

# Grundlegende Konzepte verteilter Systeme

---

Dozent: Prof. Dr. Michael Eichberg  
Kontakt: michael.eichberg@dhbw.de  
Version: 1.0.1

---

Folien: <https://delors.github.io/ds-grundlegende-konzepte/folien.de.rst.html>  
<https://delors.github.io/ds-grundlegende-konzepte/folien.de.rst.html.pdf>  
Fehler melden: <https://github.com/Delors/delors.github.io/issues>

Die folgenden Konzepte sind für die Entwicklung verteilter Systeme von zentraler Bedeutung und sind in vielen aktuellen Middlewareprodukten umgesetzt.

# 1. Zeit in verteilten Systemen

---

# Von der Bedeutung der Zeit in verteilten Systemen

- Updates, die über mehrere Systeme hinweg erfolgen, müssen in korrekter Reihenfolge erfolgen.
- Log-Einträge sollen in korrekter Reihenfolge erfolgen.
- Gültigkeit von Berechtigungen (z. B. Zertifikate)
- geographische Positionsbestimmung (z. B. GPS)

## Probleme, wenn die Zeit nicht korrekt ist


*A recent surge in GPS “spoofing”, a form of digital attack which can send commercial airliners off course, has entered an intriguing new dimension, according to cybersecurity researchers: The ability to hack time. [...]*

*“We think too much about GPS being a source of position, but it's actually a source of time,” [...]*  
*“We're starting to see reports of the clocks on board airplanes during spoofing events start to do weird things.” In an interview with Reuters, Munro [at Defcon] cited a recent incident in which an aircraft operated by a major Western airline had its onboard clocks suddenly sent forward by years, causing the plane to lose access to its digitally-encrypted communication systems.*

*—11. August, 2024 - GPS spoofers 'hack time' on commercial airlines*

# Verteilte Systeme: Reale vs. logische Zeit

## Logische Zeit

Die logische Zeit ermöglicht es uns, eine wohldefinierte Reihenfolge zwischen Ereignissen (vgl.  *happened before* Relation) zu bestimmen. *Häufig* ist dies für verteilte Systeme ausreichend.

## Reale Zeit

**Sonnensekunde:** bezieht sich auf die Zeitspanne zwischen aufeinanderfolgenden Sonnenhöchstständen.

**Atomzeitsekunde:** Bezugspunkt ist die Schwingungsdauer eines Cäsium-133-Atoms.

TAI (Temps Atomique International): Durchschnittszeit der Atomuhren von über 60 Instituten weltweit (z. B. Braunschweig), ermittelt vom BIH (Bureau International de l'Heure) in Paris

**UTC (Universal Coordinated Time):**

Basiert auf TAI; aktuell ist noch das Einfügen gelegentlicher Schaltsekunden zur Anpassung an den Sonnentag erforderlich. Ab 2035 wird die Schaltsekunde voraussichtlich abgeschafft.

# Computeruhrzeit

- Real-time Clock (RTC): interne batteriegepufferte Uhr.  
(Die Genauigkeit und Auflösung sind teilweise sehr grob.)
- Funkuhr (DCF77 aus Mainflingen, ca. 2000 km Reichweite)
- GPS-Signal (Global Positioning System) mit einer Auflösung von ca. 100 ns
- mittels Nachrichtenaustausch mit einem Zeitserver

# Uhrensynchronisation nach Christian

(Probabilistic Clock Synchronisation, 1989)

- Voraussetzung: zentraler Zeitserver mit UTC.
- Clients fragen periodisch nach und korrigieren um halbe Antwortzeit
- Client-Uhren werden niemals zurückgesetzt sondern ggf. nur verlangsamt bzw. beschleunigt.



# Network Time Protocol (NTP, RFC 5905)

## ■ Synchronisierung auf UTC

im lokalen Netz mit einer Genauigkeit von bis zu 200 Mikrosekunden

im Internet mit einer Genauigkeit von 1-10 Millisekunden

## ■ Hierarchie von Zeitservern

Stratum 0: Quelle - z. B. DCF77-Zeitzeichensender

Stratum 1: Primärserver

Stratum 2,...: Sekundär-/...server

Clients

## ■ Wechselseitiger Austausch von Zeitstempeln zwischen den Server-Rechnern wird unterstützt (NTP ist symmetrisch).

---

Aktualisierung der Zeit eines NTP Servers erfolgt aber nur wenn der anfragende Server einen höheren *Stratum*wert hat (d. h. potentiell unpräziser ist) als der angefragte Server. Der anfragende Server erhält danach den Stratumwert des abgefragten Servers +1.

# Zeit: Berechnung der Round-Trip-Time und der Zeitdifferenz/des Gangunterschieds

Origin $T_1$	Systemzeit des Clients beim Absenden der Anfrage
Receive $T_2$	Systemzeit des Servers beim Empfang der Anfrage
Transmit $T_3$	Systemzeit des Servers beim Absenden der Antwort
Destination $T_4$	Systemzeit des Clients beim Empfang der Antwort

$$RTT : r = (T_4 - T_1) - (T_3 - T_2)$$

$$Gangunterschied : x = \frac{(T_2 - T_1) - (T_4 - T_3)}{2}$$

## Achtung!

Eine exakte Uhrensynchronisation ist in einem asynchronem System nicht realisierbar!

Es wird die Annahme getroffen, dass die Zeit auf beiden Rechnern quasi gleichschnell vergeht. Die Zeitdifferenz zwischen den beiden Rechnern ist also konstant.

$(T_3 - T_2)$  ist die Zeit, die der Server zum Bearbeiten benötigt.

Die Round-Trip-Time (RTT) ist die Zeit, die ein Signal benötigt, um von einem Rechner zum anderen und zurückzugelangen.

Der Gangunterschied ist die Differenz zwischen der Zeit auf dem Server und der Zeit auf dem Client.

Probleme bei der Uhrensynchronisation entstehen aufgrund ungewisser Latenzen:

- Nachrichtenübertragungszeit (abhängig von Entfernung und Medium)
- Zeitverzögerung in Routern bei Weitervermittlung (lastabhängig)
- Zeit bis zur Interrupt-Akzeptanz im Betriebssystem (kontextabhängig)
- Zeit zum Kopieren von Puffern (lastabhängig)

Aufgrund der Probleme ist ein konsistenter, realistischer globaler Schnappschuss nicht realisierbar.

## Beispiel zur Berechnung des Gangunterschieds

Sei die Latenz 5 ms und die Bearbeitungszeit 2 ms.

Weiterhin sei  $T_1 = 110$  und  $T_2 = 100$ . D. h. der Client geht vor.

Da die Bearbeitungszeit des Servers 2 ms beträgt, gilt für  $T_3$  und  $T_4$ :

$$T_3 = 102 \text{ und}$$

$$T_4 = 110 + (2 \times 5) + 2 = 122.$$

Somit ergibt sich der Gangunterschied zu:

$$x = \frac{(100-110)-(122-102)}{2} = \frac{(-10-20)}{2} = -15 \text{ ms.}$$

# Logische Zeit

Für die konsistente Sicht von Ereignissen in einem verteilten System ist die reale Zeit in vielen Fällen nicht wichtig!

Wir benötigen nur eine global eindeutige Reihenfolge der Ereignisse; d. h. wir benötigten Zeitstempel. Jedoch beeinflussen sich nicht alle Ereignisse untereinander; d. h. sind kausal unabhängig.

-----  
Es ist wichtig zu wissen, was vorher und was nachher passiert ist, aber es ist nicht wichtig, dass wir wissen wann genau (Uhrzeit) etwas passiert ist.

# Lamport-Uhren (*logical clocks*)

## Definition

Ein Ereignis (*write*, *send*, *receive*) ist eine Zustandsänderung in einem Prozess.

## Vorgehensweise

- vor *write* und *send*: erhöhen der lokalen Zeit  $T_{local} = T_{local} + 1$
- *send* immer inklusive Zeitstempel:  $T_{msg} = T_{local}$
- vor *receive*:  $T_{local} = \max(T_{msg}, T_{local}) + 1$

Ereignis *receive* ist zeitlich immer nach *send*.

Ereignisse werden eingeordnet nach der „happened-before“ Relation: **a** → **b**

(a happened-before b)

Resultat: es ergibt sich eine partielle Ordnung (partial ordering) der Ereignisse.

Ein konsistenter Schnappschuss enthält zu jedem Empfangs- das entsprechende Sendeereignis.

---

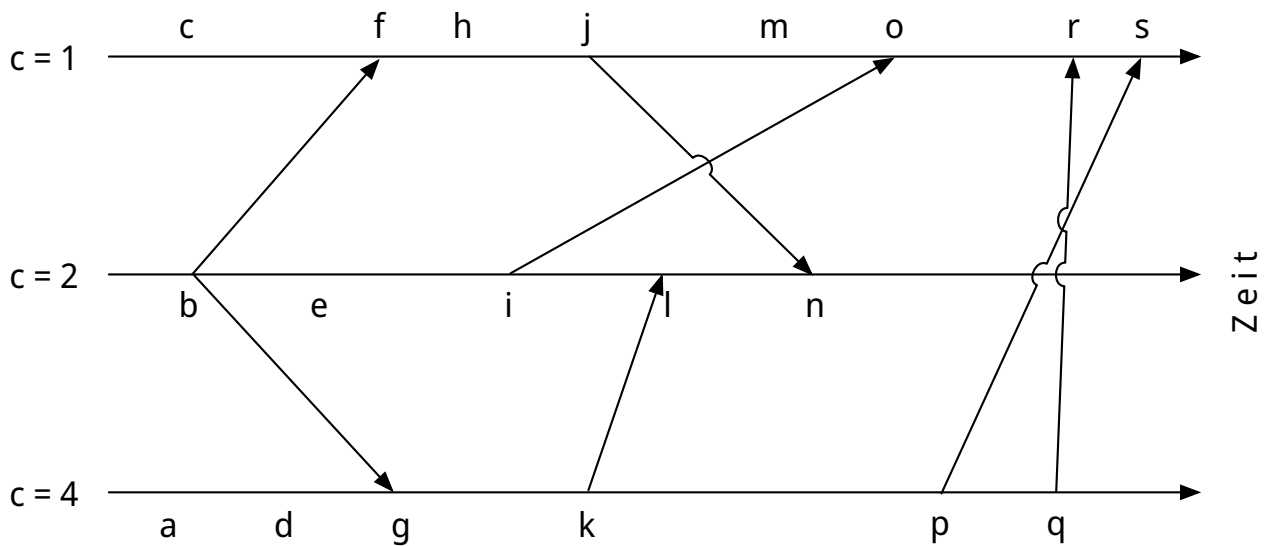
Lamport Uhren sind eine Möglichkeit, um *Totally-ordered Multicasts* zu unterstützen, was insbesondere im Zusammenhang mit Replication von Nöten ist.

# Übung

## 1.1. Lamport-Uhren

Gegeben sei die nachfolgend dargestellte Situation mit drei Prozessen in einem verteilten System. Die Zeitstempel der Ereignisse werden mittels der Lamport'schen Uhren vergeben. (Die Werte  $c$  ganz links, geben den Stand der jeweiligen Uhren zu Beginn an.)

- Versehen Sie alle Ereignisse mit den korrekten Zeitstempeln.
- Geben Sie einen konsistenten Sicherungspunkt an, der Ereignis  $r$  enthält.



## 2. Verteilte Transaktionen

# „Atomic Commit Protocol“

- Verteilte Transaktion erstrecken sich über mehrere Prozesse und meist auch über mehrere Knoten in einem verteilten System.
- Mehr Fehlerfälle müssen berücksichtigt werden.

Ein Beispiel wäre die Überweisung eines Geldbetrags (konzeptionelles Beispiel):

```
1 send_money(A, B, amount) {  
2   Begin_Transaction();  
3   if (A.balance - amount ≥ 0) {  
4     A.balance = A.balance - amount;  
5     B.balance = B.balance + amount;  
6     Commit_Transaction();  
7   } else {  
8     Abort_Transaction();  
9   } }
```

Wir brauchen ein *Atomic Commit Protocol*.

---

## Wiederholung: Transaktionen

Eine Transaktion stellt die zuverlässige Bearbeitung persistenter Daten sicher – auch in Fehlersituationen. Zentrales Merkmal ist die Garantie der ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability).

Am Ende einer Transaktion findet entweder ein commit oder abort / rollback statt.

Nach einem commit sind alle Änderungen dauerhaft.

## Fehlertoleranz

Das Ziel ist es zu ermöglichen, ein zuverlässiges System aus unzuverlässigen Komponenten aufzubauen.

Drei grundsätzliche Schritte:

1. Erkennung von Fehlern: Erkennen des Vorhandenseins eines Fehlers in einem Datenwert oder einem Steuersignal
2. Fehlereingrenzung: Begrenzung der Fehlerausbreitung
3. Maskierung von Fehlern: Entwicklung von Mechanismen, die sicherstellen, dass ein System trotz eines Fehlers korrekt funktioniert (und möglicherweise einen Fehler korrigiert)



# Two-Phase Commit Protocol - 2PC

Teilnehmer sind (1) die Partizipanten ( $P_i$ ), welche die verteilten Daten verwalten, und (2) ein Koordinator, ( $K$ ) der die Steuerung des Protokolls übernimmt. ( $K$  darf selbst einer der  $P_i$  sein)

## 1. Abstimmungsphase:

- $K$  sendet eine PREPARE-Nachricht an alle  $P_i$ .
- Jeder  $P_i$  prüft für sich, ob die Transaktion lokal korrekt abgeschlossen werden kann.
- Falls ja, sendet er READY, anderenfalls ABORT an  $K$

## 2. Entscheidungsphase:

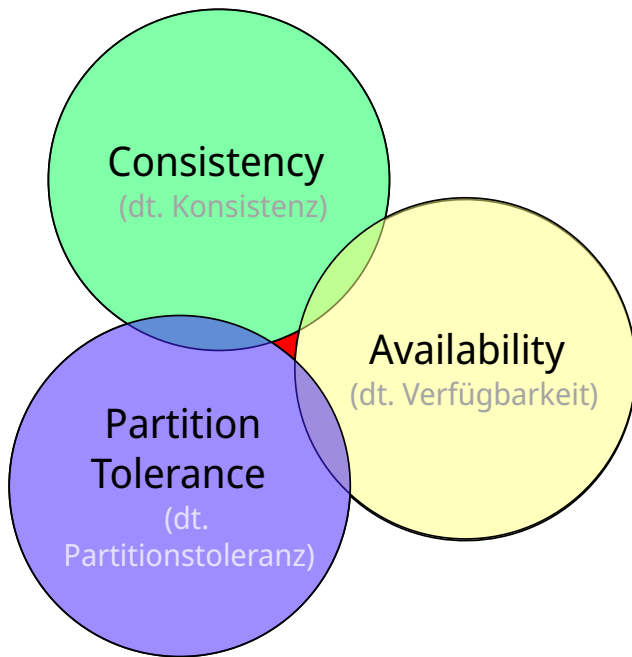
- Falls alle  $P_i$  mit READY geantwortet haben, sendet  $K$  COMMIT an alle  $P_i$ ; anderenfalls sendet  $K$  eine ABORT-Nachricht an alle  $P_i$
- Falls die Entscheidung COMMIT war, machen alle  $P_i$  die Transaktion *stabil*
- Falls die Entscheidung ABORT war, setzen alle  $P_i$  die Transaktion zurück.
- Alle  $P_i$  senden schließlich eine OK-Nachricht an  $K$

---

Das 2-PC Protokoll ist nicht Fehlerresistent. d. h. es kann Fehler erkennen, aber nicht zwangsläufig korrigieren. Um einige Fehlerszenarien zu behandeln, müssen Ergebnisse (insbesondere READY und COMMIT) in einem persistenten *write-ahead* Log-File festgehalten werden.

# CAP Theorem<sup>[1]</sup>

In **verteilten** (Datenbank-)Systemen können nur zwei der drei folgenden Eigenschaften gleichzeitig garantiert werden:



■ Konsistenz (Consistency)

(Nach Abschluss einer Transaktion ist der Rückgabewert der nächsten Leseoperation das Ergebnis der letzten Schreiboperation oder ein Fehler.)

■ Verfügbarkeit (Availability)

(Jede Anfrage erhält eine Antwort in akzeptabler Zeit.)

■ Partitionstoleranz (Partition Tolerance)

(Das System funktioniert auch bei Netzwerkpartitionierungen; d.h. Knoten können nicht mehr miteinander kommunizieren.)

---

Das CAP Theorem bezieht sich „nur“ auf verteilte Systeme. In solchen Systemen kann es immer zu Netzwerkpartitionierungen kommen. Deswegen ist Partitionstoleranz eine natürliche Eigenschaft und man kann häufig „nur“ zwischen Konsistenz und Verfügbarkeit wählen.

Welche Eigenschaften sind in welchen Szenarien wichtig?

DNS: Verfügbarkeit und Partitionstoleranz

Banking: Konsistenz und Partitionstoleranz

---

[1] 2000 Brewer(Vermutung), 2002 Gilbert und Lynch(Beweis)

# Übung

## 2.1. Two-Phase-Commit

Analysieren Sie, wie das Two-Phase-Commit-Protokoll mit Fehlersituationen umgeht.

Welche Fehler können zu welchen Zeitpunkten auftreten und welche kann das Protokoll beheben?