Verteilte Systeme – Übung

Verteilte Synchronisation

Sommersemester 2026

Paul Bergmann, Christian Berger

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Informatik 4 (Systemsoftware)
https://sys.cs.fau.de

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Überblick

Zeit in verteilten Systemen

- Echtzeit-basierte Uhren
- Logische Uhren

Synchronisation

Aufgabe 6

Zeit in verteilten Systemen

- Ist Ereignis A auf Knoten X passiert, bevor Ereignis B auf Knoten Y passiert ist?

  Beispiele: Internet-Auktionen, Industriesteuerungen, ...

- Prinzipiell keine konsistente Sicht auf Gesamtsystem möglich
  - Unabhängigkeit von Ereignissen
  - Informationsaustausch mit Latenzen verbunden

  $\Rightarrow$ Nur näherungsweise Lösungen möglich

- Bestes Verfahren abhängig von Einsatzgebiet und notwendigen Eigenschaften

Zeit in verteilten Systemen

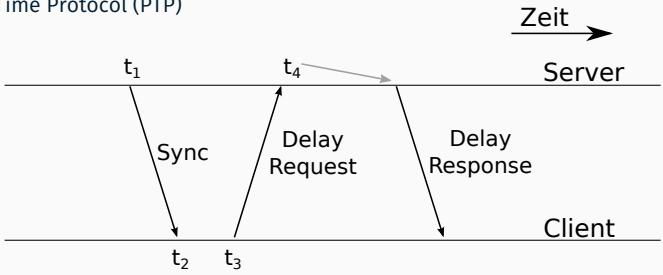
Echtzeit-basierte Uhren

Echtzeit-basierte Uhren

- Nutzung eines gemeinsamen Zeitsignals
  - Auflösung beschränkt
  - Schwierig über größere Entfernungen
    - → Ausbreitungsgeschwindigkeit: max. $30\,\mathrm{cm/ns}$
- Nachrichten mit Zeitstempel lokaler, physikalischer Uhren versehen
  - Wenig Kommunikationsaufwand
  - Ohne Synchronisation: Zunehmende Abweichungen
- Kombination verschiedener Verfahren zur Verbesserung der Genauigkeit

Synchronisation von Echtzeituhren: NTP, PTP

- Stellen lokaler Uhr basierend auf Referenzuhr
- In der Praxis verwendete Protokolle:
  - Network Time Protocol (NTP)
  - Precision Time Protocol (PTP)



- Berechnung von Umlaufzeit & Verzögerung anhand von Zeitstempel
- Annahmen: Laufzeiten symmetrisch und stabil
- Genauigkeit über Internet in der Größenordnung $10\,\mathrm{ms}$

White Rabbit im CNGS-Experiment

- Messung von Neutrino-Flugzeit zwischen CERN und LNGS ($732\,\mathrm{km}$)
- Möglichst genaue Zeitsynchronisation zwischen Standorten
- White Rabbit: Kombination verschiedener Techniken
  - Synchronous Ethernet über Glasfaser
  - Atomuhren als Taktgeber
  - Precision Time Protocol (PTP) mit Hardware-Unterstützung
  - Global Positioning System (GPS)
- Ausgleich von Temperaturschwankungen durch ständige Phasen-Messung
- Genauigkeit: $0.5\,\mathrm{ns}$, Präzision: $10\,\mathrm{ps}$ ($5\,\mathrm{km}$ Teststrecke)

M. Lipiński, T. Włostowski, J. Serrano, and P. Alvarez.
White Rabbit: a PTP Application for Robust Sub-nanosecond Synchronization.
2011 International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication (ISPCS'11), p. 25–30, September 2011.

Zeit in verteilten Systemen

Logische Uhren

Logische Uhren

Grundidee
Kausale Zusammenhänge entstehen durch gegenseitige Beeinflussung, d. h.
Nachrichtenaustausch in verteiltem System

Modell
Kommunizierende Prozesse P_i versehen Ereignisse a mit logischem Zeitstempel $C_i\langle a\rangle$

Uhrenbedingung
Wenn Ereignis b aufgrund von a aufgetreten ist ($a \rightarrow b$), muss die Relation $C_i\langle a\rangle < C_j\langle b\rangle$ gelten

- Eigenschaften: transitiv, asymmetrisch $\Rightarrow$ Striktordnung
  $\rightarrow$ Umkehrschluss **nicht** möglich: Aus $C_i\langle a\rangle < C_j\langle b\rangle$ folgt nicht $a \rightarrow b$!
- Erweiterte Ansätze können zusätzliche Eigenschaften garantieren
  - Totalordnung
  - Zuverlässige Unterscheidung abhängiger Ereignisse ($\rightarrow$ Vektoruhr)

Uhrenbedingung von Lamport

- Uhrenbedingung im Kontext von kommunizierenden Prozessen
  1. Aufeinanderfolgende Ereignisse innerhalb eines Prozesses erhalten streng monoton steigende Zeitstempel
  2. Senden einer Nachricht muss vor deren Empfang passiert sein, daher muss gelten:

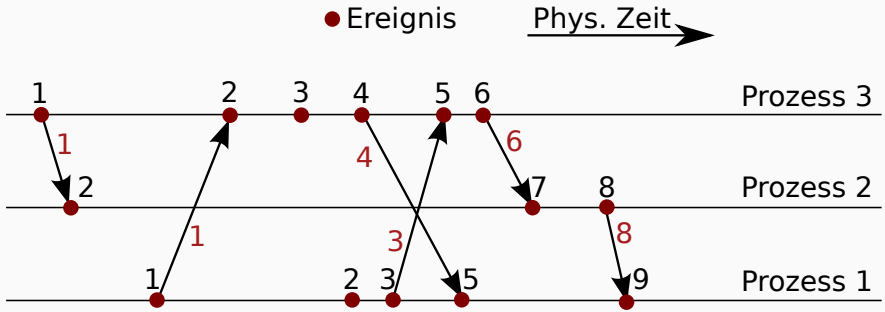
$$C_i\langle Senden\rangle < C_j\langle Empfang\rangle$$

- Regeln für **Implementierung**
  1. Die logische Uhr C_i eines Prozesses P_i muss zwischen zwei aufeinanderfolgenden Ereignissen immer inkrementiert werden
  2. Erhält ein Prozess P_j eine Nachricht und deren Zeitstempel $C_i\langle Senden\rangle$ ist größer oder gleich dem Wert der Uhr C_j des Prozesses P_j, muss die Uhr auf einen Wert größer $C_i\langle Senden\rangle$ erhöht werden

Leslie Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM, 21:558–565, July 1978.

Uhrenbedingung von Lamport

- Kein genereller Zusammenhang mit Ablauf physikalischer Zeit
  - Kein gleichmäßiger Verlauf
  - Folge von Ereignissen nach logischer Zeit nicht zwangsläufig identisch mit physikalischem Auftreten



Lamport-Uhr: Erweiterungen

- Für viele Anwendungen Totalordnung wünschenswert
  - Wenn Zeitstempel $C_i\langle a\rangle$ und $C_j\langle b\rangle$ gleich, gilt weder $C_i\langle a\rangle < C_j\langle b\rangle$, noch $C_j\langle b\rangle < C_i\langle a\rangle$
  - Beliebiges **determiniertes** Verfahren zur Festlegung möglich
  - Am einfachsten: Global eindeutige Prozess-ID entscheidet
  - Keine Beeinflussung der Aussage bezüglich kausaler Zusammenhänge

- Implementierung von Relationen in Java mittels `Comparable`

```
public interface Comparable<T> {
  public int compareTo(T obj);
}
```

- Methode `compareTo()` liefert Zahl abhängig von Relation

  Negativ : `this` $<$ `obj`

  „Null“ : `this` $=$ `obj`, entspricht `equals()`

  Positiv : `this` $>$ `obj`

Synchronisation

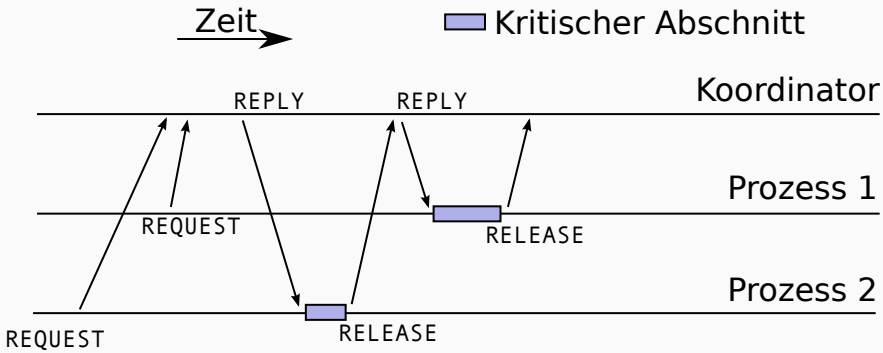
Synchronisation in verteilten Systemen

- Koordination von Zugriffen auf gemeinsame Betriebsmittel in verteilten Systemen notwendig
- Verschiedene Möglichkeiten:
  - Zentraler Koordinator
  - Koordination untereinander
- Exklusiver Zugriff äquivalent zur Bestimmung totaler Ordnung:

  $\Rightarrow$ Einigung auf Reihenfolge der Zuteilung der Ressource

Zentraler Koordinator

- Zentraler Prozess ist zuständig für Koordination
- Anfragen werden geordnet und in Reihenfolge freigegeben
- Nachrichtenfolge: REQUEST, REPLY, RELEASE



Lock-Protokoll von Lamport (1)

- **Idee:** Ausnutzen der totalen Ordnung über logische Zeitstempel bezüglich Lock-Anfragen

- Voraussetzungen:
  - FIFO-Protokoll:
    Nachrichten eines Absenders müssen in der Reihenfolge ankommen, in der sie abgeschickt wurden
  - Zuverlässiger Nachrichtenkanal
  - Toleriert ohne weitere Maßnahmen keine Ausfälle

- Ablauf:
  1. REQUEST via Broadcast an alle Prozesse versenden
  2. Warten bis eigene Anfrage vorne in der REQUEST-Warteschlange steht **und** kein anderer Prozess sich vor dem eigenen Eintrag einreihen kann
  3. Kritischen Abschnitt ausführen
  4. Broadcast der RELEASE-Nachricht zum Freigeben des Locks

Lock-Protokoll von Lamport (2)

- Warteschlangenverwaltung:
  - Einreihen von eingehenden REQUEST-Nachrichten (auch selbst gesendete)
  - Sortierung nach totaler Ordnung über Zeitstempel logischer Uhr
  - Entfernen des korrespondierenden Elements bei Empfang von RELEASE (auch selbst gesendete)

- Einreihen vor eigenem Eintrag nicht mehr möglich, wenn von allen Prozessen bereits Nachrichten mit größerem Zeitstempel als der des eigenen REQUESTS empfangen wurden

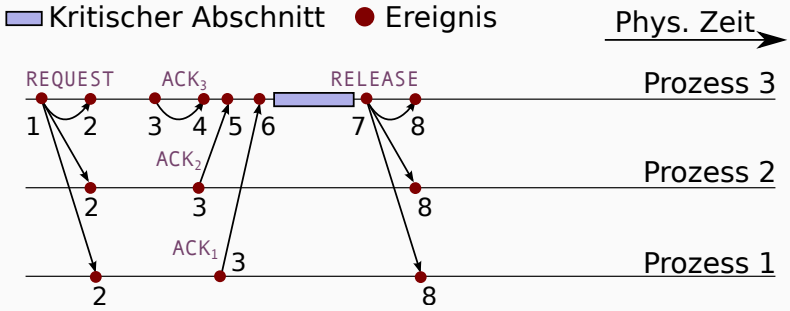
  $\Rightarrow$ Merken des jeweils zuletzt empfangenen Zeitstempels je Prozess
  - FIFO-Eigenschaft garantiert streng monotonen Anstieg

- Empfang einer REQUEST-Nachricht von anderem Prozess muss zudem mit ACK-Nachricht an Absender quittiert werden
  - Notwendig, um Fortschritt zu garantieren
  - Dient lediglich der Erhöhung und Übermittlung der logischen Uhr
  - Bestätigung durch Nachrichtenaustausch auf Anwendungsebene implizit möglich

Lock-Protokoll von Lamport (3)

- Eigenschaften:
  - RELEASE-Nachrichten sind total geordnet
  - Erweiterungsmöglichkeiten bezüglich Fehlertoleranz, da REQUEST-Warteschlange implizit repliziert
  - Geringe Latenzen bei häufig beanspruchten Locks
  - Allerdings größeres Nachrichtenaufkommen als bei zentralem Koordinator

- Beispiel:



Aufgabe 6

Übungsaufgabe 6: Überblick

- Verteilte Synchronisation mittels Lamport-Lock-Protokoll (für alle)
  - Sperrobjekt: Blockieren/Deblockieren und Umwandlung von lokalen Sperranfragen in Ereignisse für Lamport-Lock-Protokollkomponente

```
public class VSLamportLock {
    public void lock();
    public void unlock();
}
```

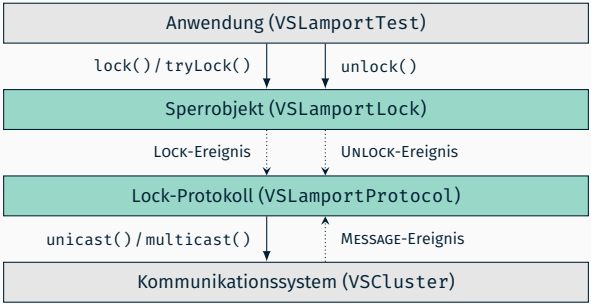
  - Implementierung der Lamport-Lock-Protokollkomponente

```
public class VSLamportProtocol {
    public void init();
    public void event(VSLamportEvent event);
}
```

- Zeitbeschränkte Sperrversuche (optional für 5,0 ECTS)
  - Spezifizierbare, maximale Wartedauer
  - Schnittstellenerweiterung

```
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException;
```

Zusammenspiel der Komponenten



- Bereitgestellte *Test-Anwendung* mit 4 Testfällen
- Zu implementierende Lock-Protokoll-Logik
  - **Benutzerschnittstelle** (Sperrobjekt)
  - **Protokollschicht** (Lock-Protokolll)
- Bereitgestelltes *Kommunikationssystem*: Klasse zum zuverlässigen Senden von Nachrichten an bestimmte (`unicast()`) oder alle Prozesse (`multicast()`) im Verbund (Cluster)

Lock-Protokoll: Benutzerschnittstelle

- Implementierung in zwei Teilen
  - Benutzerschnittstelle (`VSLamportLock`)
  - Protokollschicht (`VSLamportProtocol`)
- Bietet Anwendungen blockierenden `lock()`-Aufruf und `unlock()`-Aufruf zum Entsperren
- Implementierung des blockierenden Verhaltens durch lokalen Semaphor
- Interaktion mit Protokollschicht erfolgt mittels der übergebenen `VSLamportProtocol`-Referenz im Konstruktor von `VSLamportLock`

```
public class VSLamportLock {
    public VSLamportLock(VSLamportProtocol protocol) { [...] }
}
```

- Koordinierung von Ressourcenzugriffen
- Zentrale Methoden:

```
Semaphore(int permits);

void acquireUninterruptibly();
boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException;
void release();
```

`Semaphore()`	Initialisiert Semaphore mit Startwert
`acquireUninterruptibly()`	Semaphore **un**unterbrechbar belegen
`tryAcquire()`	Semaphore unterbrechbar belegen, schlägt nach Timeout fehl
`release()`	Semaphore freigeben

- Beispiel:

```
Semaphore s = new Semaphore(1);
s.acquireUninterruptibly();
[...]
s.release();
```

Lock-Protokoll: Protokollschicht

- Implementierung in Klasse `VSLamportProtocol` verarbeitet Ereignisse vom Typ `VSLamportEvent` sequentiell (aus Konsistenzgründen)
  → Ereignisse haben einen Typ (`type`) und ein zugeordnetes Objekt (`content`)

```
public class VSLamportEvent {
    [...]
    public VSLamportEventType getType() { return type; }
    public Object getContent() { return content; }
}
```

- Trennung Protokoll-interner Ereignisse von Ereignissen für die höhere Protokollschicht
- Vorgegebene Ereignistypen:

```
public enum VSLamportEventType { MESSAGE, LOCK, UNLOCK }
```

  - Protokollinterner Ereignistyp: `MESSAGE`
  - Typen für Ereignisse aus Benutzerschnittstelle heraus: `LOCK` und `UNLOCK`

- Vorsicht beim Umgang mit Lock-Anfragen in der Warteschlange
  - Korrekte Zuordnung zwischen `lock()`-Aufrufen und den erzeugten REQUEST-Nachrichten notwendig
  - Schnell aufeinanderfolgende Lock-Anforderungen können sonst zu Problemen führen

Kommunikationssystem

- Vorgegebene Klasse `VSClusterImpl` implementiert die Schnittstelle des Kommunikationssystems (`VSCluster`)
  - Ausgelieferte Nachrichten/Ereignisse sind immer vom Typ `MESSAGE`
  - Jeder einzelne Lamport-Protokoll-Prozess im Verbund hat ein eigenes, lokales `VSCluster`-Objekt
  - Kommunikationssystem läuft stets in einem eigenen Thread, d. h., Ereigniszustellung erfolgt immer aus demselben Thread heraus
- Methoden der Kommunikationssystemschnittstelle `VSCluster`
  - ID des lokalen Lamport-Protokoll-Prozesses und #Prozesse im Verbund

```
public int getProcessID();
public int getSize();
```

  - Nachricht an einen bestimmten Prozess im Verbund senden

```
public void unicast(Serializable msg, int processID) throws IOException;
```

  - Nachricht an alle Prozesse im Verbund senden

```
public void multicast(Serializable msg) throws IOException;
```

- Über `unicast()` bzw. `multicast()` gesendete Nachrichten werden sequentiell in FIFO-Reihenfolge durch die `event()`-Methode am jeweiligen Protocol-Objekt ausgeliefert

Test-Anwendung

- Einfaches Testen der Implementierung durch Test-Anwendung
- Konfiguration: Zu verwendende Rechner in Datei `my_hosts` ablegen
- Ausführung: Start im CIP-Pool mit `distribute.sh`
  - 1. Parameter gibt Art des Testfalls an (siehe unten)
  - Skripte können im Basisverzeichnis der eigenen Paket-Hierarchie abgelegt werden; alternativ:
    - Explizites Spezifizieren des Basisverzeichnisses (2. Parameter, optional)
    - und ggf. (3. Parameter, optional) des Verzeichnisses von `my_hosts`
- Überprüfung: Skript `checklogs.sh` ausführen
- Verschiedene Testfälle (Mindestlaufzeit: 1 Minute)
  - Einfacher Fall (Aufruf mit Parameter `simple`)
    - Beantragen (`lock( )`) und Freigeben (`unlock( )`) in Schleife
    - Darf nicht stehen bleiben
  - Komplexer Fall (Aufruf mit Parameter `fancy`)
    - Gegenseitiges Umbuchen von Beträgen zwischen Konten
    - „Sum is“-Zeile darf sich nicht ändern (max. Betrag pro Rechner: 1000)
    - Darf nicht stehen bleiben
  - Debugging-Hilfe (Aufruf mit Parameter `debug`): Testet auf häufige Probleme
  - Testfälle für erweiterte Variante: siehe nächste Folie

- Basisvariante (`lock()`) würde Anwendung so lange blockieren, bis der kritische Abschnitt tatsächlich für sie freigegeben ist
- Erweiterung der Sperrobjektimplementierung um folgende Methode

```
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException;
```

  - Spezifizieren einer maximalen Blockierzeit über `timeout` und `unit` (z. B. `TimeUnit.MILLISECONDS`)
  - Methode reagiert auf Unterbrechung des die Methode aufrufenden Threads mittels einer `InterruptedException`

- Zieht in der Regel Änderungen von `VSLamportLock` **sowie** `VSLamportProtocol` nach sich
- Zwei weitere Testfälle
  - Funktionalität von einfachem (Parameter `simple-try`) und komplexem Fall (Parameter `fancy-try`) grundlegend analog zu `simple`- bzw. `fancy`-Testfall
  - `tryLock()`- statt `lock()`-Aufrufe (jeweils so lange, bis Lock vergeben wurde)
  - Dynamische Anpassung des Timeouts in Abhängigkeit von erfolgreichen und nicht erfolgreichen Aufrufen von `tryLock()`