

- Verteilte Synchronisation
  - Zeit in verteilten Systemen
  - Logische Uhr
  - Synchronisation
  - Aufgabe 6



- Ist Ereignis A auf Knoten X passiert bevor B auf Y passiert ist?  
Beispiele: Internet-Auktion, Industrie-Steuerungen, ...
- Prinzipiell keine konsistente Sicht auf Gesamtsystem möglich
  - Unabhängigkeit von Ereignissen
  - Informationsaustausch mit Latenzen verbunden
- ⇒ Nur näherungsweise Lösungen möglich
- Bestes Verfahren abhängig von Einsatzgebiet und notwendigen Eigenschaften



## Echtzeit-basierte Uhren

- Nutzung eines gemeinsamen Zeitsignals
  - Auflösung beschränkt
  - Schwierig über größere Entfernungen  
Ausbreitungsgeschwindigkeit max. 20 cm/ns,  $\frac{1}{1\text{GHz}} = 1\text{ns}$
- Nachrichten mit Zeitstempel lokaler, physikalischer Uhren versehen
  - Wenig Kommunikationsaufwand
  - Ohne Synchronisation: Zunehmende Abweichungen
- Kombination verschiedener Verfahren zur Verbesserung der Genauigkeit



## Synchronisation von Echtzeit-Uhren: NTP, PTP

- Stellen lokaler Uhr basierend auf Referenz-Uhr
- In der Praxis verwendete Protokolle:
  - Network Time Protocol (NTP)
  - Precision Time Protocol (PTP)
- Prinzipieller Ablauf:
  - Berechnung von Umlaufzeit & Verzögerung anhand von Zeitstempel
  - Annahmen: Laufzeiten symmetrisch und stabil
  - Genauigkeit über Internet in der Größenordnung 10 ms



## White Rabbit im CNGS Experiment

- Messung von Neutrino-Flugzeit zwischen CERN und LNGS (732 km)
- Möglichst genaue Zeitsynchronisation zwischen Standorten
- White Rabbit: Kombination verschiedener Techniken
  - Synchronous Ethernet über Glasfaser
  - Atom-Uhren als Taktgeber
  - Precision Time Protocol (PTP) mit Hardware-Unterstützung
  - Global Positioning System (GPS)
- Ausgleich von Temperaturschwankungen durch ständige Phasen-Messung
- Genauigkeit: 0,5 ns, Präzision: 10 ps (5 km Teststrecke).



## Logische Uhren

- **Grundidee:** Kausale Zusammenhänge entstehen durch gegenseitige Beeinflussung, d.h. Nachrichtenaustausch in verteiltem System
- **Modell:** Untereinander kommunizierende Prozesse  $P_i$  versehen auftretende 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
- 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)



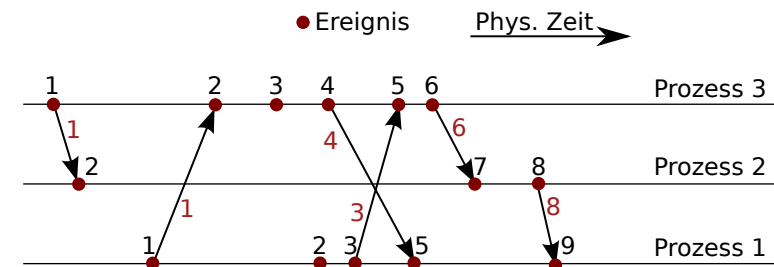
## Uhrenbedingung von Lamport

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



## Uhrenbedingung von Lamport

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



## Erweiterungen Lamport-Uhr

- 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 :  $\text{this} < \text{obj}$

0 :  $\text{this} = \text{obj}$ , entspricht equals()

Positiv :  $\text{this} > \text{obj}$



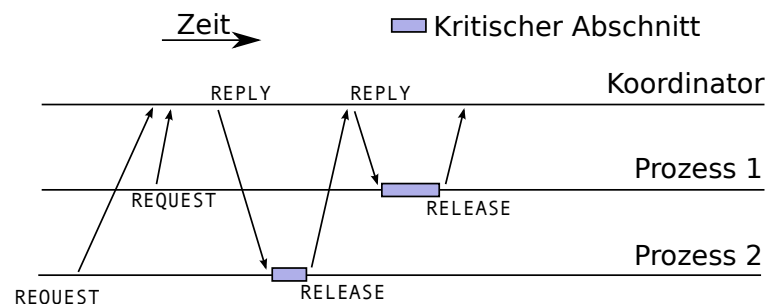
## Synchronisation in verteilten Systemen

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

- **Idee:** Ausnutzen der Totalen Ordnung über Zeitstempel von logischer Uhr bezüglich Lock-Anfragen
- Voraussetzungen:
  - FIFO-Protokoll: Nachrichten eines Absenders müssen immer 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 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

- Warteschlangenverwaltung:
  - Einreihen von eingehenden REQUEST-Nachrichten (auch selbst gesendete)
  - Sortierung nach totaler Ordnung über Zeitstempel logischer Uhr
  - Entfernen des kleinsten Elements bei Empfang von RELEASE (auch selbst gesendete)
- Einreihen vor eigenem Eintrag nicht mehr möglich wenn von allen Prozessen bereits Nachricht mit größerem Zeitstempel als der eigene REQUEST empfangen wurden
  - ⇒ Merken des jeweils zuletzt empfangenen Zeitstempels je Prozess
  - FIFO-Eigenschaft garantiert streng monotonen Anstieg



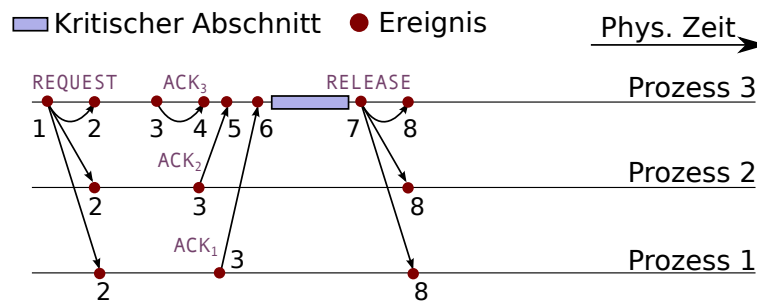
## Lock-Protokoll von Lamport

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



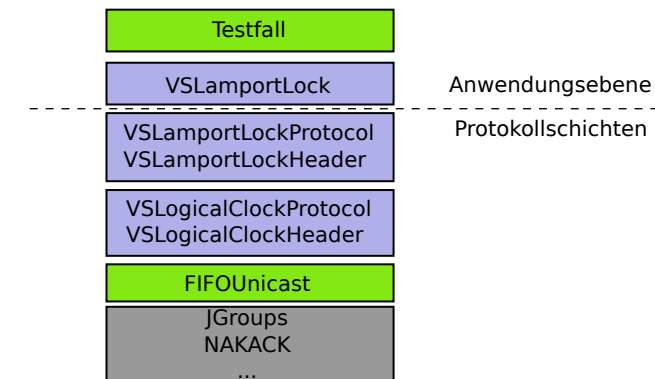
## Lock-Protokoll von Lamport

- Beispiel:



## Protokollstapel

- Grundlage für die zu implementierenden Protokollschichten bildet Basis mit FIFO-Garantien
  - JGroups NAKACK garantiert FIFO **nur** bezüglich Broadcast-Nachrichten
  - ⇒ Bereitgestellte FIFOunicast erfüllt Eigenschaft
- Zu implementieren: Schichten für verteilte Synchronisation zwischen FIFO-Protokoll und Anwendung



## Lock-Protokoll: Benutzerschnittstelle

- Implementierung in zwei Teilen: Benutzerschnittstelle und Protokollschicht
- Benutzerschnittstelle bietet Anwendungen blockierenden `lock()`-Aufruf und `unlock()` zum Entsperren
- Implementierung des blockierenden Verhaltens durch lokale Semaphore oder `wait()/notify()`
- Interaktion mit `VSLamportLockProtocol` mittels Suchen der Klasse im Protokollstack und Registrieren des Objekts für Rückrufe

```
JChannel channel;  
VSLamportLockProtocol myLockProtocol =  
    (VSLamportLockProtocol)channel.getProtocolStack()  
        .findProtocol(VSLamportLockProtocol.class);
```



## Lock-Protokoll: Protokollschicht

- Implementierung in Klasse `VSLamportLockProtocol`
- Trennung Protokoll-interner Nachrichten von Nachrichten für höhere Protokollschichten
  - Header zur Unterscheidung
  - Senden interner Nachrichten durch Aufruf von `down_prot.down()` aus `up()` heraus
  - Interne Nachrichten nicht an höhere Schichten weiterreichen  
⇒ `return null;`
- Eigene `RELEASE`-Nachrichten beim Freigeben des Locks sollten lokal und synchron aus der Warteschlange entfernt werden, da es sonst zu Problemen bei schnell aufeinanderfolgenden Lock-Anforderungen kommt



## Logische Uhr

- Implementierung in Klasse `VSLogicalClockProtocol`
- Zeitstempel werden in Form von Header an *jede* gesendete Nachricht angefügt
- Protokollschichten können nebenläufig ausgeführt werden, daher ist Synchronisation notwendig!
  - Nachrichten können sich auch innerhalb des Protokollstacks überholen
- Statische Methode `getMessageTime()` soll Extrahieren des Zeitstempels aus einer Nachricht ermöglichen



## Test-Anwendung

- Bereitgestellte Anwendung erlaubt einfaches Testen der Implementierung
- Konfiguration wie bei Übungsaufgabe 5:
  - Zu verwendende Rechner in Datei `my_hosts` ablegen
  - Start im CIP-Pool mit `distribute.sh`
  - Skripte müssen im Basisverzeichnis der eigenen Paket-Hierarchie abgelegt werden (Eclipse: `bin`-Verzeichnis)
- Zwei verschiedene Testfälle:
  - Einfacher Fall (Aufruf ohne Parameter) :**
    - In Endlosschleife Beantragen und Freigeben
    - Darf nicht stehen bleiben
  - Komplexer Fall (mit Parameter `fancy`) :**
    - Gegenseitiges Umbuchen von Beträgen zwischen Konten
    - Summe darf sich nicht ändern („Sum is“)
    - Darf nicht stehen bleiben



## Quellenangaben

---



Leslie Lamport.

Time, clocks, and the ordering of events in a distributed system.

*Commun. ACM*, 21:558–565, July 1978.

