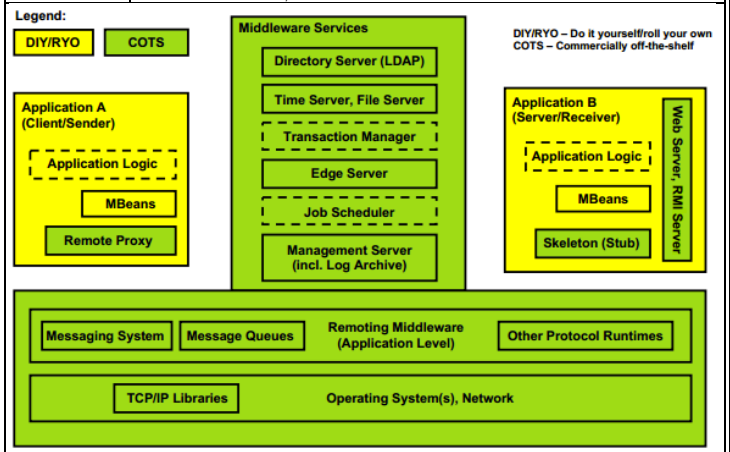
VSS

Begriffe

Idempotent	Eine Methode ist idempotent, wenn sie immer die gleichen Effekte beim Empfäger zur Folge hat. (wennn sie mehrere Male aufgerufen wird)
Middleware	infrastrukturelle Software zur Kommunikation zwischen Software-Komponenten und Anwendungen auf verschiedenen Computern. Vorteile: Vereinfachung, überwinden Heterogenität (erhöht Portabilität) Bsp: TCP/IP, Sockets, Java RMI, Web Services (WSDL/SOAP, REST)
ACID	Atomicity (Ganz oder gar nicht), Consistency (Nach jeder Transaktion sind die Daten konsistent), Isolation (Nebenläufige Ausführungen beeinflussen sich nicht), Durability (Auswirkungen bleiben persistent)
RPC	Remote Procedure Call: Technik zur Realisierung von Interprozesskommunikation. Ermöglicht Aufruf von Funktionen in anderen Adressräumen
SOAP	Simple Object Access Protocol: Netzwerkprotokoll, mit dessen Hilfe Daten zwischen Systemen ausgetauscht und Remote Procedure Calls durchgeführt werden
REST	Representational State Transfer: a way of providing interoperability between computer systems on the Internet. REST-compliant Web services allow requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations.

Transparency Types (ISO 1995)

Access	Hide differences in data representation and how a ressource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource



Edge Server: z.B. Load Balanzer → Zugang zu einem VSS für einen Client

Architekturstyle: Client-server, Distributed objects, Hub-and-spoke, Event-Driven Architecture (EDA), Peer-to-Peer (P2P)
Drei Dimensionen, die beeinflusst werden je nach Architekturstil: **Nebenläufigkeit, Persistenz, Verteilung**
Designaspekte für VSS:

- Communication Topology (Clients, Network, Server Nodes) → One client, one server? Many clients p. Server? Dynamic?
- Location Autonomy (Transparency) → Real vs. virtual address (what if a server is moved?)
- Invocation and Message Semantics → Bytestream, Document, Procedure, Remote Object
- Timeout Management → 1ms? 30s? Infinite?
- Error Handling → Retry request? Make server idempotent?

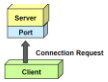
TCP Sockets

Sockets im Überblick

Basis Mechanismus für alle komplexeren Verfahren(http, RMI, ...)	- Austausch von Bytesströmen auf Programmierebene - Socket ist Verbindung zwischen zwei Endpunkten (IP + Port) - Zwei Rollen: Client / Server
Nachteile von Sockets	- Konstruieren und Parsen der Bytesströme erforderlich (keine Typsicherheit → XML/JSON) - Message Exchange Pattern muss selbst implementiert werden
Eight fallacies of Distributed Systems	The network is reliable, latency is zero, bandwidth is infinite, the network is secure, topology doesn't change, there is one admin, transport cost is zero, the network is homogenous

Socket Programmierung: Grundkonzepte

Client kennt Hostname und Port für Connect mit Server
1. stellt Connect her
2. erhält Socket-Objekt
3. kommuniziert mit Methoden des Socket-Objekts (über Streams)



Server kennt eigenen Port für Listen-Modus
4. Listening: wartet auf Client-Request über Connect-Port
5. akzeptiert Client-Request
6. erzeugt für jeden Client-Request neuen Socket zur Kommunikation (mit neuem Port!)
7. ursprünglicher Socket steht für weitere Requests zur Verfügung

Advantages and Disadvantages of Java TCP IP Sockets

Advantages:
- flexible and powerful
- low network traffic if efficiently used
- sufficient to send only updated information

Disadvantages:
- Can send only raw data
- need to interpret the data (both Client &Server)
- need to keep state information (both Client & Server)

TCP Socket

```
public class TcpClient implements Runnable {

  private final int serverPort;
  private final String host;

  public TcpClient(String host, int serverPort) {
    this.serverPort = serverPort;
    this.host = host;
  }

  @Override
  public void run() {
    try (Socket server = new Socket()) {
      server.connect(new InetSocketAddress(host, serverPort), 1000);
      server.setSoTimeout(2000);

      BufferedReader reader = new BufferedReader(new
InputStreamReader(server.getInputStream()));

      String msg = reader.readLine();

      System.out.println("tcp client> Received" + msg);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

```
public class TcpServer implements Runnable {

  private final int port;

  public TcpServer(int port) {
    this.port = port;
  }

  @Override
  public void run() {
    try (ServerSocket server = new ServerSocket(port)) {
      System.out.println("tcp server> bound to " + server.getLocalSocketAddress());

      while (true) {
        try (Socket client = server.accept()) {
          System.out.println("tcp server> accepted connection from " +
client.getRemoteSocketAddress());

          PrintWriter writer = new PrintWriter(client.getOutputStream(), true);
          writer.println(new Date().toString());
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

UDP Socket

```
public class UdpClient implements Runnable {

  private final int serverPort;
  private final String host;

  public UdpClient(String host, int serverPort) {
    this.serverPort = serverPort;
    this.host = host;
  }

  @Override
  public void run() {
    try (DatagramSocket socket = new DatagramSocket()) {
      // check if server is up. wait for 2s to receive data from the server
      socket.setSoTimeout(2000);

      InetSocketAddress serverAddr = new InetSocketAddress(host,
serverPort);
      DatagramPacket initPacket = new DatagramPacket(new byte[] {0, 1,
2 }, 3, serverAddr);
      System.out.println("udp client> will send initial packet to " +
serverAddr);
      socket.send(initPacket);

      byte[] responseBuffer = new byte[256];
      DatagramPacket response = new DatagramPacket(responseBuffer,
responseBuffer.length);
      System.out.println("udp client> waiting for response");
      socket.receive(response);

      System.out.println("udp client> Received " + new
String(responseBuffer, 0, response.getLength()));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

```
public class UdpServer implements Runnable {

  private final int port;

  public UdpServer(int port) {
    this.port = port;
  }

  @Override
  public void run() {
    try (DatagramSocket socket = new DatagramSocket(port)) {
      System.out.println("udp server> bound to " +
socket.getLocalSocketAddress());

      while (true) {
        // This method blocks until a datagram is received
        DatagramPacket initPacket = new DatagramPacket(new byte[1], 1);
        socket.receive(initPacket);
        System.out.println("udp server> received init pkg from " +
initPacket.getSocketAddress());

        String response = new Date().toString();
        byte[] bytes = response.getBytes();
        DatagramPacket responsePacket = new DatagramPacket(bytes,
bytes.length, initPacket.getSocketAddress());

        System.out.println("udp server> send reponse to " +
responsePacket.getSocketAddress());
        socket.send(responsePacket);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Websocket Protokoll (L5)

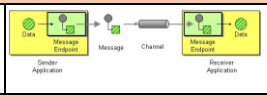
Ein auf TCP basierendes Netzwerkprotokoll, um eine bidirektionale Verbindung zwischen einer Webanwendung und einem Websocket-Server (bzw. Webserver) herzustellen. Der Client baut meistens direkt die Verbindung zum Server auf.
Vorteil: Http-Header die grossen Overhead verursachen fallen weg!

Basic Messaging Pattern

Applications connect to a messaging channel using a Message Endpoint, a client of the messaging system that the application can use to send or receive messages.

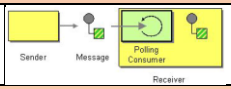
Blocking Receiver Message Pattern

Objective: create an application that waits for incoming messages.
Solution: use a synchronous messaging endpoint that blocks until a message is received.



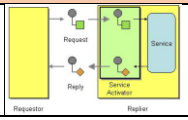
Polling (Non-Blocking) Receiver Message Pattern

Objective: create an application that can control when it consumes each message.
Solution: use a Polling Consumer, one that explicitly makes a call when it wants to receive a message.



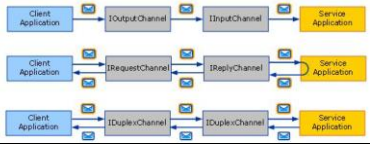
Service Activator Message Pattern

Objective: create a Service that can be invoked in an RPC-style both via various messaging technologies and via non-messaging techniques.
Solution: design and implement a Service Activator that connects messages on the channel to the service being accessed.



Message Exchange Patterns (defined by WCF)

- **Message Exchange Patterns:**
  - Datagram
  - Request-Reply
  - Duplex

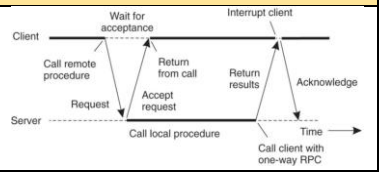


Basic Messaging Concept

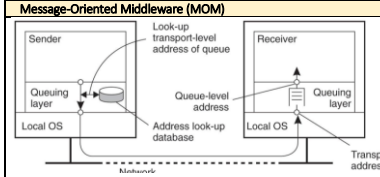
A message is a container that consists of:
- **Message descriptor**: Identifies the message and contains control information (e.g. message type, and priority) that is assigned to the message by the sending application.
- **Message Data:** Contains the application data.
- **The queue manager (not part of the message):** Is responsible for accepting and delivering messages, maintains queues of all messages that are waiting to be processed

Async Messaging

Queue-basiertes Messaging gestattet die flexible und lose Kopplung unterschiedlichster Systeme.
Message besitzt Descriptor (Messaage Type, Priority,...) und Daten. Die Queue Manager sind verantwortlich Messages zu senden und zu empfangen. Sie sind über logische Kanäle übers Netzwerk miteinander verbunden.
Wichtige Aspekte von Asyn Messaging sind:
loose coupling, location transparency, relocation transparency, dynamic, Einfachheit, Erweiterbarkeit, Skalierbarkeit, Fehlertoleranz

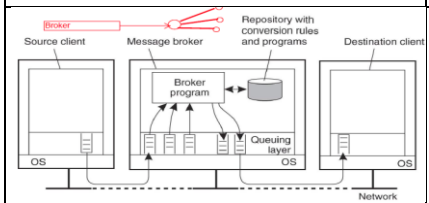


Message-Oriented Middleware (MOM)



Anzahl Verbindungen bei Peer-To-Peer bei n Teilnehmern: n * (n - 1) / 2
einige Vorteile, wie :
- geringe Kosten (die Kosten, die in ein solches Netzwerk fließen sind die für Hardware, Kabel und Instandhaltung)
- bewährte Einfachheit
- dezentral → keine zusätzliche Hardware, Felhertolerant
- Anhäufung von Ressourcen und Interoperabilität

Nachteile:
- das System ist überhaupt nicht zentralisiert und damit sehr schwer zu administrieren
- es gibt kaum Sicherheit
- kein einziges Glied im System ist verlässlich
- O(n^2) mögliche Connections



Hub-and-Spoke Architecture with Central Message
Verbindung zwischen zwei Endknoten A und B wird nicht direk geführt, sondern über einen Zentralknoten Z geleitet.
Anzahl Verbindungen in Abhängigkeit der Knoten n: n - 1 (von allen Knoten 1 Verbindung zum Hub ausser von Hub aus selber)
+ Weniger Verbindungen sind nötig als bei Peer-To-Peer
+ Kleinere Netzbelastung
- Single Point Of Failure
- Umweg über "Router"
- Hub kann Kommunikation abhören sofern diese nicht verschlüsselt ist.

MOM API Primitives (Platform independent)

Put	Append message to specified queue	
Get/Poll	Synch (block until queue is nonempty) /Asynch: remove first messageBrowse (vgl top() bei Stacks)	Non-Consuming Read
Notify	Install handler to be called when message is put in queue	

Implementations-Beispiele Message Queues

IBM MQ Persistent and Non-Persistent Messages
IBM MQ assures once-only delivery of persistent messages and it assures at-most-once delivery of non-persistent messages.

use persistant messages for critical business data (eg. receipt of payment for an order) → performance < data integrity
use non-persistant messages when loss of data is not crucial (eg. query data) → performance > data integrity

MS MQ
Message Queuing (MSMQ) technology enables applications running at different times to communicate across heterogeneous networks and systems that may be temporarily offline. Applications send messages to queues and read messages from queues.
can be used to implement solutions to both asynchronous and synchronous scenarios

JMS Java Messaging Service → Provider Bsp: RabbitMQ

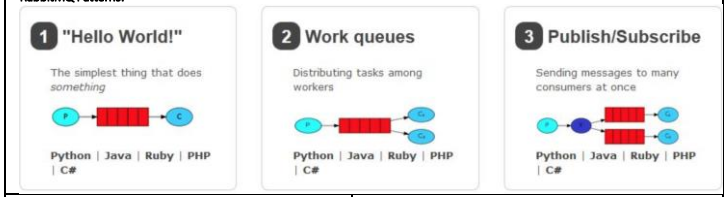
JMS hat das Ziel, lose gekoppelte, verlässliche und asynchrone Kommunikation zwischen den Komponenten einer verteilten Anwendung zu ermöglichen. JMS bietet verschiedene Chanels (Point-to-Point, Publish-Subscribe) sowie Message Types (Text, Serializable Object, Result Set, Binary) an.
JMS Message besteht aus Header (Infos zu Routing, Id), Properties (optional, Infos zum Filtern, ...), Body

JMS Message Reliablity Levels:
Best effort nonpersistent: Messages werden verworfen wenn eine Messaging Engine stoppt. Messages werden evtl auch verworfen wenn eine Connection unerreichbar wird oder aufgrund von beschränkten Systemressourcen unerreichbar ist.
Express nonpersistent: Messages werden verworfen wenn eine Messaging Engine stoppt. Ebenso wenn Connection unerreichbar.
Reliable nonpersistent: Messages werden verworfen wenn eine Messaging Engine stoppt.
Reliable persistent: Messages werden eventuell verworfen wenn eine Messaging Engine stoppt.
Assured persistent: Messages werden nicht verworfen.

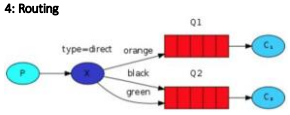
RabbitMQ based on AMQP

AMQP Model:
Messages werden auf **Exchanges** gepublished, vergleichbar mit Poststelle. Exchanges verteilen dann Messages auf **Queues** mit Regeln die sich **Bindings** nennen. AMQP Brokers stellen Messages den **Consumern** die auf diese Queue **subscribed** sind zu oder Consumer **fetchen/pullen** die Messages on demand.

RabbitMQ Patterns:

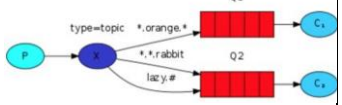


4: Routing



Example with direct exchange with two queues bound to it. The first queue is bound with binding key orange.
The second queue has two bindings, one with binding key black and the other one with green.
- Messages published to the exchange with a routing key orange will be routed to queue Q1.
- Messages with a routing key of black or green will go to Q2.
- All other messages will be discarded.

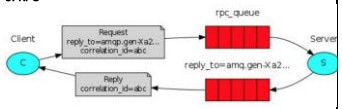
5: Topics



Messages have a routing key that consists of three words <celerity>.<colour>.<species> Q1 is bound "*.orange.*" and Q2 with "*.*.rabbit" and "lazy.#".
- Q1 is interested in all the orange animals.
- Q2 wants to hear everything about rabbits, and everything about lazy animals.

Messages with "quick.orange.rabbit" will be delivered to both queues. Messages "lazy.orange.elephant" also will go to both of them.
"quick.orange.fox" will only go to the first queue, and "lazy.brown.fox" only to the second. "lazy.pink.rabbit" will be delivered to the second queue only once, even though it matches two bindings. "quick.brown.fox" doesn't match any binding so it will be discarded.

6: RPC



When the Client starts up, it creates an anonymous exclusive callback queue.
For an RPC request, the Client sends a message with two properties: reply_to, which is set to the callback queue and correlation_id, which is set to a unique value for every request.
The request is sent to an rpc_queue queue.
The RPC worker (aka: server) is waiting for requests on that queue. When a request appears, it does the job and sends a message with the result back to the Client, using the queue from thereply_to field.
The client waits for data on the callback queue. When a message appears, it checks thecorrelation_id property. If it matches the value from the request it returns the response to the application.

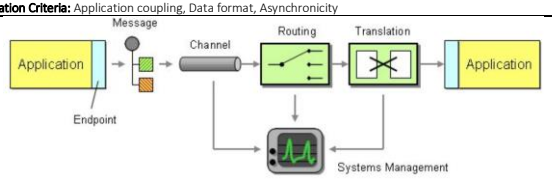
AMQP Exchange Types:

Headers Exchange	Routing Key wird ignoriert, dafür Routing basierend auf Message Headers. Falls gewisse Attribute matchen, wird Message an Queue geschickt (x-match: any/all)
Direct Exchange	Basierend auf einem Routing Key wird auf ein Exchange gehört (Unicast, aber Multicast auch möglich)
Fanout Exchange	Routing Key wird ignoriert. Eine Message wird an alle gebundenen Queues gesendet. (Broadcast)
Topic Exchange	Routing basierend auf Routing Key und das Pattern, welches verwendet wurd um die Queue an den Exchange zu binden. Wenn mehrere Consumers selektiv Messages auswählen.

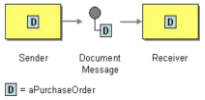
Enterprise Integration Patterns (EIP)

Application Integration Criteria: Application coupling, Data format, Asynchronicity

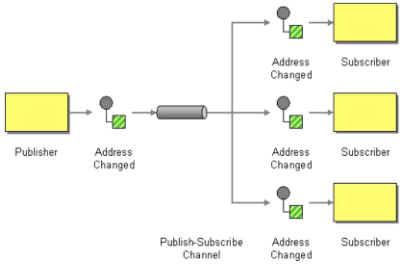
Point-to-Point Pattern



Nachricht wird in 5 Schritten übermittelt: 1. Create: Create message, 2.Send: add to channel, 3.Deliver: system moves message to receivers PC, 4.Receive: take message from channel, 5.Process: extract data.
Nachrichten können Ablaufdatum besitzen.
Document Messages: reine Datenübermittlung (z.b.Order)
Command Message: tells the receiver to invoke certain behavior (z.B. getLastOrder())



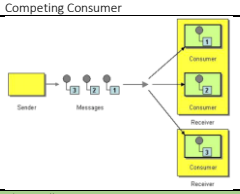
Publish-Subscribe Pattern



When an event is published into the channel, the Publish-Subscribe Channel delivers a copy of the message to each of the output channels (receivers).
topic-based: messages are published to topics/named logical channels. Subscribers will receive all messages published to the topic to which they subscribe. publisher is responsible of defining the topic of their messages.
content-based: messages are only delivered if attributes match constraints defined by subscriber. The subscriber specifies the (set of) attributes that characterize the messages of interest.

Receiving Message Endpoint

Competing Consumer



Selective Consume

Konsument behandelt bestimmte Meldungen

RPC: Remote Procedure Call

Der Client muss sich nicht darum kümmern, ob die Prozedur lokal oder remote ausgeführt wird. Wärend der Server einen Request verarbeitet, ist der Client typischerweise blockiert. (synchroner Aufruf) Der Client muss mit Netzwerk Ausfällen umgehen können. RPC Prozeduren sollten idempotent sein!

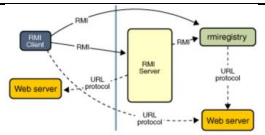
1. Client ruft Client Stub auf 2. Client Stub verpackt die Parameter in eine Message (Marshalling) 3. Das lokale OS versendet die Message an den Server 4. Das Server OS leitet die eingehende Message an den Server Stub (Skeleton) 5. Der Server packt die Parameter aus (unmarshalling) 6. Die Methode wird mit den übermittelten Parameter aufgerufen.

RMI: Remote Method Invocation (Java OO-RPC)

RMI ist der objektorientierte Ansatz von Java für RPC. Es erlaubt das übermitteln von komplexen Objekten. RMI bietet kein Timeout Management. (Aufgabe des Clients)

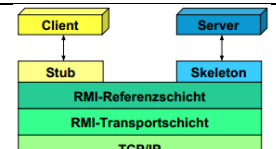
RMI Registy ist das Telefonbuch und beschreibt wo sich ein Objekt befindet und wie es angesprochen werden kann.

1. Der Server registriert ein Remote Objekt in der Registry
2. Der Client such nach dem Objekt in der Registry
3. RMI benutzt Webserver für das Verteilen von Klasseninformationen



Stub und Skeleton implementieren das selbe Interface !

1. Ein Stub ist ein Stellvertreterobjekt (Remote Proxy), das den Clientaufruf an den Server weiterreicht
2. Ein Skeleton nimmt die Aufrufe des Stubs entgegen und leitet sie an das Serverobjekt weiter
3. Die RMI Referenzschicht stellt den Nameservice (=Registry) zur Verfügung
4. Die RMI Transportschicht verwaltet die Verbindungen und wickelt die Kommunikation ab.



Vorteile von RMI gegenüber Sockets: - Garbage Collector, direkter Methodenaufruf, nicht nur Verbindungsaufbau
Nachteile von RMI gegenüber Sockets: - proprietäre Lösung, - grosser Overhead, - kein Timeout-Handling, - Enge Client/Server Bindung

```
import java.rmi.Naming;  import java.rmi.RemoteException;  import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.*;
public class RmiServer extends UnicastRemoteObject implements RmiServerIntf {
  public static final String MESSAGE = "Hello World";
  public RmiServer() throws RemoteException {
    super(0);   //required to avoid the 'rmic' step, see below }
  public String getMessage() {
    return MESSAGE;    }
  public static void main(String args[]) throws Exception {
    System.out.println("RMI server started");
    try { //special exception handler for registry creation
      LocateRegistry.createRegistry(1099);
      System.out.println("java RMI registry created.");
    } catch (RemoteException e) {
      //do nothing, error means registry already exists
      System.out.println("java RMI registry already exists.");
    }
    //Instantiate RmiServer
    RmiServer obj = new RmiServer();
    // Bind this object instance to the name "RmiServer"
    Naming.rebind("//localhost/RmiServer", obj);
    System.out.println("PeerServer bound in registry");   } }
```

```
import java.rmi.Remote;  import java.rmi.RemoteException;
public interface RmiServerIntf extends Remote {
  public String getMessage() throws RemoteException;    }
```

```
import java.rmi.Naming;
public class RmiClient {
  public static void main(String args[]) throws Exception {
    RmiServerIntf obj = (RmiServerIntf)Naming.lookup("//localhost/RmiServer");
    System.out.println(obj.getMessage());    } }
```

IDL: Interface Definiton Language

IDL ist eine Spezifikationsprache für sprachunabhängige Interfaces. IDL wird für die Kommunikation bei RPC verwendet.

WSDL : Web Services Description Language

WSDL ist eine XML-basierte Interface Spezifikationsprache für Web Services.

RPC Style SOAP Message	Klare Vorgabe, wie der SOAP Body auszusehen hat
Document Style SOAP Message	Keine Vorgaben wie der SOAP Body auszusehen hat (kein Standard)

Operationale Modelle/NFR(wie?)

Nicht-Funktionale Anforderungen in zwei Kategorien geteilt:

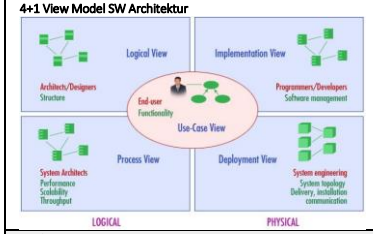
Qualities (Eigenschaften / Charakteristik)	Constraints (Einschränkungen)
Runtime(Service Level Requ., messbar): zB Performance/Capacity, Reliability, Efficiency, Availability, Manageability, Security, Usability, Data Integrity Non-Runtime: Portability, Scalability, Maintainability	Business: Regulatory, Organisational, Risk Willingness, Market-place factors, Schedule/Budget Technical: Legacy Integration, Development Skills, Existing Infrastructure, Technology State o.t. Art, IT Standards

NFR Beispiele: ***Efficiency***: Responsezeiten unter 1s spezifiziert; ***Interoperability***: Multiplattform Support
Accuracy: Bestellungen dürfen nicht verloren gehen und Reservationen müssen rückgängig gemacht werden können
Modifiability: Know-how für ausgewählte Technologien müssen vor Ort vorhanden sein

Alle Faktoren müssen mit Kosten in Balance sein. Gefahr von Scheitern auf einer Seite, Over-Engineering auf anderer!
Beispiel Design-Tradeoff: Security vs. Performance → TLS dafür Responseeinbussen oder ressourcenintensive Antivirensw

IT-System auf mehreren Knoten verteilt und besteht aus Komponenten. Komponenten befinden sich auf verteilten Knoten. Das **Component Model** zeigt die *Funktionalität* eines Systems, könnte auch Application Model heissen.
Operational Model (-> Knoten) zeigt Infrastruktur auf, Alias Architectural Model.

4+1 View Model SW Architektur

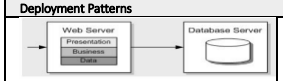


Component Model: Bestandteile eines IT-System, Zuständigkeiten für Funktionalität in Komponenten unterteilt. Verbindung zw Requirements & Lösung

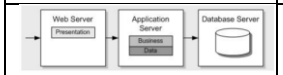
Operational Model: Logischer Viewport, geographische Struktur der Orten & Grenzen des IT-System. Enthält Verbindungen zw Knoten. Organisation wird in Zonen eingeteilt

Deployment Units: Verbindungsglied zw Operational und Component Model, ~Deployment Diagramm!

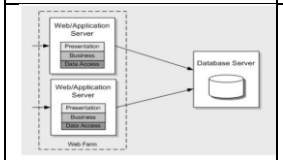
Deployment Patterns



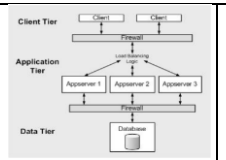
Non-Distributed Deployment: Alle Layers und Funktionlität mit Ausnahme der Data Storage auf 1 Server.
Vorteil: Simplizität, wenige Server
Nachteil: Falls 1 Layer ausgelastet, andere ebenso betroffen



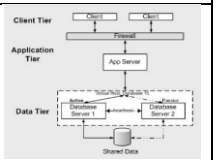
Distributed Deployment: Layer auf separaten Tier.
Vorteil: zwischen Layers zB Firewall möglich. Application Server genau für Requirements konfigurierbar. Einfach skalierbares System
Nachteil: Komplexer/Aufwendiger/teurer



Web Farm: *Performance Pattern.* Collection von Servern wo dieselbe Applikation läuft. Jedem dieser Server wird etwa gleiche Anzahl an Anfragen weitergeleitet.



Load Balancing Cluster: Last der Anfragen auf mehrere Application Server teilen. Auch hier Firewall zw Tiers möglich



Failover Cluster: *Reliability Pattern.* Bei Ausfall übernimmt automatisch Failover(passiver Server mit gleichem Stand). Mit Heartbeat kann Ausfall entdeckt werden

Design for Performance **WSK: Wahrscheinlichkeit**

Performance: Ausmass, an welchem System/Komponente seine Funktion mit Einschränkungen wie Speed, Genauigkeit, Speicherbedarf ausführen kann. Drei hauptsächliche und voneinander abhängenden Aspekte: **Antwortzeiten, Durchsatz, Kapazität**
--> Kapazität nötig für guten Durchsatz, guter Durchsatz nötig für kurze Antwortzeiten:

Bei hoher Last steigen die Responsezeiten fast **exponentiell**! Anfragen benötigen unterschiedlich viele Resourcen, gibt rechenintensive Prozesse, je mehr Anfragen desto höher WSK dass langer Prozess dabei ist, der längere Wartezeit verursacht. Das wiederum erhöht WSK(mehr Anfragen in dieser Zeit) dass erneut langer Prozess dabei ist, führt zu Auslastung des Servers

Scale out: Einsatz von mehreren Servern, Skalierung horizontal, Erweiterung der Infrastruktur mit Load-Balancern, etc.

Scale up: Verbesserung der bestehenden Komponenten, bessere Hardware (CPU, RAM, NIC, etc)

E2E Response Time: Summe der Responsezeiten aller Komponenten

Schneller nicht immer genug: Real-time System verlangt auch extreme **Konsistenz**!

Scalability

Scalability is the capability to endure increasing workloads without decreasing agreed service levels if underlying resources are also increased. Types of Scalability:
Size Scalability →Faster/More Components, **Generation Scalability** → Better Components due to technological innovation,
Vertical Scalability (Sclae up) → increase performance of individual node, **Horizontal Scalability (Scale Out)** → Increase number of nodes (App sollte nicht zustandsbehaftet sein)

Load Balancer → bsp: nginx

provides site selection, workload management, session affinity, and transparent failover. intercepts data requests from clients and forwards each request to the server that is currently best able to fill the request. High availybility can be achieved by installing a backup load balancer. Load Balancer has a single virtual IP address. App Servers have own physical IP addresses. Scheduling Tactics: Round Robin, Least-recently-used, workload-based

Session affinity overrides the load-balancing algorithm by directing all requests in a session to a specific application server. (sometimes necessary for app to work correctly) It uses cookies to track session information by adding routing information to the cookie.

Availability

Availability = Lack of failures which are visible to stakeholders

A system is called available if it is up and running and produces correct results, meeting other NFRs, e.g. response time

Key Availability Terms MTTR + MTTF = MTBF

Mean Time to Recover (**MTTR**) = von Fehler bis zu Reparatur (AVG)
Mean Time to Failure (**MTTF**) = von Reparatur bis Fehler
Mean Time between Failure (**MTBF**) = von Fehler bis Fehler

RTO: Recovery Time Objective) time within system must be restored (MAX)
RPO: Recovery Point Objective maximum tolerable period in which data might be lost after failure

Effect on Availability

Using components in series → Each component relies on the previous component (fällt einer aus = No service)
→ total av is lower than that of weakest link: A = A1 * A2* A3
Using components in parallel → Component redundancy through duplication (fällt einer aus = Reduzierter SErvice)
→ Total av is higher than that of the individual links A = 1 – [(1 – A1) * (1 – A2) * (1 – A3)]

24x7 availability is (almost) impossible – getting close is costly

Availability numbers:

Availability %	Downtime per year	Downtime per month	Downtime per week
90% ("one nine")	36.5 days	72 hours	16.8 hours
95%	18.25 days	36 hours	8.4 hours
97%	10.96 days	21.6 hours	5.04 hours
98%	7.30 days	14.4 hours	3.36 hours
99% ("two nines")	3.65 days	7.20 hours	1.68 hours
99.5%	1.83 days	3.60 hours	50.4 minutes
99.8%	17.52 hours	86.23 minutes	20.16 minutes
99.9% ("three nines")	8.76 hours	43.8 minutes	10.1 minutes
99.95%	4.38 hours	21.56 minutes	5.04 minutes
99.99% ("four nines")	52.56 minutes	4.32 minutes	1.01 minutes
99.999% ("five nines")	5.26 minutes	25.9 seconds	6.05 seconds
99.9999% ("six nines")	31.5 seconds	2.59 seconds	0.605 seconds
99.99999% ("seven nines")	3.15 seconds	0.259 seconds	0.0605 seconds

Branch	Cost of Downtime / Hour
Manufacturing	28.000
Logistics	90.000
Retail	90.000
Home Shopping	113.000
Media („pay per view")	1.100.000
Bank (back office)	2.500.000
Credit Card Processing	2.600.000
Stock Brokerage	6.500.000

Circuit Breaker Pattern: Erkenne und melde Fehler schnell (Fail Fast). Failure Threshold für Services/Funktionen werden monitored und beim Überschreiten der Threshold wird der Circuit unterbrochen -> Alerts/Events senden und darauf reagieren

Techniques to improve availybility — CFIA: Component Failure Impact Analysis

Avoid single Points of Failure → CFIA, potential SPOF: Directory Server, rarely used Component with special dependency, …
Use redundancy (Clusters, Backups) Detect failures as fast as possible → Fail Fast!

Redundancy

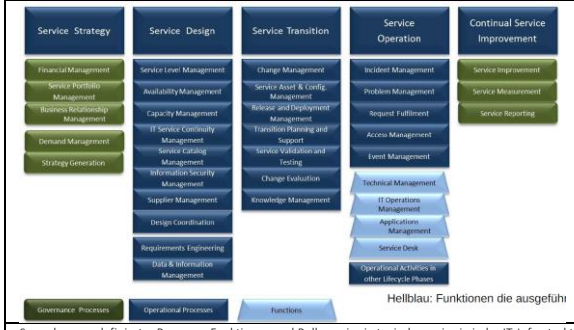
Hot Standby	Alle laufen und sind aktiv. Secondary führt Transaktionen nach
Warm Standby	Alle laufen und nur einer ist aktiv
Cold Standby	Immer nur einer aktiv

Systems Management

Systems Management umfasst: Software distribution and upgrading, version control, virus protection, user profile management, backup and recovery uvm.
Configuration Management: Dokumentation aller Komponenten eines Systems, Build Recreation für Maintainability, Zugriffsrechte.
Sind da um NFR(uA. **Auditability**(Nachvollziehbarkeit) zu erfüllen) Um Faktoren weiter zu überprüfen gibt es **ITIL:**

FCAPS: Fault, Configuration, Accounting/Administration, Performance, Security
Fault: Recognize, Isolate, Correct and Log Faults. **Configuration:** Monitor system configuration and other changes that take place.
Accounting: Collect statistics and bill based on them **Performance:** Ensure Acceptable performance **Security:** Auth. & Authorizate

ITIL 2011 (ITIL = IT Infrastructure Library)



Hellblau: Funktionen die ausgeführ

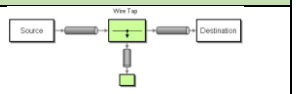
Sammlung vordefinierter Prozesse, Funktionen und Rollen, wie sie typischerweise in jeder IT-Infrastruktur mittlerer und großer Unternehmen vorkommen. ITIL beschreibt in fünf Kernbänden mit derzeit 37 Kernprozessen die Komponenten und Abläufe des Lebenszyklus von IT-Services.

Eine Kernanforderung an die Prozesse ist dabei die Messbarkeit.
Es handelt sich dabei lediglich um Best-Practice-Vorschläge, die an die Bedürfnisse des Unternehmens angepasst werden müssen.

Systems Management Patterns to monitor and control message-/queueing-based distributed software systems

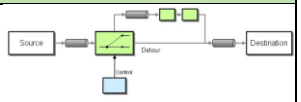
Wire Tap

- Inspect Messages that travel on a point to point channel
- Consume messages off the input channel and publish them to the destination and a separate inspection channel
- The Wire Tap is generic and reusable (is its own component)



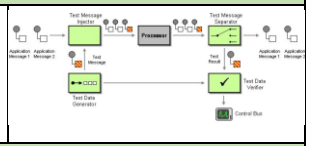
Detour

- Route a message through intermediate steps to perform validation, testing or debugging
- Context based router: One state routes directly to the destination, while another takes additional steps (→detour)
- Components in the detour can inspect/modify the message
- In the end all the messages have the same endpoint



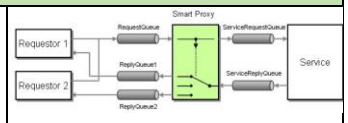
Test Message

- Assure the health of a message processing components
- Identify message processing components that are actively processing messages, but garble outgoing messages due to an internal fault



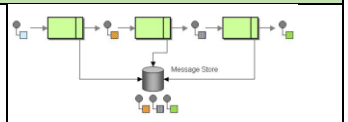
Smart Proxy

- Track/inspect messages to/from a service that sends reply messages to the Return Address specified by the Requestor
- intercepts messages sent to the service
- stores the return address specified by the original sender
- replaces the return address in the message with its own address
- on the reply message, retrieves the stored return address and forwards the unmodified reply address to the original requestor



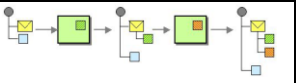
Message Store

- capture information about each message in a central location
- have each component send duplicates of each message to the message store
- analogy: Einschreibe-Brief



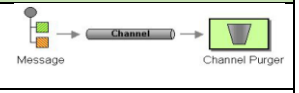
Message History

-create / maintain list of all applications that the message passed through (for analyzing/debugging the message-flow)
- every component that processes the message adds one entry to the list
- the message history is kept in the message header (separate from the body)



Channel Purger

- Eliminate 'left-over' messages(from previous tests)
- reset the system into a consistent state
- remove messages based on specific criteria (eg ID)



Logging Requirements	- easy to use - centralized	- no impact on performance - filtering capabilites

Log content should support developers during: **root cause analysis, event storms (high number of events), event correlation**

Log Levels

Trace	Debugging während Entwicklung	**Debug**	Debugging während Produktion (z.B. eingehender HTTP Request)
Info	Normale Prozesse im System (User X hat Item Y bestellt, erfolgreicher HTTP Req.)	**Warning**	Vorgänge die evtl. näher betrachtet werden müssen (z.B. mehrmaliges falsches pw)
Error	Fehler die zum Abbruch des Vorgangs oder Runterfahren des Systems führen (z.B. Exceptions)		

CFIA: Component Failure Impact Analysis

Bei der CFIA gehts darum, vorbereitet zu sein, falls etwas schief läuft. Es ist immer anzunehmen, dass Fehler passieren. Hier geht es darum, wie damit umgegangen wird, um den Impact möglichst klein zu halten. (proaktiver Ansatz)

1. Kritische Bereiche identifizieren 2. Daten sammeln um die Wahrscheinlichkeit eines ausfalls zu verkleinern 3. Dokumentieren, sodass die Zeit für die Reparatur möglichst klein ist

Workflow : Operational Model für Key Use Cases → CFIA → Risiko Logs → Technischer Fahrplan (Roadmap)

Oft resultiert eine Roadmap die in 3 Phasen unterteilt ist.
Phase 1 : Verbesserung welche direkt umgesetzt werden müssen (innerhalb von wenigen Wochen)
Phase 2 : Wichtige Verbesserungen die innerhalb von einigen Monaten umgesetzt werden müssen
Phase 3 : Verbesserungen mit tiefer Priorität die fundamentale Änderungen an der Architektur zur Folge haben.

Wichtigste Probleme, die mit CFIA gelöst werden können:

- Identification of Single Points of Failure (SPOF), where loss of a single component would impact on the non-functional character-istics of an IT service.
- Missing or inadequately documented architecture for the IT service and operational procedures.
- Monitoring may be missing or otherwise deficient, resulting a component or service outage not being detected (eg. No monitoring of overall IT service availability).
- Deficiencies in backup and restoration procedures that may impede recovery operations (eg. Data being stored on lots of separate tapes or backup data not being held securely).
- Processes and procedures to recover from a failure (or failover) are missing or deficient. Manual intervention means much more significant delays in recovery.
- Adequacy of backup and restore procedures
- "Key Person" dependencies – where a single individual is responsible for technical support or operations for one or more components essential to the successful operation of the IT service.

CFIA Matrix : Auflisten aller Geräte und Services (Business Critical Use Cases) in einer Matrix
→Ausfüllen: Auswirkungen eines Fehlers bei diesem Gerät auf den entsprechenden Service
leer = keine Auswirkungen, X = inoperativ, A=alternative available, M=alternative but manual intervention

CI	Service 1	Service 2
PC1	M	M
PC2	M	M
Cable 1	M	M

CFIA – Risk Log

Document key findings for each CI (Node, Component, Link, etc.) to support prioritization of risks, and selection of the risks for which a solution will be proposed

- *Risks that are obvious from the systems architecture prior to any formal walkthrough.*
- *Risks that arise from the system-level CFIA table*
- *Risks that arise from the detailed 'node' analysis.*
- *Risks that arise from previous problem records, and RCAs*

RCA = Root Cause Analysis

- **Resilience risks:** Risks which may cause a service to become unavailable in the first place. They are fundamental weaknesses.
  - Single point of failure in the IT infrastructure
  - No hot failover capability
  - Log files filling up
  - Bugs in code
  - Operator error
  - Old hardware
- **Recovery risks**: Risks which prevent the service from being recovered from an outage in a timely manner.
  - Responsibilities not defined clearly
  - Lack of, or incomplete recovery procedures
  - Lack of skills
  - Over-complex manual tasks with no automation
  - Recovery process is not tested
- **Security Risks:** Risks which can render a service to become useless, for example, through:
  - Security patch management
  - Privileged users who misuse their privileges
  - Denial of service attacks

Architectural Significant Use Cases → Use Case, bei dem alle Komponenten benötigt werden

Teil Mehta

Begriffe

NTP	Network Time Protocol: Standard zur Synchronisierung von Uhren in Computersystemen über paketbasierte Kommunikationsnetze. NTP verwendet das verbindungslose Transportprotokoll UDP.

Naming

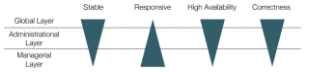
Wieso Naming? Muss Ressource die mich interessiert finden können
Naming: Namensgebung an Entitäten
Namensauflösung: Wie eine Ressource zugegriffen werden kann mithilfe eines Namens
Name: Sequenz von Bytes/Characters, die für das Referenzieren einer Entität gebraucht werden
Entität: Anything that can be operated on. Ein Objekt, das mir eine Ressource bietet
Access Point: Spezieller Typ einer Entität der für das Zugreifen auf eine andere Enität benutzt wird
Address: Name eines Access Points
Location Independent Name: Name, der unabhängig von der Adresse ist (IP, FQDN)
Identifier: Identifiziert genau eine Entität → eine Entität hat genau einen Identifier, wird nicht wiederverwendet

System	Name	Address	Access Point	Entity
Post	Recipient's name	City + ZIP Code + Street + Number	Mailbox	Recipient (Person)
File System	Path	File Handle	Disk Drive	File (String of bytes)
DNS	Fully Qualified Domain Name	IP	Network Interface	Host
ARP	IP	MAC	Network Interface	Host

mDNS wie DNS, aber Name: «.local»-Domain mit lokaler Signifikanz

Flat Naming: Namen als unstrukturierte Identifier, können interne Struktur haben welche aber nicht für Namensauflösung gebraucht wird. zB Broadcasting (ARP)
Vor- und Nachteile:
+ self managed (keine master nodes), einfache Implementation, location independend
- broadcasting does not scale and is easy to exploit

Structured Naming: Namen sind hierarchisch strukturiert (split into layers) -> unterricht.hsr.ch Namensauflösung kann in mehrere Schritte/Verantwortlichkeiten gesplittet werden, zB DNS:



Am Beispiel DNS (Verwendet Kombination von beiden)**:**
Iterative Resolution: Client muss für jede Domain separate Abfrage starten.
Vorteil: Entlastung der Nameservers
Nachteil: Aufwendig für Client, Hohe Latenzzeiten, weniger Caching möglich
Recursive Resolution: Client setzt eine Abfrage für gesamte URL ab und bekommt IP zurück.
Vorteil: besseres Caching möglich, einfach für Client.
Nachteil: hohe Performanceanforderungen für Nameserver

Iterative Resolution macht Sinn wenn: Der Benutzer viel processing power hat und Security wichtig ist
Recursive Resolution macht Sinn wenn: Caching gefordert ist

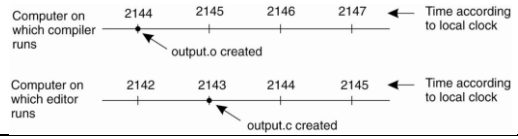
Attribute Based Naming. Ziel: Benutzer sucht Entitäten mittels Attributen. Die Entitäten sind in Form von Key-Value Paaren gespeichert (Adresse <-> Entity). Bsp: LDAP (Lightweight Directory Access Protocol (Mix Att.based/Structured))

Bitcoin Naming: Flat Naming, Entity = Person, Name = Person name, Access point = Einträge in Blockchain, Address = Public Key

Synchronization

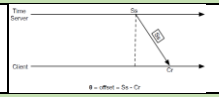
Synchronising Physical Clocks

- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
- Plays havoc with systems like "make"



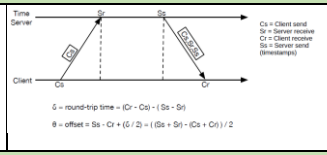
One-sided update

- Master clock periodically notifies slaves about current time
- Problem: Network latency affects accuracy



Network Time Protocol (NTP)

- Client adjusts its clock by slowing down or speeding up its clock interrupt interval
- Why not just directly add or subtract the offset from the current time?
→2 Messages may have the same timestamp as a result



Synchronizing Logical Clocks

Updates on a replicated database

Solution 0:
The node that wishes the update, updates its copy of the database and sends the desired update to all its peers.

Problems:
The update may be lost, receiver doesn't know that he is requested to perform an update.
The order in which updates are performed may be different at different nodes.

Solution 1:
All nodes must acknowledge the receipt of all updates to all other nodes. Each node maintains a set of pending updates. A node performs an update only if the update has been acknowledged by all its peers.

Problem:
It can be, that the order in which updates are performed is different at different nodes.

Solution 2:
Each node timestamps all messages it sends with the value of its physical clock. Pending updates are stored in a priority queue (priority = timestamp). A node can only perform updates in the order they are present in the queue.

Problem:
Physical clocks may not be correctly synchronised. Even with
reasonable synchronisation, it can still be the case, that two updates have the same timestamp.

→ It is often more important that processes agree on the order of events rather than the exact point in time (physical clock) at which an event happens.

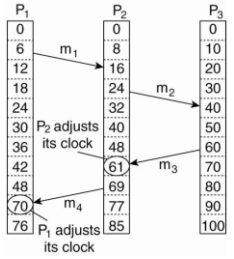
Causality

a → b	- Event a happens before event b	a → b if at least one of the following conditions are true - a and b are in the same process and a occurs before b - a is the event of sending a message and b is the event of receiving the same message - ∃ x. a → x ∧ x → b (transitive closure)
a ∥ b	- Event a and event b are concurrent	a ∥ b ⇔ (a ↛ b ∧ b ↛ a) Questions: Do concurrent events have to occur «simultaneously»(have the same timestamp)? Are all simultaneous events concurrent? →No to both: concurrent != simultaneous, simultaneous = same physical timestamp

Causal ordering doesn't solve all the problems in ordering events. There still needs to be a way to order concurrent events.

Lamports Logical Clock

- Each process P_i maintains an internal counter $C_i(a)$ that assigns a number to all events a occurring in process i.
- Each process P_i increases $C_i(.)$ between any two successive events.
- The current value of the counter is sent with each message.
- If the event a is "sending m from P_i" and the event b is "receiving m on P_j", then
  $C_j(b)$ is set to (max($C_i(a)$, $C_j(b)$) +1)



Note:
- C.(.) preserves causality:
  a → b ⇒ C.(a) < C.(b)
  Examples?
- Converse not true:
  C.(a) < C.(b) ⇏ a → b
  Examples? 48 < 50; event at 50 didnt happen before event at 48

Q. Is C.(.) totally ordered?

A. Not yet. We need to introduce a fixed, but arbitrary order between event on different processes with the same counter value to make it a strict total order. Idea: use the process ID to order concurrent events.

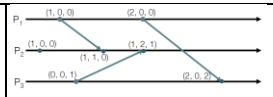
- Each process P_i maintains an internal counter $C_i(a)$ that assigns a number to all events a occurring in process i.
- The (distributed) function C(.) assigns the tuple C(a) to any event a, where C(a) = (C_i(a), i) where a is an event in process i.
- The (overloaded) lexicographic ordering "<" is used for these tuples.
- Properties of C(.):
  - "C(.) < C(.)" is a *strict total order* (think: linked list) over events:
    - The relation C(a) < C(b) is a strict partial order over events a & b. (property of the lexicographic ordering "<")
    - Additionally comparable: (C(a) < C(b)) xor (C(a) > C(b)) xor (a = b)
  - C(.) preserves causality: a → b ⇒ C(a) < C(b)

Result: With C(.) we have a global (distributed) counter (timestamp) that is total and preserves causality!

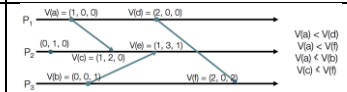
Vector Clocks

Since Lamport's Clock defines a total order on events, it cannot capture causality exactly:
a → b ⇔ C(a) < C(b)
• In some cases it is useful to know that two events are not causally related. For example:
• To detect conflicts amongst distributed updates
• Scheduling and debugging concurrent computations
• Vector clocks track causality between events exactly:
a → b ⇔ V(a) < V(b)

Each process Pi maintains an internal vector Vi with the following properties:
1. Vi[i] is the number of events that have occurred so far at Pi.
2. If Vi[j] = k then Pi knows that k events have occurred at Pj.

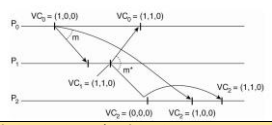


Vector clock properties are maintained with the following rules:
1. Increment Vi[i] with the occurrence of each new event at process Pi.
2. Receiving a message at Pj from Pi causes Pj to additionally update each entry Vj[k] to max(Vi[k], Vj[k]).



Vector Clocks
Example application: Enforcing causal communication

Requirement: A message is delivered to the application only if all messages that causally precede it have also been delivered.

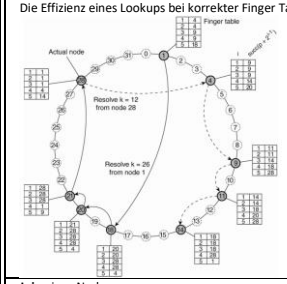


Distributed Hashtables (DHT) werden oft als redundanter / ausfallssicherer Datenspeicher eingesetzt.

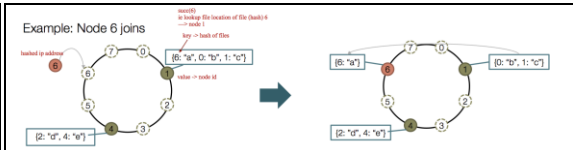
Chord ist eine Consistent Hashing implementierung für P2P Datenspeicherung → Ermöglicht die Zuordnung von Keys zu Nodes, ohne grosse Änderungen bei neuen Nodes.
Hash im Bereich von 0 bis $2^m - 1$, mit *m* meistens 128 oder 160. Die Hashes werden in einem Ring angeordnet.
Hostadresse $n = hash\ (IP\ )$
Nachfolger $n.successor$ ist jedem Node für seinen Nachfolger bekannt (n.successor = first node clockwise from n in the ring).
Jeder Node ist für den Adressbereich bis zu seinem Vorgänger zuständig.
Die verteilte Funktion *succ( )* ist dafür da, den für einen Bereich zuständigen Node zu finden.
n.successor = succ(n + 1)
n.predecessor = first node counter-clockwise from n in the ring
Schlüssel: alle Objekt-Ids der Datensätze, für die ein Knoten zuständig ist
Finger Table: Ist eine «Shortcut» Tabelle, ähnlich Binary Search. Jeder Node *n* hat eine Finger Table mit Index $i \in \{1, \dots, m\}$
Die Tabelle wird mit $n.finger[i] = succ(n + 2^{i-1})$ befüllt.

Stabilisierung: Aktualisierung der Informationen im Node über das Netzwerk
- Der direkte Nachfolger eines Nodes wird regelmässig Überprüft: (es gilt immer: $n < x \leq n.successor$)
  x += (n.successor).predecessor;
  if (n < x < n.successor) { n.successor := x }
- Der direkte Nachfolger wird regelmässige über die Existenz informiert. Beim erhalt der Nachricht von *p* macht Node *n*:
  if (p.predecessor is undefined OR n.predecessor < p < n) { n.predecessor := p }
- Einträge der Finger Table werden auf einem Node *n* in regelmässigen Abständen zufällig aktualisiert:
  i := random integer >1 and ≤m
  finger[i] = n.successor.lookup(n + 2^{i-1}, n)

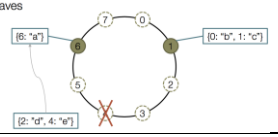
n.lookup(k,a): Zuständigen Node für Hash/Key *k* von Node *n* aus gesehen finden und Ergebnis an Node *a* senden:
1. Falls *n.predecessor < k ≤ n* ist der Node *n* zuständig.
2. Falls der Nachfolger von *n* Zuständig ist *n < k ≤ n.successor*, die Suchanfrage an diesen Node weiterleiten: *n.successor.lookup(k,a)*
3. Andernfalls wird die Anfrage an den nächsten Vorgänger *q* von *k* aus der Finger Table angefragt ($q \leq k$): *q.lookup(k,a)*
Die Effizienz eines Lookups bei korrekter Finger Table ist $O(\log N)$. (Ohne Fingertable → $O(N)$).



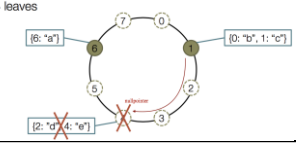
Join eines Nodes *n*:
1. Bestimmen der Hostadresse *n = hash(IP)*
2. Bestimmen des Nachfolgers *n.successor := succ(n+1)* und des Vorgängers *n.predecessor := succ(n-1)*
3. Stabilisierung
   a. Informieren des Nachfolgers *s* → *s.predecessor = n*
   b. Vorgänger *p* fragt bei *s* periodisch nach, ob ein neuer Node da ist, und aktualisiert seinen *p.successor = n*
   c. *p* informiert *n*, dass er sein Vorgänger ist → *n.predecessor = p*
4. Stabilisierung der Finger Tables



Leave eines Nodes *n* **geplant**:
1. Alle Ressourcen werden an *n.successor* übergeben
2. n.predecessor (=p) wird darauf hingewiesen, seinen *p.successor* anzupassen.
3. *n.successor (=s)* wird darauf hingewiesen, seinen *s.predecessor* anzupassen.
Example: Node 4 leaves



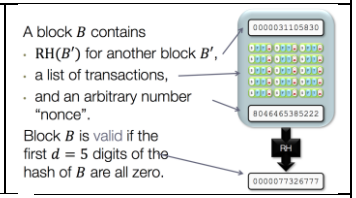
Leave eines Nodes *n* **ungeplant**:
1. Vorgänger *p* wechselt seinen Nachfolger zu *s* → *p.successor = s*
2. Vorgänger *p* informiert *s* über die änderung → *s.predecessor = p*
3. Stabilisierung der Finger Tables
Ungeplante Leaves führen zu Datenverlust, wenn keine Redundanz gespeichert wird.
Example: Node 4 leaves

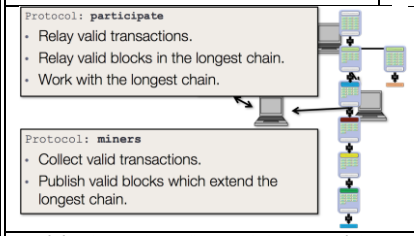


Bitcoin

Transaktion: Betrag, AbsenderPubKey, EmpfängerPubKey, Signatur mit PrivKey → an die Keys sind keine Namen gebunden (Privacy)

Blockchain: Transaktionen werden in Blöcke und danach in eine Kette eingereit. Der nachfolgende Block beinhaltet den Hash des vorherigen Blocks.
Eine totale Ordnung wird mit dem Konsensus-Protokoll erreicht.





Proof of Work: Damit das erstellen eines Blocks nicht zu einfach ist, muss der Hash jedes Blocks mit einer Anzahl Nullen beginnen. Dies kann erreicht werden, indem die **Nonce** eines Blockes geändert wird. Ein Miner probiert jeweils so viele Nonces durch, bis er so einen Hash erreicht hat. Danach wird der valide Block im Bitcoin-Netzwerk verteilt und dessen Hash in den nächsten Block einbezogen. (Die Anzahl Nullen wählt Bitcoin so, dass im Schnitt etwa 10 Min. daran gerechnet werden muss.) Als Belohnung erhält ein Miner, welcher diesen Hash findet, eine optionale **Transaction-Fee** sowie «Initial Reward» von 25BTC. Transaction-Fees sind Transaktionsbeträge ohne expliziten Empfänger.

Um **Double Spending** zu verhindern, braucht es Konsensus-Protokoll. Bitcoin nimmt jeweils den längsten gültigen Baum als wahr an. Da es bei jeder Blockchain-Verkettung schwieriger wird, die vergangenen Transaktionen zu fälschen (da ein längerer Baum berechnet werden müsste), reichen ca. 7 Blöcke, damit eine Transaktion sicher durchgelaufen ist.

Google

- Hardware failure
- Network cuts
- Traffic spikes/shifts
- Cascading overload
- Hotspotting
- Software bugs

- Monitoring
- Redundancy & Redundancy
- Avoid single points of failures
- Load balancing
- Throttling & rate limiting
- Backups

OSI Layer

Layer 7	Application	Communiontion Application
Layer 6	Presentation	
Layer 5	Session	SIP, RTP, RTCP
Layer 4	Transport	TCP, UDP, SCTP
Layer 3	Network	Ipv4, IPv6
Layer 2	Data Link	Ethernet, etc
Layer 1	Physical Link	Coax, RF link, etc