

## Verteilte Synchronisation

- Zeit in verteilten Systemen

- Logische Uhr

- Synchronisation

- Aufgabe 6

  - JGroups

  - Lock-Protokoll



## Verteilte Synchronisation

Zeit in verteilten Systemen

Logische Uhr

Synchronisation

Aufgabe 6

JGroups

Lock-Protokoll



- 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
- ⇒ Nur näherungsweise Lösungen möglich
- Bestes Verfahren abhängig von Einsatzgebiet und notwendigen Eigenschaften

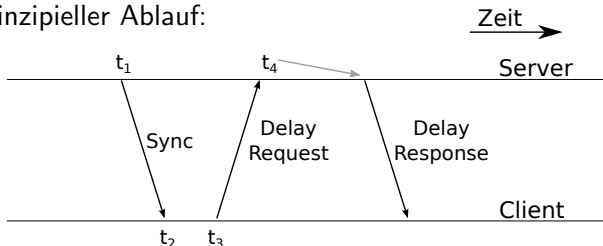


- Nutzung eines gemeinsamen Zeitsignals
  - Auflösung beschränkt
  - Schwierig über größere Entfernungen
    - Ausbreitungsgeschwindigkeit: max. 20 cm/ns,  $\frac{1}{1\text{GHz}} = 1\text{ ns}$
- Nachrichten mit Zeitstempel lokaler, physikalischer Uhren versehen
  - Wenig Kommunikationsaufwand
  - Ohne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



# Synchronisation von Echtzeituhren: NTP, PTP

- Stellen lokaler Uhr basierend auf Referenzuhr
- 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
  - Atomuhren 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).



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.



## Verteilte Synchronisation

Zeit in verteilten Systemen

**Logische Uhr**

Synchronisation

Aufgabe 6

JGroups

Lock-Protokoll



- **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  
 $\rightarrow$  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 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.



Leslie Lamport.

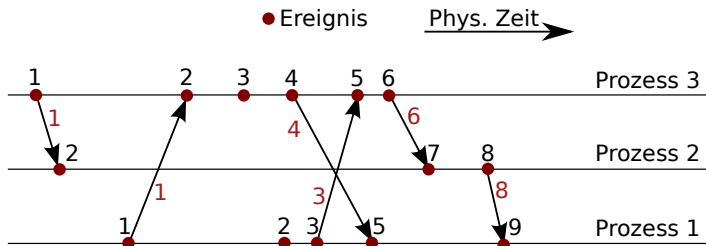
**Time, Clocks, and the Ordering of Events in a Distributed System.**

*Communications of the ACM*, 21:558–565, July 1978.



# Uhrenbedingung von Lamport

- Kein genereller Zusammenhang mit Ablauf physikalischer Zeit
  - Kein gleichmäßiger Verlauf
  - Abfolge 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`

0 : `this = obj`, entspricht `equals()`

Positiv : `this > obj`



## Verteilte Synchronisation

Zeit in verteilten Systemen

Logische Uhr

Synchronisation

Aufgabe 6

JGroups

Lock-Protokoll

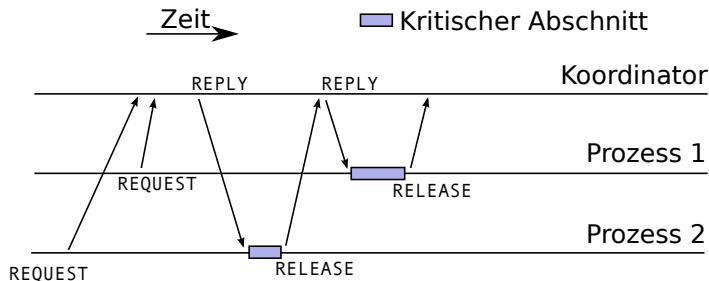


- Koordination von Zugriffen auf gemeinsame Betriebsmittel in verteilten Systemen notwendig
- Verschiedene Möglichkeiten:
  - Zentraler Koordinator
  - Koordination untereinander
- Exklusiver Zugriff äquivalent zur Bestimmung totaler Ordnung: Einigung auf Reihenfolge der Zuteilung der Ressource



# Zentraler Koordinator

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



# Lock-Protokoll von Lamport (1)

---

- **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



- 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 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





## Lock-Protokoll von Lamport (3)

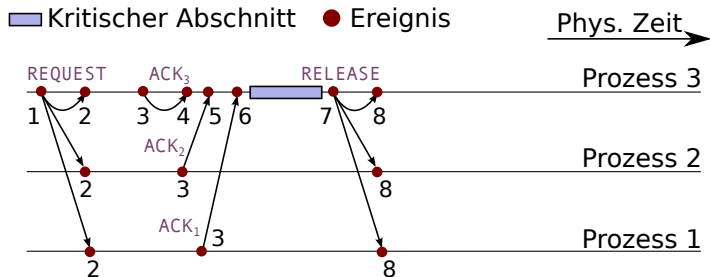
---

- 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 (4)

## ■ Beispiel:



## Verteilte Synchronisation

Zeit in verteilten Systemen

Logische Uhr

Synchronisation

### Aufgabe 6

JGroups

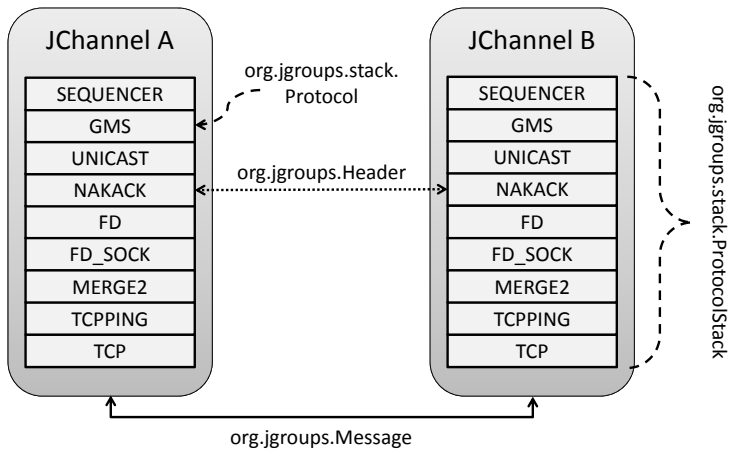
Lock-Protokoll



- JGroups
  - Bibliothek und **Framework für zuverlässige Gruppenkommunikation**
    - Virtual Synchrony, Zustandstransfer, Zustellungsgarantie
  - Durch **modularen Aufbau** über Konfiguration an bestehende Erfordernisse anpassbar
- Verwendung
  - Knoten verbinden sich mittels eines `org.jgroups.Channel`-Objekts
  - Nachrichten (`org.jgroups.Message`) können per Unicast oder Multicast versendet werden
  - Auslieferung von Nachrichten erfolgt asynchron (`org.jgroups.MessageListener`)
  - Benachrichtigungen über Gruppenzusammensetzung (`org.jgroups.View`) sind ebenfalls asynchron (`org.jgroups.MembershipListener`)
- Siehe auch Folien zu Übungsaufgabe 5 (Replikation)
- API-Dokumentation: <http://jgroups.org/javadoc/index.html>



Protokoll-Stack von JGroups ist **konfigurier- und erweiterbar**



- Bestehende Protokolle (Auswahl)
  - SEQUENCER
    - Realisiert eine totale Ordnung auf Basis von NAKACK
  - GMS (Group Membership Service)
    - Protokoll für Gruppenmitgliedschaft und Sichten (Views)
  - NAKACK (Not Acknowledge, Acknowledge)
    - Implementiert FIFO-Multicast
  - FD (Failure Detection)
    - Heartbeat-Protokoll für Ausfallerkennung
  - TCP/UDP
    - Transportprotokolle
- Vollständige Liste: `jg-protocol-ids.xml` (→ JGroups-Quellcode)
- Implementierung eigener Protokolle möglich
  - Protokolle leiten von der Klasse `org.jgroups.stack.Protocol` ab
  - Registrierung mittels XML-Datei oder zur Laufzeit

```
ClassConfigurator.addProtocol(short id, Class protocol);
```



- Empfang und Versand von Nachrichten sowie Statusänderungen werden als *Ereignisse* im Protokoll-Stack propagiert

- Zugehörige Methoden der Klasse `Protocol`

```
Object down(Event evt); // Aufruf durch hoehere Schicht
Object up(Event evt);   // Aufruf durch untere Schicht
```

- Rückgabe von Ergebnis der unteren (`super.down(evt);`) oder höheren Schichten (`super.up(evt);`) bzw. `null`, wenn Ereignis verworfen wurde

- Klasse `org.jgroups.Event`

```
int getType(); // Typ des Ereignisses
Object getArg(); // Mitgeliefertes Argument
```

Typ (Event.*)	Beschreibung	Argument
MSG	Versand (down) oder Empfang (up) einer Nachricht	<code>org.jgroups.Message</code>
VIEW_CHANGE	Änderung der aktuellen Sicht (up und down)	<code>org.jgroups.View</code>
SET_LOCAL_ADDRESS	Setzen der lokalen Adresse (down)	<code>org.jgroups.Address</code>



- Protokolle tauschen über *Header* interne Daten zwischen Knoten aus
  - Header leiten von `org.jgroups.Header` ab
  - Müssen vergleichbar zu Protokollen registriert werden

```
ClassConfigurator.add(short id, Class header);
```

- Header sind Teil von Nachrichten und werden mit diesen übertragen
- Zugehörige Methoden der Klasse `org.jgroups.Message`

- Hinzufügen eines Header an Nachricht

```
void putHeader(short id, Header hdr);
```

- Rückgabe des Header einer Nachricht oder `null`, wenn nicht vorhanden

```
Header getHeader(short id);
```

- Beispiel:

```
Message msg = new Message(...);  
VSTotalOrderHeader hdr = VSTotalOrderHeader.createMulticast(msgid);  
msg.putHeader(id, hdr); // id: Attribut aus Oberklasse
```





- JGroups verwendet eigene Mechanismen zur Serialisierung und Deserialisierung zum Beispiel von Headern
  - Schnittstelle `org.jgroups.util.Streamable`

```
void writeTo(DataOutput out);    // Serialisierung
void readFrom(DataInput in);    // Deserialisierung
```

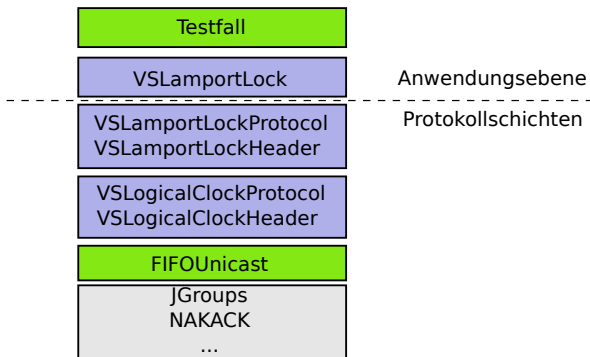
- Klassen müssen über XML-Datei oder zur Laufzeit registriert werden (siehe Registrierung von Headern)
- Hilfsmethoden in Klasse `org.jgroups.util.Util`

```
byte[] objectToByteBuffer(Object obj);
Object objectFromByteBuffer(byte[] buf, int off, int len);
```



# 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 `VSLampportLockProtocol` mittels Suchen der Klasse im Protokoll-Stack und Registrieren des Objekts für Rückrufe

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



- Implementierung in Klasse `VSLamportLockProtocol`
- Trennung Protokoll-interner Nachrichten von Nachrichten für höhere Protokollschichten
  - Header zur Unterscheidung
    - ⇒ `Message m = ...; m.getHeader(<header_id>)`
  - Senden interner Nachrichten
    - ⇒ Aufruf von `down_prot.down()` aus `up()` heraus
  - Interne Nachrichten nicht an höhere Schichten weiterreichen
    - ⇒ `return null;`
- Vorsicht beim Umgang mit Lock-Anfragen in der Warteschlange
  - Eigene Lock-Anfragen sollten lokal und synchron aus der Warteschlange entfernt werden (im Falle von `unlock()`-Aufruf)
  - Schnell aufeinanderfolgende Lock-Anforderungen können sonst zu Problemen führen



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



- 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 (Eclipse: `bin`-Verzeichnis); alternativ:
    - Explizites Spezifizieren des Basisverzeichnisses (2. Parameter, optional)
    - und ggf. (3. Parameter, optional) des Verzeichnisses der Konfigurationsdateien (→ `stack.xml`, `logging.cfg`, `my_hosts`)
- Überprüfung: Skript `checklogs.sh` ausführen
- Zwei verschiedene Testfälle (Minstdauer: 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

