

## Willkommen bei Verteilte Systeme!

Von Datenbanken  
über Webdienste  
bis zu p2p und Sensornetzen.



### Heute: Einführung und Überblick

A distributed system is a system that prevents you from  
doing any work, when a computer you have never heard  
about fails.

## Dozent

Anne Babenhausenheide

- Physik (Dipl., Dr., CO<sub>2</sub>)
- Seit 2004 p2p Netze
- Seit 2017 Softwareentwickler als Beruf
- Python, Scheme, HTML/CSS, Java, JS/TS, Fortran, Bash, Emacs, Ruby, ...
- arne\_bab@web.de

Vorlesung bis 2020 gemeinsam mit Carlo Götz.

## Sie

- Motivation für das Studium?
- Vorwissen zu verteilten Systemen?
- Erinnerung an Netztechnik?

## Ihre Wünsche

- Projekt Erfolgreich abschließen
- 
- 
- 

Im Cryptpad / auf Tafel sammeln.

## Präsentationen

- Wählen Sie ein Thema
- Themenvorschlag per E-Mail
  - gerne auch eigene
  - sehr gerne mit Vorwissen / Hobby
  - bis zur dritten VL

## Projekte 2022

- Berechnung auf mehreren Servern
- Botnetz mit CnC
- Chat mit 2 Servern
- Chat mit Python
- ESP32-Chat Captive Portal
- Game Of Life verteilt
- Gruppenchat in Java
- Gruppenchat mit Javascript
- IoT im Heimnetz
- Kommunikation zwischen Spielservern
- MQTT Broker
- Multithreading in Spice
- Nerdlegame als 1vs1
- Online TicTacToe
- Online-Spiel in GoDot
- P2P-Chat mit Directory-Server
- RPi Captive Portal
- Schiffeversenken 1vs1
- P2P Voice Chat über UDP
- Teilen von Trainingsplänen
- VS Chat in Java
- Viewer-Room

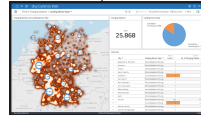
## Materialien

- Distributed Systems
  - Martin van Steen and Tanenbaum (2017)
  - kostenloses ebook: <https://www.distributed-systems.net>
  - ISBN-13: 978-1543057386
- Distributed Systems - An Algorithmic Approach – Sukumar Ghosh (2015).

## Arbeitsstelle

- Disy Informationssysteme GmbH in KA
- Datenanalyse, Reporting, Geoinformation und Geo-Analytics
- ~ 200 MA, Gründung 1997 (ich bin seit 2017 dabei)

<https://www.disy.net>



## Ziele der Vorlesung

- Sie verstehen, wo sie Verteilung vermeiden sollten.
- Sie verstehen, was sie beachten müssen, wenn Sie verteilen.
- Sie kennen Werkzeuge und Techniken, die ihnen helfen.
  - Sie können einschätzen, welche Garantien sie wirklich brauchen.
- Sie können einschätzen, welche **Kompromisse** sie eingehen sollten und welche nicht.
  - Latenz < 300ms für eine Webseite
  - Latenz < 30ms für Interaktive Systeme!
- Das passiert Ihnen nicht:  
<https://www.ccc.de/de/updates/2022/web-patrouille-ccc>

## Fragen

- Verteilung der Slides?
- Mail der Kurssprecher(in)?

## Themenideen

- FOSDEM Vorträge
- Verbindung in GNU Taler
- Wifi and LoRa mit Reticulum
- libresilient (p2p web fallback)
- Distributed Game Architecture
- Virtualization & Containerization
- Load Balancing + Autoscaling
- Verschlüsselung in Signal
- Botnet(-s)
- Message Queues: Rabbit, Kafka, ZeroMQ, ...
- Seti@home / Folding@Home
- telnet
- IRC
- ssh
- OAuth
- Ein eigenes Projekt
- Papers we love
- OWASP Top 10

## Projekte 2023

- Kaesekaestchen
- BlackJack
- ChannelMaster
- DistributedKeyValueStore
- distripool
- Password Encryption App
- Sonoff Basic R2 Webservice

## Folien u.ä.

- Weitere Quellen werden bei Verwendung aufgeführt.
- Folien: <https://www.draketo.de/software/vorlesung-verteilte-systeme>
- Glossar zu Netztechnik:  
<https://www.draketo.de/software/vorlesung-netztechnik#glossar-netztechnik>
- Gemeinsamer Glossar und Notizen in einem cryptpad.

Zur Vorbereitung: Drucken Sie bitte die Version mit vielen Folien  
pro Seite für schnelle Notizen.

## Arbeitshintergrund

Tech hinter Cadenza

- Java 24 (mit Spring und Ignite)
- Web-Tech (Web-Components, JS/TS)
- Verschiedene Datenbanken
- 3 mio LOC

Organisation

- ~80% in Karlsruhe, weitere international verteilt.
- N-Wochen Sprints, viel Home-Office ⇒ Zoom, Rocket Chat
- Tools: Linux, Mac, Windows, IntelliJ, VS Code, Emacs, ...
- Infrastruktur: CI mit Jenkins und Gitlab, Kubernetes-Cluster
- Trunk-based development mit slbs gegen Merge-Konflikte

## Erwartungen

### Meine Wünsche

Ich will, dass Sie gerne kommen.  
Es ist Arbeit, und Arbeit sollte Spaß machen.  
Ich will, dass Sie Verständnis von Verteilten Systemen mitnehmen.

## Ablauf Semester

- 0 Grundlagen
- 1 p2p (peer-to-peer)
- 2 Clocks/Zeit, usw.
- 3 Algorithmen, Shared State
- 4 Datenbanken konkret: CAP, DBs
- 5 Sicherheit in der Praxis, Sensornetze
- 6 Präsentationen der Studierenden, 15min pro 3er-Gruppe
- 7 tbd, Wiederholung

## Projekte 2021

- All Chats Are Beautiful
- Distributed Key/Value Store
- FreeChat
- HTTP-Tunneling
- IPC in Interpreter
- Kooperative WLED-Steuerung
- Load Balancer
- p2p Chat in Minecraft
- p2p iOS Kommunikation (Local Pal)
- Risiko Multiplayer
- Schiffe versenken
- Snake Multiplayer (Butchered Snake)
- Statuspage (openmonitor)
- Vier gewinnt
- Vollständig dezentralisierte MessageQueue
- WebChat

## Projekte 2024

- PrivateCA
- bilder malen und verschicken
- Nudge
- MonteCarloPi
- Paradoxs
- Schaltsimulation
- Ambi Mesh
- Suntimer
- ChatApp
- Schachspiel
- Schiffeversenken
- JWT Chat
- Chat in Python
- Konsolenbasierter Gruppenchat in Java
- Verteilte 3DS-ButtonBox
- Dokument-Kollaborationsanwendung
- Taskify

## Projekte 2025

- Task Manager
- Online 4-gewinnt
- SSL Chat Projekt
- Musik Synchronisation
- Kollaboratives Whiteboard
- Ich packe meinen Koffer
- lightweight Load Balancer
- Multiplayer Dartscounter
- Tic Tac Toe
- Reversi
- Asynchrone Kommunikation in Microservice Architekturen
- Verteilte Sensorauswertung
- Telefunken
- FileShare
- Verteilte Datenbank
- pushNotification
- QuizTogether
- Werwolf
- YT-Watch-Together
- Hot Potato

## Was sind verteilte Systeme?

- Was sind für Sie verteilte Systeme?
- Beispiele?

Im Cryptpad / auf Tafel sammeln.

## Sammlung von Knoten: Gruppenzugehörigkeit

- Einstieg ins System
- Darf jeder Knoten beitreten?
- Wie finden sie sich?
- Wie wird sichergestellt, dass nur mit Knoten innerhalb des Systems kommuniziert wird?
- Oft als Overlay Network realisiert.

## Middleware

- Separate Schicht über dem Betriebssystem.
- Von Applikationen verwendete Funktionalität:
  - verteilte Transaktionen
  - Fehler Recovery
  - Authentication & Authorization
  - Kommunikation mit anderen Knoten
  - ...
- Risiko: Effizienzverlust durch zu starke Garantien.

## Teilen von Ressourcen

- Auf geteilte Ressourcen zugreifen
- Beispiele:
    - Dropbox, GDrive etc.
    - Google Docs
    - p2p Filesharing (Bittorrent, Blizzard Launcher)
    - p2p Streaming (Spotify anfangs)
    - p2p Accountsuche (Skype anfangs)

## Offenheit

- Verteilte Systeme bieten und verwenden Komponenten, die einfach integriert oder wiederverwendet werden können
- Anforderungen:
  - definierte Schnittstellen (IDL (Syntax) + docs (Semantik))
  - Anwendungen portabel
  - Systeme erweiterbar

IDL: Interface Definition Language.<sup>3</sup>

Wer kontrolliert die API?

<sup>3</sup>Im OS: Hurd translator. Beispiele: [https://git.savannah.gnu.org/cgit/hurd/hurd.git/tree/hurd/msg\\_defs#n28](https://git.savannah.gnu.org/cgit/hurd/hurd.git/tree/hurd/msg_defs#n28)

## Ablauf heute

- Grundlagen und -begriffe verteilter Systeme
- Architekturen verteilter Systeme
- Prozesse und Threads
- Kommunikation

## Was sind verteilte Systeme?

*Ein verteiltes System ist eine Sammlung autonomer Rechenelemente, die den Nutzenden wie ein einzelnes kohärentes System erscheint.*

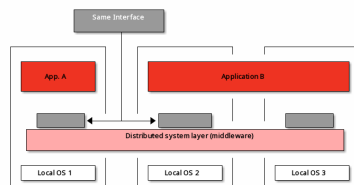
**autonome Rechenelemente** Arbeiten voneinander unabhängig, egal ob Soft- oder Hardware, auch **Knoten** genannt

**einzelnes kohärentes System** Nutzende haben den Eindruck ein einzelnes System zu bedienen (Erfordert Zusammenarbeit der Knoten).

## Overlay Networks

- Meist dicht verbunden
  - Für jedes Knotenpaar existiert ein Pfad zwischen den Knoten
- Zwei Varianten:
  - strukturierte Overlays** Jeder Knoten hat eine definierte Auswahl an Nachbarn mit denen er kommuniziert.
  - unstrukturierte Overlays** Jeder Knoten hat Referenzen zu zufällig ausgewählten anderen Knoten.

## Middleware als Betriebssystem für verteilte Systeme



## Verteilungstransparenz

Nutzenden soll nicht auffallen, dass Berechnungen und Daten über mehrere Computer verteilt sind.

Transparenz	Beschreibung
Zugriff	Versteckte Unterschiede in Datenrepräsentation.
Ort	Nutzer können nicht sagen wo sich ein Objekt physisch befindet
Relokation	Objekte können während ihrer Benutzung den Ort ändern.
Replikation	Verberge, dass ein Objekt repliziert ist.
Concurrency	Verberge gleichzeitige Nutzung eines Objekts.
Fehler	Verstecke Ausfall und Wiederinbetriebnahme von Objekten.

## Skalierbarkeit

3 Dimensionen:

- Größe: Nutzer- und Ressourcenanzahl können ohne Performanceprobleme steigen.
- Geographie: Nutzer und Ressourcen können durch große Distanzen getrennt sein, ohne dass Latenz wirklich auffällt.
- Administration: Das System kann unterschiedliche Organisationen umspannen.

$\log(N)$  ist gut.

## Einführung

- Sie kennen Charakteristiken verteilter Systeme
- Sie kennen Ziele verteilter Systeme
- Sie kennen die Dimensionen und Probleme der Skalierung
- Sie erkennen jede der „Fallacies of distributed systems“
- Sie kennen Arten verteilter Systeme

## Prozesse? (Abgrenzung)

Bilden mehrere Prozesse auf einem Computer ein verteiltes System?

- Autonome Knoten verfügen über eigenen Zeitbegriff.
- Es gibt keine globale Uhr.
- Probleme bei Synchronisation und Koordination.
- Prozesse auf einem System können sich hardwaregestützt synchronisieren.
  - aber mit Effizienzverlust<sup>1</sup>

<sup>1</sup>Auch low-level → branchless algorithms for koordinationsfreien Code

## Kohärentes, einzelnes System

- Nutzer kann nicht sagen, ob:
  - Berechnungen verteilt stattfinden
  - Daten verteilt gespeichert werden
  - Daten repliziert werden
- „Verteilungstransparenz“
- Problem: Knoten und Verbindungen zwischen Knoten können (und werden) jederzeit ausfallen.
  - Ausfall-Transparenz schwierig bis unmöglich.

## Warum? Ziele verteilter Systeme

*Just because it is possible to build a distributed system does not necessarily mean that it is a good idea.*

- Teilen von Ressourcen
- Verteilungstransparenz
- Offenheit
- Skalierbarkeit
- Macht minimieren

## Verteilungstransparenz: Probleme

- Latenz<sup>2</sup>
- Tradeoff: Verteilungstransparenz vs. Performance
- Konsistenz bei Replikation
- komplette Verteilungstransparenz ist unmöglich
  - Verteilung für Entwickelnde explizit? (Abstraktionsbruch; aber Effizienz!)

<sup>2</sup>Vielleicht nur eine langsame Datenbank? Optimieren mit Promises? Beispiel: on-demand overlay

## Probleme bei Skalierung

- “There is no free lunch”
- “There is no silver bullet”<sup>4</sup>
- “Law is hard”



<sup>4</sup>Bild: GermanWoodcut1722.



## Probleme bei Skalierung der Größe

- Speicherkapazität inkl. I/O Transferrate
- Rechenkapazität, begrenzt durch CPUs
- Netzwerk zwischen Nutzer und System



## Skalierungstechniken für Anwendungen

- vertikale Skalierung** mehr CPU, RAM etc. für die Computer (limitiert)
- horizontale Skalierung** mehr Kapazität durch Hinzufügen neuer Computer



## Fallbeispiel

Um welche Dimension der Skalierung handelt es sich? Welche Skalierungstechnik wird eingesetzt?

- Ziel: Windows Updates gleichzeitig laden
- Problem: Einbruch Netzleistung durch viele Downloads
- Skalieren: Proxyserver, der die Updates einmal von MS lädt
  - Downloads aus dem Firmennetz vom Proxy

**Dimensionen:** Größe, Geographie, Administration

**Techniken:** Latenz verbergen, Partitionierung, Replikation

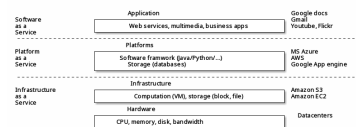


## Arten verteilter Systeme

- High Performance Distributed Computing (HPC)
- Cloud
- Distributed Information Systems
- Pervasive Systems



## Cloud Computing



Forschung: Probleme wie beim Grid, aber Cloud gibt Förderung.



## Distributed Information Systems (DIS)

- Einzelne Anwendungen zu einem verteilten System zusammenfassen
- Oft Legacy-Anwendungen.
- Methoden: verteilte Transaktionen, Enterprise Application Integration



## Probleme bei geographischer Skalierung

- viele bestehende Systeme erwarten schnelle LANs
  - Oft synchrone Kommunikation
  - werden langsam durch erhöhte Latenz<sup>5</sup>

<sup>5</sup>Manchmal sogar mit InfiniBand als Anforderung.



## Verstecken von Latenz

- Asynchrone Kommunikation
  - manchmal nicht möglich (Bsp: interaktive Anwendungen)
  - oft komplexere Algorithmen
- Berechnung im Client
  - Bsp: Form-Validierung in JS
  - Konsistenz?
    - Gleicher Code?
    - Gleiche Daten?



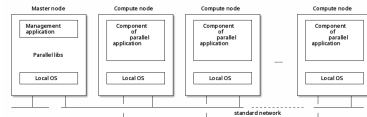
## PAUSE

## PAUSE



## HPC: Cluster Computing

- einzelne (rechenintensive) Anwendung wird parallel auf mehreren Computern ausgeführt
- Knoten durch LAN verbunden
- homogen



## Cloud Computing - Schichten

**Hardware** CPUs, Router, USVs, Kühlung

**Infrastruktur** verwendet Virtualisierung, um Kunden mit virtuellen Servern und Speichern zu versorgen

**Plattform** bietet Kunden APIs für Speicher usw. (Amazon S3)

**Anwendung** Programme für Endanwender (Google Docs)



## Pervasive Systems

Das Netz ist immer dabei.

- Treten durch Mobile und IOT verstärkt auf.
- Wird unterteilt in:
  - Ubiquitous Computing
  - Mobile Computing
  - Sensornetze



## Probleme bei Administrativer Skalierbarkeit

- Unterschiedliche Richtlinien für Verwaltung, Sicherheit etc.
- teilweise politische und soziologische Probleme

DSGVO?



## Partitionierung, Replikation, Caching

**Partitionierung** verteilte Komponenten auf mehrere Maschinen

- Bsp: DNS, DB-Sharding, WWW

**Replikation und Caching** Kopien auf mehreren Computern

- Bsp: Replizierte DBs, Browsercache, Proxies
- Führt zu Inkonsistenz
- globale Synchronisation ist langsam
- Abhängig von der Anwendung



## Fallacies of distributed Systems

- 1 The network is reliable
- 2 The network is secure
- 3 The network is homogeneous
- 4 The topology does not change
- 5 Latency is zero
- 6 Bandwidth is infinite
- 7 Transport cost is zero
- 8 There is one administrator



## HPC: Grid Computing

■ keine Annahmen bzgl. Ähnlichkeit von:

- Hardware
- Betriebssystem
- Netzwerk
- Sicherheit
- Administrative Domänen

■ Bsp: mehrere Hochschulen schließen ihre Cluster zu einem Grid zusammen.

- Grid kann von Allen verwendet werden.

Forschung: Probleme wie beim Cluster, aber Grid gibt Förderung.



## Cloud Computing - Gefahren

- Vendor Lock-in
- Sicherheit
- Datenschutz



## Ubiquitous Computing

- Geräte sind vernetzt
- Interaktion mit Benutzer ist kaum merkbar
- System erkennt Nutzerkontext und optimiert Interaktion
- Geräte laufen weitestgehend autonom
- System beherrscht viele Interaktionen
- Wer weiß, was ich mache?



## Mobile Computing

- Viele unterschiedliche Geräte: Smartphones etc.
- ständige Änderung des Ortes
- -> keine stabilen Routen, schwankende Geschwindigkeiten, Verbindungsausfall

## Zusammenfassung 2

- Skalierung in: Größe, Geographie und Administration.
- Techniken: Verstecken von Latenz, Partitionierung, Replikation und Caching.
- 'Fallacies of distributed systems' erkennen!
- Verteilte Systeme: Cluster-, Grid-, Cloud-, Ubiquitous-, Mobile Computing, verteilte Informationssysteme, Sensornetze

## Fallacies

Um sie zu erkennen.

## One Administrator

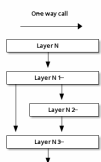
- Wo läuft „das System“ — wer kontrolliert die Rechner?
- Gelten für alle die gleichen Gesetze?
- Welche Regeln gelten für Kommunikation?<sup>6</sup>
- Gibt es mehr als eine Implementierung?
- ...?

<sup>6</sup>Kontext: Solving the Moderator's Trilemma with Federation

## Ziele

- Sie kennen verbreitete Architekturstile
- Sie verstehen das Konzept von Overlay-Netzwerke

## Layered | Schichten



- Calls gehen (meist) nur in eine Richtung
  - Bsp für Ausnahme: Callback in async IO

## Sensornetze

- Viele Sensoren (bis zu mehreren Tausend)
- Teilweise Grundlage für ubiquitous computing
- Arbeiten zusammen für effiziente Verarbeitung der Daten
- Meistens drahtlos und batteriebetrieben (leicht aufzusetzen)
- Energieverbrauch und Kommunikation minimieren! WLAN ist hier teuer.

## Fallacies of distributed Systems

- 1 The network is reliable
- 2 The network is secure
- 3 The network is homogeneous
- 4 The topology does not change
- 5 Latency is zero
- 6 Bandwidth is infinite
- 7 Transport cost is zero
- 8 There is one administrator

## Fallacies of distributed Systems

Nochmal?? Meinst du das ernst?? Ja :-)

- 1 The network is reliable
- 2 The network is secure
- 3 The network is homogeneous
- 4 The topology does not change
- 5 Latency is zero
- 6 Bandwidth is infinite
- 7 Transport cost is zero
- 8 There is one administrator

## Fallacies of Consumer-Level Services

- 1 The harddisk is reliable
- 2 Power is constant
- 3 Your IP is reachable
- 4 Constant factors are negligible
- 5 Libraries are stable<sup>7</sup> and API's are maintained
- 6 Textfiles are simple, the protocol is stable, and the dataformat is fixed

<sup>7</sup>Volatile Software:

<https://stevelosh.com/blog/2012/04/volatile-software/>

## Merkmale von Architekturstilen

- Verwendete Komponenten und ihre Schnittstellen.
- Verbindung zwischen Komponenten (RPC, Messaging).
- Daten, die zwischen Komponenten ausgetauscht werden.
- Konfiguration der Komponenten zu einem System

Beispiel: Web, Backend, Datenbank.<sup>8</sup>

<sup>8</sup>Disy Cadanza ist (inzwischen) ein klassischer Vertreter. Mit (wenig) Server-side Rendering) für Startzeit.

## Zusammenfassung

- Verteilte Systeme sind eine Sammlung autonomer Knoten, die als einzelnes kohärentes System erscheinen.
- Verteilte Systeme versuchen zu bieten:
  - Teilen von Ressourcen
  - Verteilungstransparenz
  - Offenheit
  - Skalierbarkeit
- Verteilung hat immer einen Preis

## PAUSE

## PAUSE

## Transport cost is zero

The two most expensive operations in terms of cost were the orchestration workflow and when data passed between distributed components.

...

Moving our service to a monolith reduced our infrastructure cost by over 90%. It also increased our scaling capabilities.  
— Marcin Kolny, 2023, für Amazon Prime Video

## Architekturen verteilter Systeme

Aus der Vogelperspektive.

## Architekturstile

- Layered | Schichten
- Object-based | Objektbasiert
- Resource-centered | Ressourcenzentriert | wie REST
- Event-based | Ereignisbasiert

Stile meist kombiniert.

## Object-based | Objektbasiert



- losere Organisation der Komponenten
- Kommunikation durch z.B. RPC
- Zustand meist in seiner Komponente/Objekt gekapselt
- Service statt Objekt: SOA (Service oriented architecture, jetzt „Microservices“)<sup>9</sup>

<sup>9</sup>Capabilities im Netz:  
<https://spritelyproject.org/>

## Verschiedene Arten von Koordination

- **Komponenten** kommunizieren durch Events
- Events durch **Event Bus** propagiert
- Komponenten referentiell entkoppelt (⇒ kein shared memory)
- persistente Speicherung der Events führt zu temporaler Entkoppelung
- Verwandte Events als topic abonnieren

### Zweischichtige Konfiguration

- 
- The diagram illustrates three architectural models:
- Client/Server:** A User Interface connects to an Application, which in turn connects to a Database.
  - Thin Client:** A User Interface connects to an Application, which connects to a Database. The User Interface and Application are shown as separate boxes, while the Database is a single box.
  - Web-based:** A User Interface connects to an Application, which connects to a Database. The User Interface and Application are shown as separate boxes, while the Database is a single box.



- Arne Babenhauer/ehd
- Einführung und Grundlagen Verteilte Systeme
- Einstieg Einführung Fallacies Architekturen Prozesse Kommunikation Praxis Fin
- Zentralisierte System-Architekturen

## Übung 2/2: Randbedingungen: Latenz Client-Server

- A: 1ms (Lokaler Rechner, effizient<sup>9</sup>)
- B: 10ms (Lokaler Rechner, effizient aber mit Compositor, Regionaler Spiele-Ping, Typische USB-Tastatur)
- C: 100ms (Lokaler Rechner<sup>10</sup>, Spiele-Ping: 20.000km)
- D: 1s (Neue Webseite öffnen)
- E: 2.5 Minuten bis 5 Minuten (Mars) 50–100 mio km
- F: 3 Wochen (Delay Tolerant Networking — Nomaden)

Arne Babenhauseide

Einführung und Grundlagen Verteilte Systeme

Einstieg Einführung Fallacies Architekturen Prozesse Kommunikation Praxis Fin

Dezentralisierte System Architekturen (o2p)

## Flooding

- 

- Algorithmus:
  - Ein Knoten erhält eine Anfrage für einen Wert.
  - Der Knoten sucht lokal nach dem Wert.
  - Findet er ihn nicht, übergibt er die Anfrage an **alle** Nachbarn.
  - Das Spiel wiederholt sich.
- Kann hohe Last erzeugen.
- Anfragen mit einer Time-To-Live (TTL).
  - z.B. maximale Anzahl von Sprüngen

Arne Babenhauzerheide

Einführung und Grundlagen Verteilte Systeme

Einstieg Einführung Fallacies Architekturen Prozesse Kommunikation Praxis Fin

Prozesse

### Prozesse: Ziele

- PAUSE

- Sie verstehen den Unterschied zwischen Prozessen und Threads

## Threads

- Betriebssystem sorgt für Isolation zwischen Prozessen:
  - Eigene Speicherbereiche.
  - Unerlaubte Zugriffe (durch einen anderen Prozess): segfault.
- Kommunikation über Message passing.
  - Einfachste: Unix pipe.

- Threads sind nebenläufig:  
führen Code unabhängig von  
anderen Threads aus.

- **Stack:** Threads führen minimalen Kontext mit sich (Java: 1MiB Stack => -Xss512k).
- **Shared Memory:** Threads im gleichen Prozess können auf den gleichen Speicher zugreifen.
- **Kontextwechsel, Erzeugen und Zerstören von Threads** ist billiger.
- **Im Kernel- oder Userspace**



## Lightweight Threads: fibers/goroutines/virtual threads/...

- Threads in Userspace
- Millionen von Threads, fast gratis wenn Inaktiv
- Oft explizite Kommunikation
- Skynet Benchmark: 1 Million Threads erstellen und als Baum Kommunizieren in <https://github.com/atemerev/skynet>
  - Eigenes Beispiel: <https://github.com/atemerev/skynet/blob/master/guile-fibers/skynet.w>
- Java: Project Loom:  
<https://openjdk.java.net/projects/loom/>

## Zusammenfassung

- Prozesse werden vom OS isoliert: Messages.
- Threads teilen Speicher.
- Threads sind günstiger zu switchen.
- Threads in Kernel- oder Userspace implementiert.

## Synchronization points

Worauf Sie warten können.

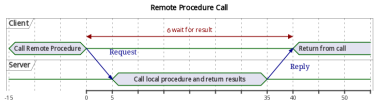


## Middleware Schicht

- Middleware soll allgemeine Dienste und Protokolle bereitstellen:
  - Kommunikation
  - (Un)marshalling (binär  $\Rightarrow$  Datenstruktur)
  - Namensprotokolle: Teilen von Ressourcen
  - Sicherheit
  - Skalierung: Replikation/Caching
- Die Implementierung eines verteilten Systems kann sich auf das Anwendungsprotokoll konzentrieren.

## Remote Procedure Call (RPC)

- Soll möglichst wie ein normaler Methodenaufruf aussehen.
- $\Rightarrow$  Zugriffstransparenz



## Risiko: unpassende Garantien

- Zu viel garantiert: viel Synchronisierung, langsam
- Zu wenig garantiert: Bugs

Wir kommen in einem späteren Block mit CALM und CRDTs darauf zurück.

## Lightweight Threads: Glossar

- Fiber Kooperativ statt Präemptiv.
- Green Thread Braucht keine OS-Unterstützung.
- Coroutine Funktion mit yield statt return.

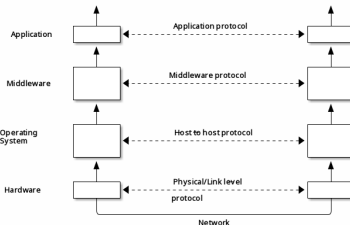
## Kommunikation

Interprocess communication is at the heart of all distributed systems.

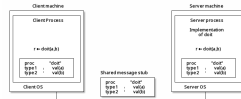
## Arten der Kommunikation

- transient vs. persistent
  - transient: Nachricht wird verworfen, falls sie nicht ausgeliefert werden kann.
  - persistent: Nachricht wird gespeichert bis sie übermittelt wurde.
- asynchron vs. synchron
  - asynchron: Sender fährt nach Übergabe der Nachricht an Kommunikationsstack fort.
  - synchron: Sender wird geblockt bis Nachricht übermittelt wurde.

## Middleware Schicht 2



## Ablauf RPC



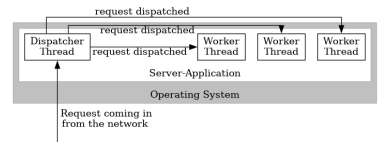
- 1 Client ruft stub auf.
- 2 Stub erstellt Nachricht.
- 3 Client OS sendet Nachricht.
- 4 Server OS ruft stub auf.
- 5 Stub entpackt Nachricht.
- 6 Stub erhält Ergebnis.
- 7 Stub erstellt Nachricht.
- 8 Server OS sendet Nachricht.
- 9 Client OS gibt an stub.
- 10 Stub entpackt Nachricht.

## Zusammenfassung

- Middleware als Schicht im Netzwerkmodell gesehen
- Kommunikation nach transient/persistent und asynchron/synchron unterschieden
- RPC ist ein entfernter Methodenaufruf
- Gibt Garantien.

## Dispatcher/Worker Modell

- Ein Thread (Dispatcher) liest eingehende Requests
- Die Request wird an Worker Thread gegeben, der die eigentliche Arbeit erledigt.



## Ziele

- Sie erkennen, wie Middleware als Schicht fungieren kann
- Sie kennen RPC (Remote-Procedure-Calls)
- Sie erkennen Messaging-Strukturen
- Sie können Kostenmetriken für Overlays rechnen

## Schichtmodelle

Definition von OSI:

- Schicht bietet der darüberliegenden Schicht einen Dienst an.
- Dieser Dienst wird durch eine Schnittstelle spezifiziert.
- Schichten verwenden jeweils ein Protokoll um mit der Gegenseite (auf der gleichen Schicht) zu kommunizieren.
- Beim Senden einer Nachricht wird die Nachricht an die darunterliegende Schicht gereicht.

Anschauliche Kurzbeschreibung: [osi-model.com/](http://osi-model.com/)

## Middleware in Cadenza

Backend, physisch verteilt

- Apache Ignite shared-nothing Clustering
- Präzise Cache-invalidation durch explizite Nachrichten

Frontend, logisch verteilt

- Redux store zur Synchronisierung und für Events
- $\Rightarrow$  PATCH statt PUT/POST

## Parameter Marshalling

- Client und Server haben evtl. unterschiedliche Datenrepräsentation (z.B. Little vs. Big Endian)
- Parameter in Bytes umwandeln
- Auf Formate einigen
- Komplexe Datenstrukturen?
- Referenzen?
- Änderungen kommunizieren?
- Nie völlig transparent?

Referenzen

- Post-Messages zwischen iframes: Keine Referenzen
- OCap: Mögliche Referenzen als explizite Capabilities. Entrance to the rabbit hole: <https://fosdem.org/2021/schedule/event/spritelygoblins/> (down the ASCII rabbit hole)

## PAUSE

PAUSE

## Praxis: Ziele

- Sie kennen grundlegende Beispiele zum schnellen Einstieg
- Sie kennen Kostenmetriken für Overlays

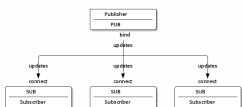
## Messaging mit ZeroMQ

- Sockets mit definierten Kommunikationsmustern.
- Sockets erlauben one-to-one
- ZeroMQ bietet auch many-to-one und one-to-many:
  - request-reply
  - publish-subscribe
  - pipeline

## Einwurf: Was bringt mir das?

- Verständliche(-re) Muster  $\Rightarrow$  leichter zu kombinieren
- Kommunikations-Struktur erzwingen  $\Rightarrow$  Fehlererkennung

## Publish-Subscribe mit ZeroMQ



- Implementiert Multicast
- Subscribers bekommen nur Messages entsprechend ihrer Subscription.

## Pipeline mit ZeroMQ



- Erlaubt Verteilung von Arbeit auf mehrere Consumer
- Im Gegensatz zu Publish-Subscribe wird jede Message nur an einen Consumer weitergegeben

## Message Broker



- bestehende Systeme integrieren
- Für jedes System eigenes Messageformat
- Nachricht von Prozess A an B muss Bs Protokoll nutzen
- Konvertierung von Nachrichten durch Komponente

## Einfacher RPC Server in Python

```
from xmlrpc.server import SimpleXMLRPCServer, SimpleXMLRPCRequestHandler

server = SimpleXMLRPCServer(("localhost", 8001))

def hi():
    return "hi"

server.register_function(hi)
server.serve_forever()
```

## Request-Reply mit ZeroMQ - Server

```
import zmq

context = zmq.Context()
p = "tcp://127.0.0.1:8001"
s = context.socket(zmq.REP)

s.bind(p) # bind = listen here

while True:
    message = s.recv_string()
    if not "STOP" in message:
        s.send_string(message + "*")
    else:
        break
```

## Publish-Subscribe mit ZeroMQ - Publish

```
import zmq, time

context = zmq.Context()
p = "tcp://127.0.0.1:8001"
s = context.socket(zmq.PUB)

s.bind(p)

while True:
    time.sleep(3)
    s.send_string("TIME " + time.asctime())
```

## Pipeline mit ZeroMQ - Producer

```
import zmq, time

context = zmq.Context()
s = context.socket(zmq.PUSH)
p = "tcp://127.0.0.1:8001"

s.bind(p)

for i in range(100):
    s.send_string(str(i))
    time.sleep(0.1)
```

## Message-oriented persistent communication

- „message-queuing systems“ oder „Message-Oriented Middleware (MOM)“
- **Persistente** asynchrone Kommunikation.
- Sender und Empfänger müssen nicht gleichzeitig aktiv sein.
- Wie E-Mail.

## Kostenmetriken für Multicast

Den Preis der Abstraktion quantifizieren.

## Einfacher RPC Client in Python

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy("http://localhost:8001")
print(s.hi())

Wireshark-Time!
```

## Request-Reply mit ZeroMQ - Client

```
import zmq

context = zmq.Context()
p = "tcp://127.0.0.1:8001"
s = context.socket(zmq.REQ)

s.connect(p)

s.send_string("hi")
message = s.recv_string()
print(message)
s.send_string("hi2")
# violate request-reply => error!
s.send_string("too hasty")
message = s.recv_string()
s.send_string("STOP")
print(message)
```

## Publish-Subscribe mit ZeroMQ - Subscribe

```
import zmq

context = zmq.Context()
s = context.socket(zmq.SUB)
p = "tcp://127.0.0.1:8001"

s.connect(p)

s.setsockopt_string(zmq.SUBSCRIBE, "TIME")

for i in range(5):
    time = s.recv_string()
    print(time)
```

## Pipeline mit ZeroMQ - Consumer

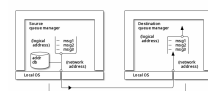
```
import zmq, time

context = zmq.Context()
s = context.socket(zmq.PULL)
p = "tcp://127.0.0.1:8001"

s.connect(p)

while True:
    message = s.recv_string()
    print(message)
    time.sleep(int(message) * 0.01)
```

## Allgemeiner Aufbau

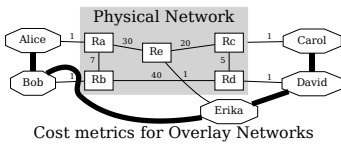


- Prozess hat einen lokalen Queue-Manager.
- Queue Manager verwaltet die Queue für den Prozess.
- Prozess kann in die lokale Queue Messages einstellen und entnehmen.
- Messages adressiert.
- Adressdatenbank im Queue-Manager.

## Multicast in der Anwendungsschicht

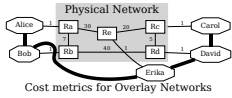
- Ziel: Daten an mehrere Empfänger
- Multicast auf IP Ebene zwischen ISPs selten umgesetzt  $\Rightarrow$  Multicast in der Anwendungsschicht  $\Rightarrow$  Abstraktions-Overhead
- Overlay Netzwerk
  - Oft Baum: Pfade sind eindeutig
  - Mesh-Strukturen benötigen Routing

## Metriken für Multicast mit Overlay



- Kosten für physische Verbindungen sind gegeben.

## Übung



Berechne Link Stress und Stretch für die Verbindung Erika zu Carol.

- Link Stress: (Rd, D) (Rc, Rd) mit Link Stress jeweils 2
- Stretch:  $1 + 20 + 5 + 1 + 1 + 5 + 1 = 34$ ,  $1 + 20 + 1 = 22$   
→  $34/22$

## Gesamtzusammenfassung 2

- **Architektur:** Layered, Object, Resource, Event
- Schichten und Overlay Netze
- Prozesse sind isoliert, Threads teilen Speicher.
- **Middleware** als Schicht: Übernimmt Verteilung, gibt Garantien.
- **Messaging:** Request-Reply, Pub-Sub, Pipeline.
- **Overlay metriken:** Link Stress und Stretch

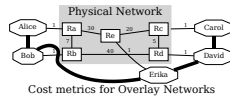
## Verweise

Ghosh, S. (2015). *Distributed Systems - An Algorithmic Approach*. Computer & Information Science. Chapman & Hall/CRC, 2 edition.

Steen, M. v. and Tanenbaum, A. S. (2017). *Distributed Systems*. CreateSpace Independent Publishing Platform; 3.01 edition (February 1, 2017), 3 edition.

Bilder:

## Link Stress für Multicast mit Overlay



Link Stress: Anzahl wiederholter Nutzungen einzelner Verbindungen.

Beispiel Link Stress von Alice zu Erika: (B, Rb), (Ra, Rb) mit Link Stress jeweils 2.

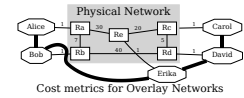
## Zusammenfassung

- Messaging erlaubt entkoppelte Kommunikation
- ZeroMQ bietet die Kommunikationsmuster:
  - Request-Reply
  - Publish-Subscribe
  - Pipeline
- Multicast:
  - Wird auf Anwendungsebene umgesetzt
- Link Stress und Stretch als Metriken für Overlay Netzwerke

## Fallacies of distributed Systems

- |                        |                    |
|------------------------|--------------------|
| 1 reliable             | 1 hard disk        |
| 2 secure               | 2 power            |
| 3 homogeneous topology | 3 IP               |
| 4 latency              | 4 constant factors |
| 5 bandwidth            | 5 APIs             |
| 6 transport cost       | 6 text             |
| 7 administrator        |                    |

## Stretch für Multicast mit Overlay



Stretch: Verhältnis aggregierter Kosten auf dem Weg im Overlay und dem optimalen Kommunikationsweg.

Beispiel Stretch von Alice zu Erika: Overlay =  $1 + 7 + 1 + 1 + 7 + 30 + 1 = 48$ , Optimal =  $1 + 30 + 1 = 32$  → Stretch  $48/32$

## Gesamtzusammenfassung 1

- Sammlung autonomer Knoten, die als ein kohärentes System erscheinen.
- **Ziele:** Ressourcen, Verteilungstransparenz, Skalierbarkeit
- **Skalierung:** Größe, Geographie, Administration
- Latenz, Partitionierung, Replikation, Caching
- **Fallacies!**
- Cluster, Grid, Cloud, Ubiquitous, Mobile, DIS, Sensornetze

## Fin

Viel Erfolg in den nächsten Wochen!