

# Verteilte Systeme

**Definition:** Rechnernetz ohne gemeinsamen Speicher, Datenaustausch per Nachrichtenkommunikation

**Vorteile:** Leistungssteigerung, Verfügbarkeit

**Nachteile:** Konsistenz, Uhr, Komplexität

**Speedup:**  $T_1(n)$ , Rechenzeit auf einem Kern,  $T_k(n)$ , Rechenzeit auf k Kernen

$$\frac{T_1(n)}{T_k(n)} \leq k$$

Eine schlechte Partitionierung (Aufteilung der Arbeit) kann zu einem Speedup  $< 1$  führen

## ISO Referenzmodell:

4. Transportlayer: TCP/UDP

5. Session Layer: Sicherungspunkte, Recovery

6. Presentation Layer: Verschlüsselung, Umwandlung von Datenformaten

7. Application Layer: FTP,SMTP etc.

## IP Adressen

V4: 32 Bit, 7/14/21 Bit Netzteil und 24/16/8 Hostteil

V6: 128 Bit

## UDP

Best effort, max 64kb

## ZMQ

Message Protocol: Daten werden als Nachrichten versendet

Dabei bietet ZMQ ein hohes Abstraktionslevel

### Patterns:

Request-Reply: Synchron, Request -> warten auf Reply

Push-Pull: Keine Antwort erwartet

Pub-Sub: 1 zu n, Datenverteilung

### Beispiel:

Source: Request

Broker: Reply (Source), Publish(Sink), Push(worker)

Worker: Pull und Request

Sink: Sub

### Transportvarianten:

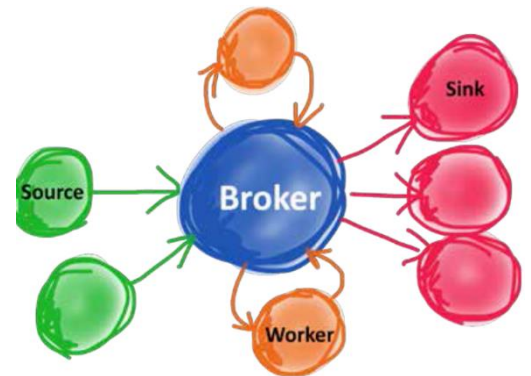
INPROC: Zwischen Threads desselben Prozesses

IPC: Zwischen Prozessen

TCP: TCP halt

PGM: pragmatic multicast

EPGM: encapsulated pgm



## Lastverteilung

**Kreative Lastverteilung:** Verteilte Programmierung

**Mechanische Lastverteilung:** Auslastung mehrerer Rechner

Aufteilung der Last in kleinere Pakete, je kleiner, desto eher haben alle Rechner die gleiche

Last, aber: Auf-/Verteilen kostet, zu kleine Pakete erhöhen Last

**Grundlage:** Lastwert auf Rechner verteilen, dann Lastpakete verteilen

**Metriken:** Prozessorauslastung (#Prozesse, Auslastung, Zeit), Speicherauslastung, Kommunikationslast

Dabei ist die Vergleichbarkeit nicht einfach: Unterschiedl. Prozessoren, andere Systeme

**Pull-Metrik:** Rechner ziehen Last an

**Push-Metrik:** Verteiler verteilen Last an wenig beschäftigte Rechner  
Broadcast, Teilmenge oder zufällig

Lastwerte, die ein Verteiler mitbekommt sind aufgrund der Laufzeit immer leicht veraltet  
Interpolieren und aus Last/Paket lernen

Bei Verteilung 2 Möglichkeiten: Code ist vorhanden -> Ausführen  
Code nicht vorhanden -> Code übertragen

**Verteilungsverfahren:**

Statische Verfahren: Optimale Verteilung vor dem Start ermitteln

Dynamische Verteilung: Ausführungsort für jedes Paket bei Erzeugung ermitteln

Ohne Migration: Lastpaket wechselt Ort nicht

Mit Migration: Lastpaket kann Ort wechseln

**Statische Lastverteilung:** Als Graph darstellbar und berechenbar

**Dynamisch mit Migration:** Identische Umgebung auf beiden Systemen notwendig, Adressraum und Prozessstatus übertragen

**Dynamisch ohne Migration:** Einfach, schnell und gute Ergebnisse. Bei Erstellung von Paket, Zielort bestimmen (gerne auch zufällig)

## Verteilte Zeit

Es fehlt eine globale genau gleiche Zeit

**Ansätze:** Synchronisation der Uhren (z.B. NTP)  
Logische Uhren (Lamport, Vektor)

## Uhrensynchronisation

1. 1 Rechner hat eine genaue Uhr, daran wird angeglichen

2. Jeder Rechner gleicht sich den anderen Uhren an

Grundannahme: Abweichung linear

**DCF77:** Zentraler Zeitserver in Braunschweig

Senden per Funk, Reichweite 1500km

**F. Christian:** Passiver Zeitserver, der auf Anfrage Timestamp sendet  
Transportzeit ignoriert

**NTP:** Hierarchisch, liefert Offset, Roundtrip und Fehlerrate

Stratum 1 (höchste Stufe) = GPS/Atomuhr, auch selbst baubar

**Verteilter Abgleich:** In Intervallen senden alle Rechner aktuelle Zeit per Broadcast  
Jeder berechnet Mittelwert

## Logische Uhren

**Ereignis:** Lokales oder Sende-/Empfangsereignis, vom Entwickler definiert, oder Anweisung

**Kausalität:** e ist kausal abhängig von f, wenn f Auswirkungen auf e hat  
Transitiv, Partielle Ordnung

**Logische Uhr:**  $LC: E \rightarrow H$  mit E: Ereignisse mit Kausalrelation, H: Zeitbereich

**Uhrenbedingung:**  $e_n <_k e_m \rightarrow LC(e_n) < LC(e_m)$

Wenn  $e_m$  kausal abhängig von  $e_n$  ist, dann muss der Zeitstempel von  $e_m$  größer sein, als der von  $e_n$

Umgekehrt ist das nicht notwendig (umgekehrte Uhrenbedingung)

### Lamportzeit:

Jeder Rechner hat eine eigene Logische Uhr (LC)

Lokales Ereignis:  $LC = LC + 1$

Sendeereignis:  $LC = LC + 1$ ;  $\text{Send}(\text{Message}, LC)$ ;

Die aktuelle Uhrzeit nach Erhöhen wird mitgesendet

Empfangsereignis:  $\text{Receive}(\text{Message}, LC_s)$ ;  $LC = \max(LC, LC_s) + 1$

Die Uhrzeit wird auf die des Senders gesetzt, wenn diese höher ist, danach wird sie um 1 erhöht

Uhrenbedingung gilt durch Empfangsereignis

Umkehrung der Uhrenbedingung gilt nicht!

$$LC(a) < LC(b) \rightarrow (a <_k b) \vee (a || b) \quad (|| = \text{unabhängig})$$

Zähler muss groß genug sein, sonst überlauf (64Bit ist noch genug)

### Erweiterte Lamportzeit:

Um totale Ordnung zu erreichen wird dem Zeitstempel noch eine RechnerID angefügt

$$LC_E(A, a) < LC_E(B, b) \leftrightarrow LC(a) < LC(b) \vee (LC(a) = LC(b) \wedge A < B)$$

### Vektorzeit:

Feste Menge von Rechnern, Erweiterung allerdings möglich

Zeitstempel ist ein Vektor der Größe n, die Anzahl der Rechner

$$VC(a) = \begin{pmatrix} vc_0 \\ \dots \\ vc_n \end{pmatrix}$$

Initialisiere alle Komponenten auf 0.

Lokales Ereignis auf Rechner k:  $vc_k[k] = vc_k[k] + 1$

In der lokalen Uhr wird die lokale Komponente erhöht

Sendeereignis auf Rechner k:  $vc_k[k] = vc_k[k] + 1$ ;  $\text{Send}(\text{Message}, VC_k)$

Empfangsereignis auf Rechner k:

$$vc_k[k] = vc_k[k] + 1$$

$\text{Receive}(\text{Message}, vc_{\text{sender}})$

$\text{for}(i=0; i < n; i++)$

$$vc_k[i] = \max(vc_k[i], vc_{\text{sender}}[i])$$

Kausale Abhängigkeit:  $vc(a) \leq vc(b)$

Kausale Unabhängigkeit:

$$vc_a || vc_b \rightarrow \exists i, j \in \{0, \dots, n-1\}: (vc_a[i] < vc_b[i]) \wedge (vc_b[j] < vc_a[j])$$

Uhrenbedingung und Umkehrung gelten