eingesetzt.

# Zusammenfassung Vss - TEIL MEF

# Bitcoin

Ein Bitcoin ist nichts anders als ein durch Transaktionen gesichertes Guthaben welches in einem Peer-2-Peer Netzwerk in einer Block-Chain gespeichert ist. Geschützt durch deinen Privaten Key welcher benötigt wird den dir zugeteilten berechneten Betrag weiter zu überweisen.

## Transaktion

Eine Transaktion wird mit deinem Privaten Schlüssel garantiert und muss durch das Peer2Peer Netzwerk bestätigt werden. Eine Bestätigung dauert min. 10 min bis hin zu x Stunden. Dies hängt auch vom freiwilligen bezahlten Transferbetrag ab, je grösser desto schneller wird ein Transfer bestätigt

## Blocks

Blocks werden verwendet um die Reihenfolge für Transaktionen zu ordnen. Die verschiedenen Blocks werden in einer Block Chain gelegt welche alle zu den führenderen Blocks referenziert werden (-> Block Chain).

Blocks werden durch generieren von SHA256 Hashes mit Zufallszahlen erzeugt. Das Erzeugen von Blocks nennt sich Mining. Dafür sind grosse Rechenleistungen notwendig. Je mehr Leute die Hashes erzeugen desto grösser wird die Schwierigkeit den nächsten Hash bzw. Block zu finden. Die Währung reguliert sich selber, so dass ca. pro 10 min ein Block generiert wird und jeden 14. Tag wird die Schwierigkeit angepasst. Die Momentane Grösse der Blocks von 2 MB lässt 3 Transaktionen pro Sekunde zu.

## Confirmation

Je mehr Bestätigungen eingetroffen sind, desto sicher ist es, dass die getätigte Transaktion gültig ist. Für grössere Beträge macht es Sinn bis zu 6 Bestätigungen abzuwarten.

## Schwächen

Wenn ein Minor, mehr als 50% der auf der Welt für minen zur Verfügung gestellte Rechenpower verfügt kann er seine ‘eigene’ block-Chain machen und so bestimmen welche Transaktionen akzeptiert werden.

# Naming

Naming Systeme werden benötigt um Entitäten im Netzwerk zu benennen und so ‘erreichbar’ zu machen.

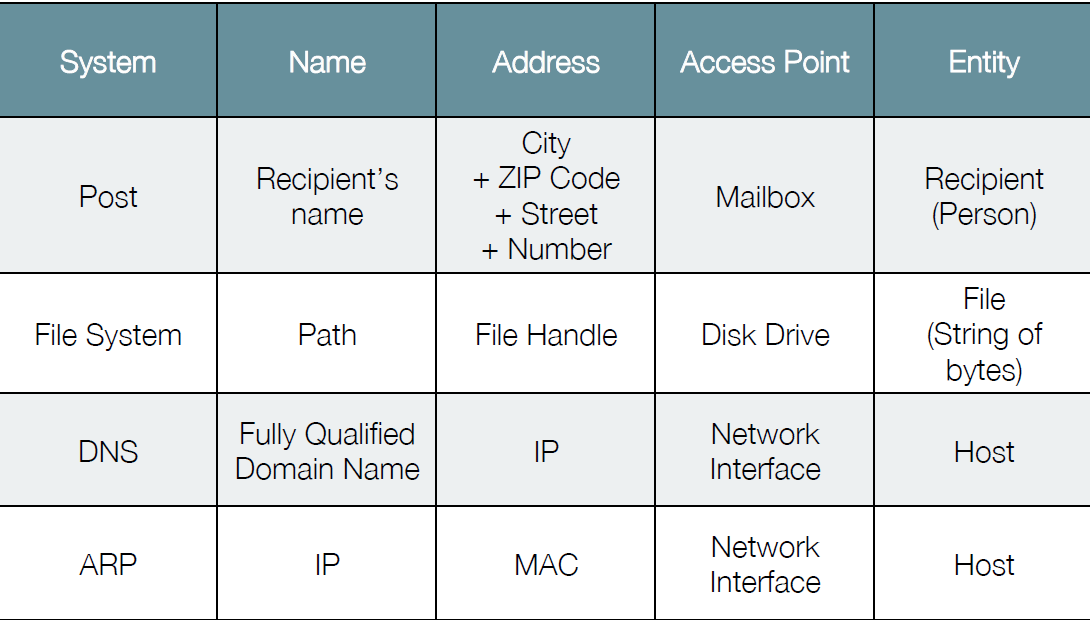
**Entity**: Ein Objekt das Adressiert werden möchte.

**Name**: Bytestring als Referenz zu einer Entität

**Access Point**: Spezielle Entität die benötigt wird um auf andere Identitäten zuzugreifen

**Address**: Name von einem Access Point

**Identifier**: Ein Name der auf höchstens eine Entität referenziert.



## Flat Naming

Ein Name der aus dem Name keine Informationen zum Objekt schliessen lässt. Nicht Strukturiert. Eine Broadcast Domäne ist ein gutes Beispiel für eine Flat Naming Umgebung. Jede Entität hat eine Adresse, und fragt «was ist die Adresse zum Namen xyz»

## Structural naming

Namen sind strukturiert wie bsp ein DNS Name. Hohe Ansprüche an einen Namensserver. Kann gut gecached werden

Interative resolution: Wenn der DNS Server den angefragten Host nicht kennt, setzt er interactive Request beginnend zuhinterst vom DNS Name (Bsp. .com) ab um die Anfrage vom Client zu beantworten

Recursive Resolution: Anfrage vom Client an den DNS Server

## Attribute-based Naming

LDAP (mix aus Strukturbased und Attribute Based naming)

## Bojour/Zeroconf

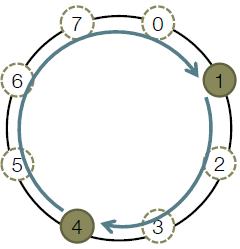
Protokolle, die mehrere Naming Standards verwenden bsp: DHCP. Funktioniert nur Lokal, alle müssen dem DHCP Server vertrauen.

# Distributed Hash Table (DHT)

## Chord

P2P Lookup Service, Verbindet P2P Standorte im Netzwerk. Ermöglicht das Suchen im P2P Netzwerk

Aufbau: p ist der Hash der Host IP Adresse, jeder Node hat die Adresse für seinen Nachfolger und Vorgänger Node:



**Chord Lookup**

Geschwindigkeit O(log n), n = anzahl Nodes im Netzwerk. O(log^2 n) Messages sind maximal notwendig um einen key zu finden.

**Node Adresse:** SHA1(IP + Port) -> die ersten m Zeichen davon. m wird definiert.

**Finger Table**: Beinhaltet alle direkten Nodes für jeden Client. Nämlich seine id + 2 hoch i - 1 (i = Laufvariable 1 bis m). Die Finger Table hat m Einträge.

**Files Stored:** Mit dem Key vom File (Bsp: Filename) wird durch hashen: SHA1(key) einen Schlüssel generiert welcher beim Node der >= sein Schlüssel ist abgelegt.

**Ausfallsicherheit**: Die Ausfallsicherheit von einem chord P2P Netz wird dadurch, dass alle Dateien nicht nur auf dem dafür vorgesehenen Host abgelegt, sondern auch noch auf dem nächsten vorherigen Host mit 1 – (1/n^2) nahezu Unendlich.

**Stabilisierungsprotokoll:** Stabilisierungsprotokoll aktualisiert seinen Vorgänger und seinen Nachfolger. Jeder Knoten stabilisiert sich periodisch unabhängig und prüft ob sein Nachfolger und Vorgänger noch verfügbar sind.

## Lookup

Lookup, also ein P2P Netzwerk durchsuchen, ist eine grosse Herausforderung für ein P2P Netzwerk. Es gibt verschiedene Variaten:

**Broadcast eine Suche an alle Peers** (immer an die bekannten Peers weitergegeben): Problem hohe Last mit Suchanfragen.

**Zentraler Server, der ein Verzeichnis mit allen Standorten führt**: Bottleneck, 1. Angriffspunkt.

Suchbefehle wie DNS organisieren: Hohe Last auf «Root Server» Ausfall von «Root Server» führt zu einem Zusammenbruch

**Symetrische Suche**, Suche von Node zu Node bis ein Suchergebnis gefunden worden ist: Anonymität nicht gewährleistet da Standort der Daten bekannt.

**DHT als Suche:** Verschiedene Algorithmen implementieren die Distributed Hash Table Suche. Das Grundprinzip ist jedoch immer das gleiche. Es wird eine lokale Tabelle mit ausgesuchten Nodes geführt.

# Synchronization

Wir können die Reale Zeit nicht "genau" Messen. Millisekunden sind schon schwierig, Nanosekunden unmöglich

## .beat

Anstatt von Stunden und Minuten wurde ein Tag in 1000 Stücke aufgeteilt. Ein Beat dauert 86.4 Sekunden. Es gibt keine Zeitzonen BMT (Biel Meantime) basiert auf dem Hauptstandort von Swatch.

## Logical vs. Physical clock

Die Physische Zeit ist die für uns interessante Zeit. Die Logische Zeit kann eine aufzählende Nummerierung sein welche für eine gewisse Applikation die Reihenfolge von Transaktionen bestimmt. Die Logische Zeit ist nicht an die Physikalische Zeit gebunden.

## Lamport’s logical clock

Als Uhr wird ein Counter eingesetzt welcher bei jedem Event um 1 erhöht wird. Der Counter muss immer > sein als der erhaltene Counter, wenn das nicht der Fall ist muss sein Counter überschrieben werden. Pro Prozess oder Host wird noch eine eindeutige ID mitgeschickt damit auch zeitgleich und mit dem gleichen Counter ereignete Events unterschieden werden können.

## Vector clocks

Erweiterung von Lamport’s Uhr. Jeder Prozess/Host hat seine eigene Uhr und schickt jeweils einen Vektor mit allen Uhren an die anderen Prozesse/Hosts weiter. Inkrementiert wird nur die eigene Uhr, jedoch jeweils die anderen Uhren werden aktualisiert. Es können nebenläufige Events erkannt werden.

## Kausalität

Kausalität bezeichnet die Beziehung zwischen Ursache und Wirkung: Aktion und Reaktion.

**Transitive**: (a 🡪 b ^ b 🡪 c) => a 🡪 c [wenn a vor b, b vor c dann ist auch a vor c]

**Irreflexiv**: a ↛ a [kein Element in der Relation steht zu sich selbst]

**Asymmetrisch**: a 🡪 b => b ↛ a [wenn a vor b gilt aber nicht b vor a]

**Concurrency** (gleichzeitig/nebenläufig/Parallelität): a || b -> Heisst 2 Berechnungen können parallel ablaufen, möglicherweise völlig unabhängig, können sich jedoch integrieren, sprich Resultate austauschen. Es kann, muss jedoch nicht Parallel ausgeführt werden.

# Map-Reduce / Spark

Map-Reduce bzw. Spark kann eingesetzt werden um über einen Cluster Logfiles zu analysieren und die Werte daraus zu entnehmen. Ein Node um grosse Logs zu verarbeiten sollte folgende Eigenschafen haben: angemessene Reaktion, Resistent (bsp. Bei Ausfall von einem Node) Skalierbar.

## RDD

Deklarative API zum Beschreiben von Schritten auf einer Collection. Code wird auf irgendeinem Node ausgeführt. Cache und Wiederholung muss beachtet werden. Vorteile zu Parallel Stream: Schneller, Abbrechbar, elastisch

## RDD - Scala

Programmiersprache um beispielsweise Spark zu bedienen.

val t = ('a', 2, 1) //Array mit Tuples erstellen

t.\_1 == 'a' && t.\_3 == 1 //ergibt true

var t = 1 //val gleich final var normale var.

t = 2

println (t) //println -> Ausgabe mehrere Werte

names.map(s => { //lamda

val length = s.length

val firstChar = s(0)

println(firstChar, length)

}) //Ausgabe 1. Buchstabe + länge von String

**Scala mit RDD**

val nums = sc.parallelize(0 to 1000)

nums.reduce (a, b) ) => a + b) //summe

nums.count //anzhal berechnen

nums.takeSample(true, 10) //Zufallsdaten entn.

nums.filter(n => n % 12 == 0) //filter mit Mod.

lengthWithNum.groupByKey.map(pair => (pair.\_1, pair.\_2.size)).collectAsMap // 1. Gruppieren, dann die Anzahl berechnen

names.map(s => (s(0), s)).collectAsMap //S->Susi

**//get from File -> map -> reduce by key -> col.**

val wordCounts = sc.textFile(“hdfs:/words”)

.map(w -> (w, 1))

.reduceByKey((a, b) -> a + b)

.collect()

**//NUR Error codes und diese gezählt**

accessLog.filter(e => (e.status > 399)).map(al => (al.method, al.path, al.status)).countByValue

**//Parsen von Werten**

case class Entry(time: String, thread: String, level: String, logger: String, msg: String, req: String)

val LogEntryP = """^([0-9:]+)\.[0-9]+ +(\S+) +(\S+) +(\S+) +- +(.+) +(\[\S+\])$""".r

def **parseEntry**(s: String): Option[Entry] = {

s match {

case LogEntryP(time, thread, level, logger, msg, req) => Some(Entry(time, thread, level, logger, msg, req))

case \_ =>

None }}

val rawLog = sc.textFile("frontend.log")

**val messages = rawLog.flatMap(parseEntry)**

# Peer-to-Peer Network

Peer to Peer Netzwerk wächst kontinuierlich ca. 50% pro Jahr.

**Zeile:** Mehr Performance und Verfügbarkeit

**Tracker:** Hilft den Client einander zu finden. Koordiniert das Netzwerk über Port 80 UDP/http

**Web Server:** Publizieren von .torrent Files

Client: Upload und Download von Files, Hört Standardmässig auf Ports 6881-6889

**Torrent File:** beinhaltet alle Informationen über das File, wie Anzahl Teile, Grösse, Länge (Anzahl Files/Verzeichnisse)

**Fairness**: Wer mehr hoch lädt wird nicht gedrosselt.

## File

Eine Datei wird in 256KB grosse Blöcke aufgeteilt, welche jeweils mit einer SHA-1 Hash verifiziert werden können. Es werden jeweils 16Kb grösse Paketstücke priorisiert: 1. Selten vorhandene Stücke, 2. Erste Stücke zuerst damit auch Upload möglich ist. 3. Ausstehende Teile werden von mehreren Clients angefordert.

# Stichwortverzeichnis

|  |  |
| --- | --- |
| ACID 1.4  AMQP 4.5  Asymmetrisch 15.5  Asynchronous Messaging 4  Availability 7.3, 7  beat 15.1  Bitcoin 12.1  Bojour 13.4  Capacity 6.3  CFIA 9  CFIA Matrix 9.1  Channel Purger 2.6  Chirp 10.3  Chord 14.1  CloudSum 10.1  Concurrency 15.5  Contingency 6.3  CORBA 1.5  Deployment Patterns 2.2  Deterministisch 1.1  Detour 2.6  DHT 14  Enterprise and integration Patterns 2  Fan-Out 2.7, 2.3  Flat Naming 13.1  Gatling 10.3  Hub-And-Spoke 2.1  Idempotent 1.6  IDL 5.5, 1.5  inproc 10.2  Interprozesskommunikation 10  Intraprozesskommunikation 10  IP Spraying 7.1  Irreflexiv 15.5  ITIL 8.1  Java Socket API 3.4  JMS 4.4  JMX 8.2  Kausalität 15.5  Lamport’s logical clock 15.3  Load balancing 7.1  Logback 10.3  Logging 8.3  Map-Reduce 16  Middleware 1.3  MPI 3.2 | MTBF 7.3  MTTF 7.3  MTTR 7.3  Marshalling 5.6  Message Store 2.6  Naming 13  Netcat 3.3  NFR 7.3  NFRs 6.1  Peer-to-Peer Network 17  Performance 6.2  Presisten vs non Presistent 4.1  RabbitHutch 4.6  RabbitMQ 4.5  RDD 16.1  Redis 10.3  RESP 10.3  Response Time 6.3  Risk Rating Tabelle 9  RMI 5  RPC 5.1  RPO 7.3  RTO 7.3  Scala 16.2  Scalability 7.5, 7  Service Activator 3.6  Slf4j 10.3  Smart Proxy 2.6  Socket 3.1  Spark 16  Structural naming 13.2  Stub 5.2  Synchronization 15  Throughput 6.3  Torrent File 17  Transitive 15.5  Vector clocks 15.4  Verfügbarkeit berechnen 7.4  WCF 2.7  WebSockets 3.5  Wire Tap 2.6  WSDL 5.6  Zeroconf 13.4  ZeroMQ 10.2, 2.7 |