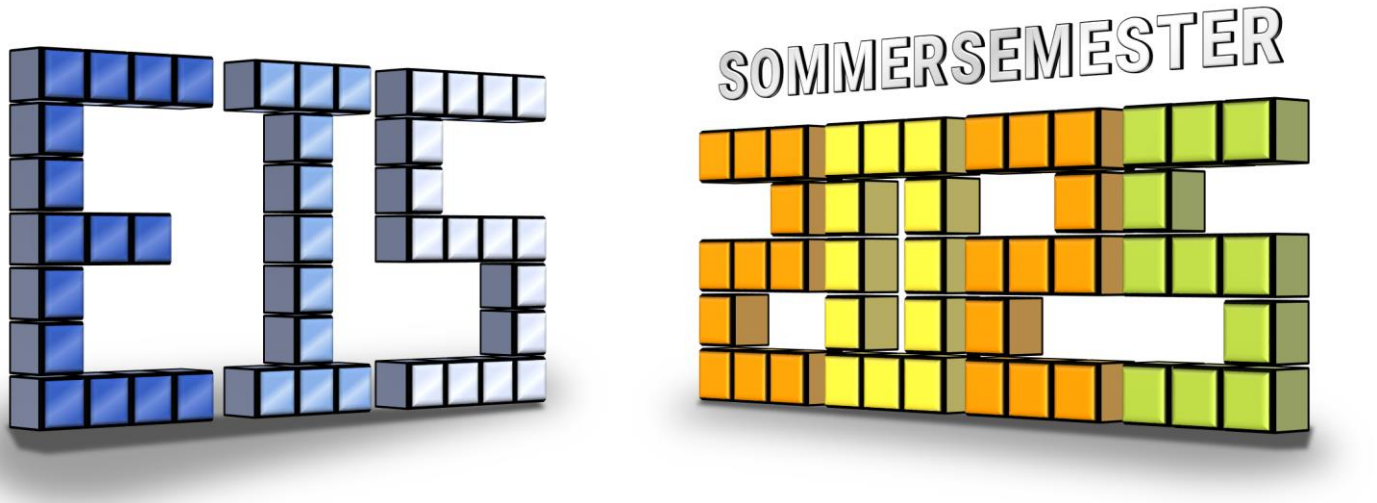
EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2024



Foliensatz #11

Parallele Programmierung

Michael Wand
Institut für Informatik
Michael.Wand@uni-mainz.de

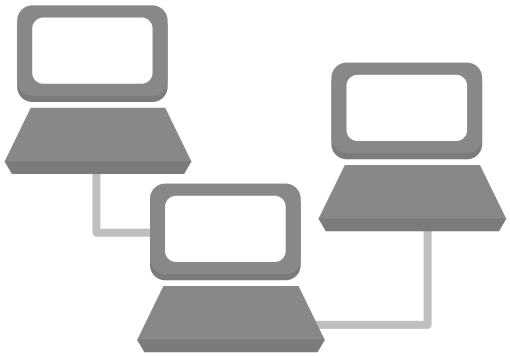
Parallele Programmierung

Übersicht

Themen

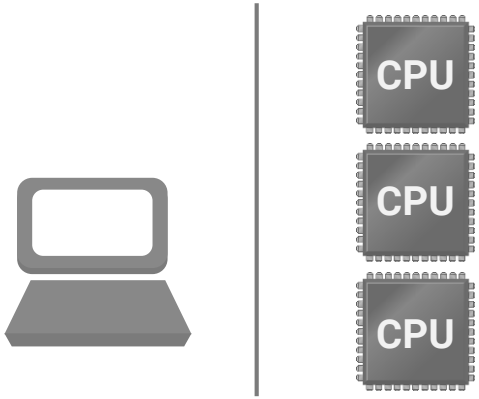
- Technischer Hintergrund: Parallele Architekturen
- Parallele Herausforderungen
  - Deadlocks!
  - Race Conditions!
- Synchronisationsprimitive
- Architekturmuster
  - Traditionelle Client-Server Architektur
  - Moderne „Asynchrone“ Programmierung

Parallele Architekturen



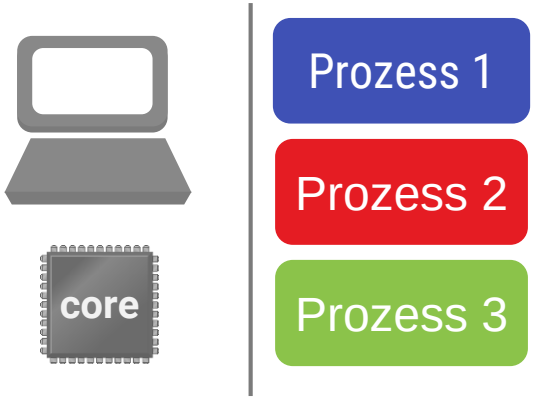
Distributed System („verteilt“)

Mehrere Rechner



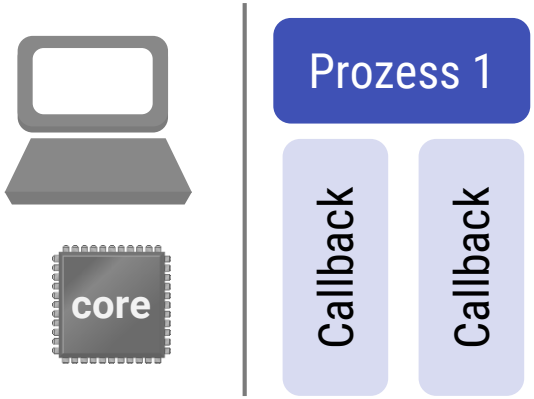
Shared-Memory System

Ein Rechner, mehrere CPUs



Präemptives Multi-Tasking

Eine CPU, mehrere Tasks



Kooperatives Multi-Taksing

Eine CPUs, ein Task

Wie funktioniert es?

Verteiltes System

- Netzwerk (Ethernet, Fiberchannel, InfiniBand, etc.)
- Kommunikation:
  - Nachrichten schicken („message passing“)

Shared Memory

- Gemeinsamer Hauptspeicher
  - I.d.R. Cache-koheränt (keine explizite Synchronisation nötig)
- Kommunikation
  - Gemeinsamen Speicher lesen/schreiben
  - Semaphoren / mutexes
  - Nachrichten schicken (simuliert in Software)

Wie funktioniert es?

Präemptives Multi-Tasking

- Automatisches Umschalten
  - Typisch: alle 20-50ms (20-50x pro Sekunde)
  - Keine echte Parallelverarbeitung
  - Aber es sieht für den Nutzer so aus
- Zwei Modelle für Multi-Tasking
  - „Multi-Threading“: gemeinsamer Hauptspeicher
  - Mehrere Prozesse: getrennter Speicher (simuliert)
    - Jeder Prozess enthält i.d.R. mehrere Threads
- Kommunikation
  - Gemeinsamen Speicher lesen/schreiben
  - Semaphoren / mutexes
  - Nachrichten schicken (simuliert in Software)

Wie funktioniert es?

Kooperatives Multi-Tasking

- Explizit programmiertes Umschalten
  - Wenn man Code schreibt, muss man ihn selbst in Häppchen aufteilen und umgschalten
- Architektur
  - Callbacks
  - Event-driven programming
  - Genau wie bei den meisten GUIs! (→ Eventqueues)
- Kommunikation:
  - Gemeinsamen Speicher lesen/schreiben
  - Callbacks / Events „posten“
    - Neues Ereignis in Ereigniswarteschlange

Multi-Tasking: Begriffe

Präemptiv / Multi-Core / Multi-CPU

- Nebenläufiger Kontrollfluss (Programm)
- Task: Oberbegriff
  - Threads: gemeinsamer Adressraum
  - Prozesse: getrennter Adressraum
- Tasks werden vom OS automatisch auf die verfügbaren CPUs/Cores verteilt

Parallele Herausforderungen

Nebenläufigkeit ist schwierig

- Parallele Algorithmen
  - Nicht jedes Problem kann man parallel lösen
  - Unterschiedlich schwer (Vorlesung „Parallele Algorithmen“)
- Synchronisationsprobleme
  - „Race-Conditions“ – inkonsistenter paralleler Datenzugriff
  - „Deadlocks“ – zyklisches Warten

Beispiel: Drucker-Spooler

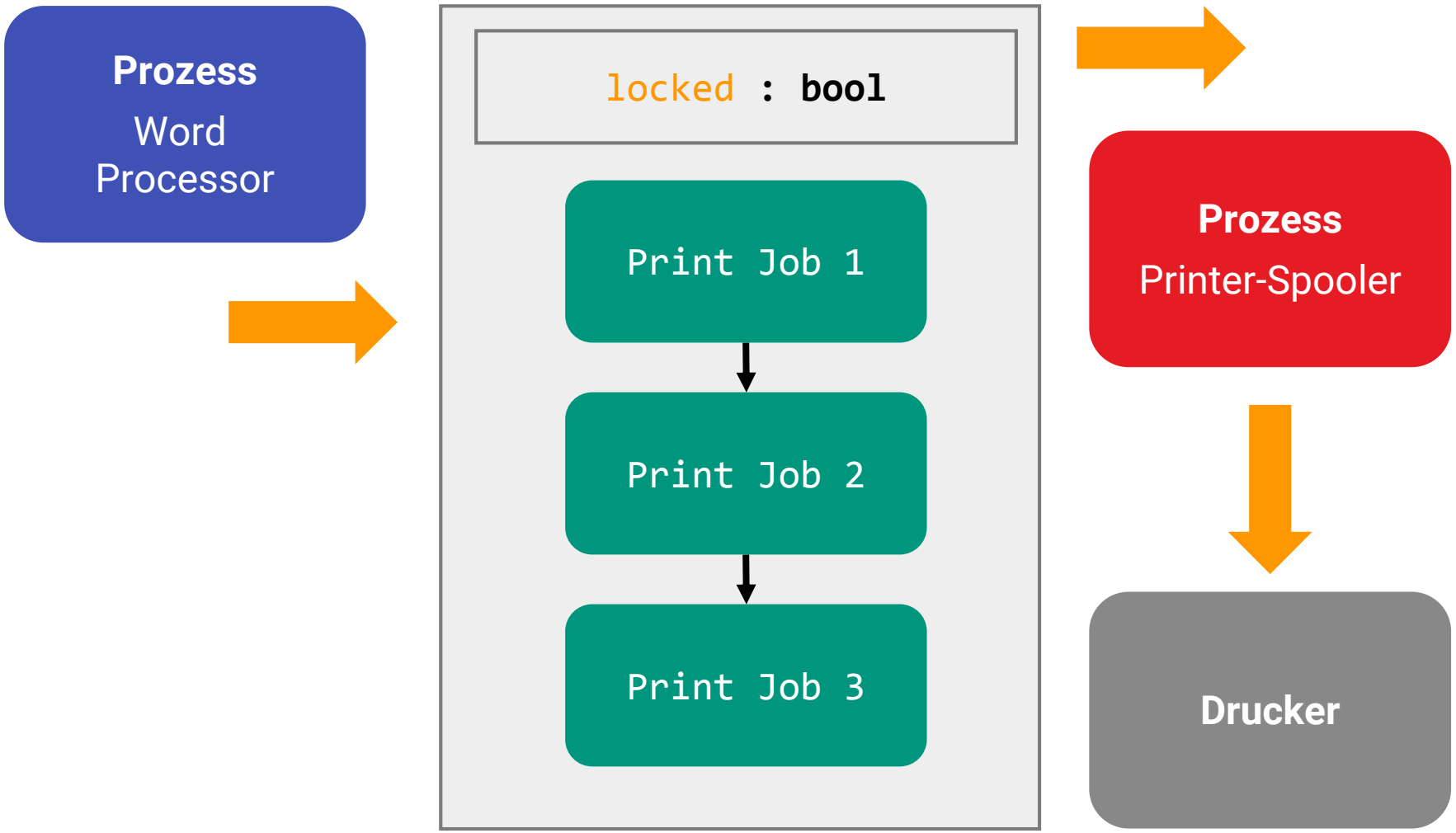
Zwei Prozesse

- Einer produziert Dokumente
- Ein zweiter schickt die zum Drucker
  - Man kann weiterarbeiten, während gedruckt wird

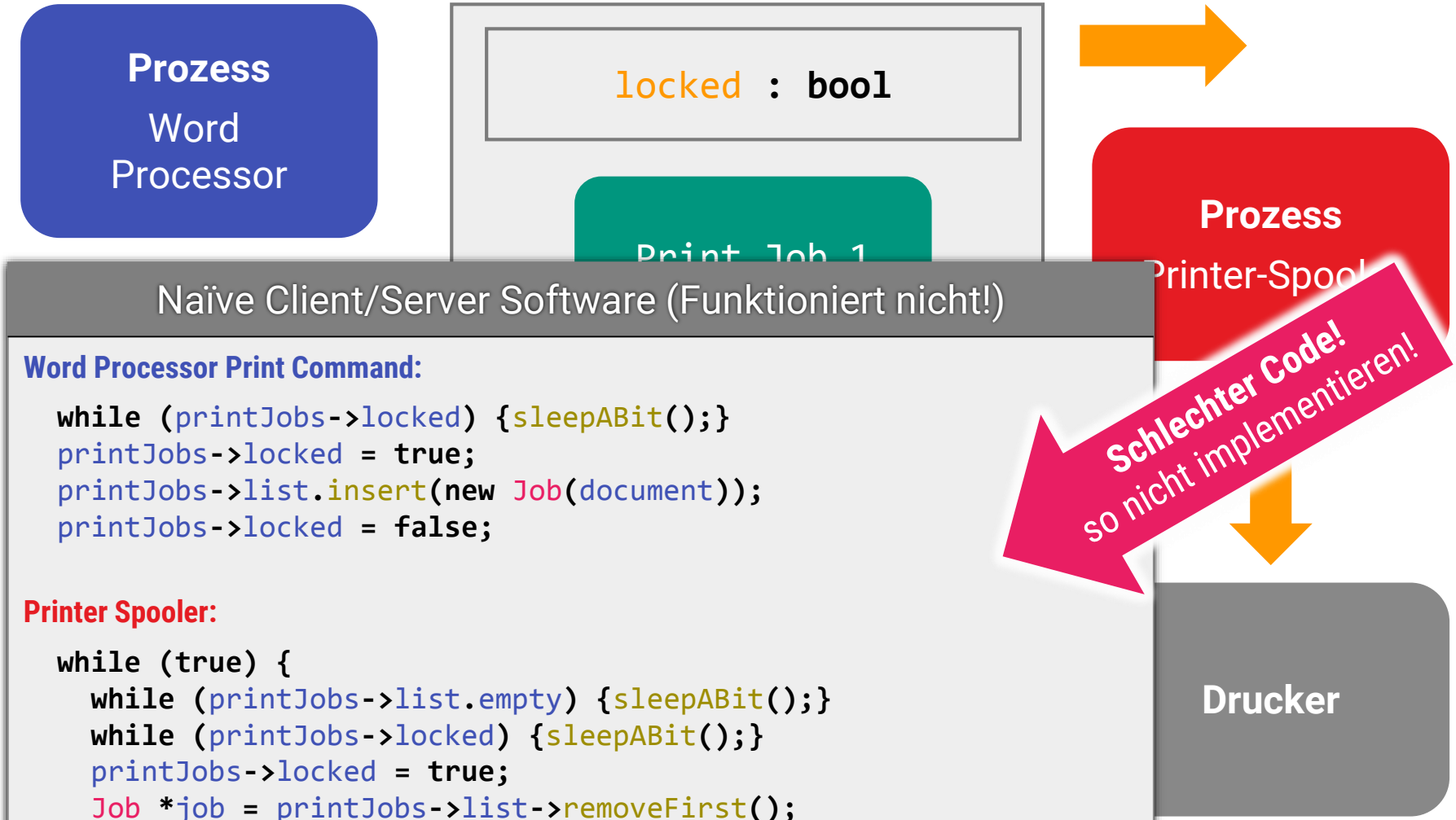
Implementation

- Verkettete Liste von Druckerjobs
- Gemeinsamer Speicher (einfachste Situation)

Race Conditions



Race Conditions



Naïve Client/Server Software (Funktioniert nicht!)

Word Processor Print Command:

```
while (printJobs->locked) {sleepABit();}
printJobs->locked = true;
printJobs->list.insert(new Job(document));
printJobs->locked = false;
```

Printer Spooler:

```
while (true) {
  while (printJobs->list.empty) {sleepABit();}
  while (printJobs->locked) {sleepABit();}
  printJobs->locked = true;
  Job *job = printJobs->list->removeFirst();
  printJobs->locked = false;
  printer->print(job);
}
```

Race Conditions

Problem: Reihenfolge der Zugriffe

- Inkonsistente Zustände möglich
- Abfrage und Schreiben von Variable „`locked`“ kann unterbrochen werden

Lösung

- Atomare Operation „`getLockIfPossible()`“
- Atomare Operation „`releaseLock()`“
- Atomar: Während Ausführung kein Zugriff durch nebenläufigen Vorgang
- Hardwareunterstützung (alle moderenen CPUs)
  - Softwarelösung möglich (eher akademisch, Peterson‘s solution)

Synchronisationsprimitive

Fertige Komponenten (moderne OS)

- Mutex – atomare „Lock“
- Semaphore – atomare Zähler
  - Warten auf „mindestens 1 Dokument im Druckerspooler“
- Shared Memory
  - Zwischen mehreren „threads“: gesamter Speicher
  - Zwischen „Prozessen“: Muss vom OS angefordert werden
- Message Queues
  - Netzwerk (z.B. tcp/ip ports)
  - Simulation von Nachrichten auch bei shared memory

Bei allen: Warten kostet **keine** Rechenzeit (Task schläft)

Architekturen / Pattern

Architekturmuster

Beliebte Architekturmuster

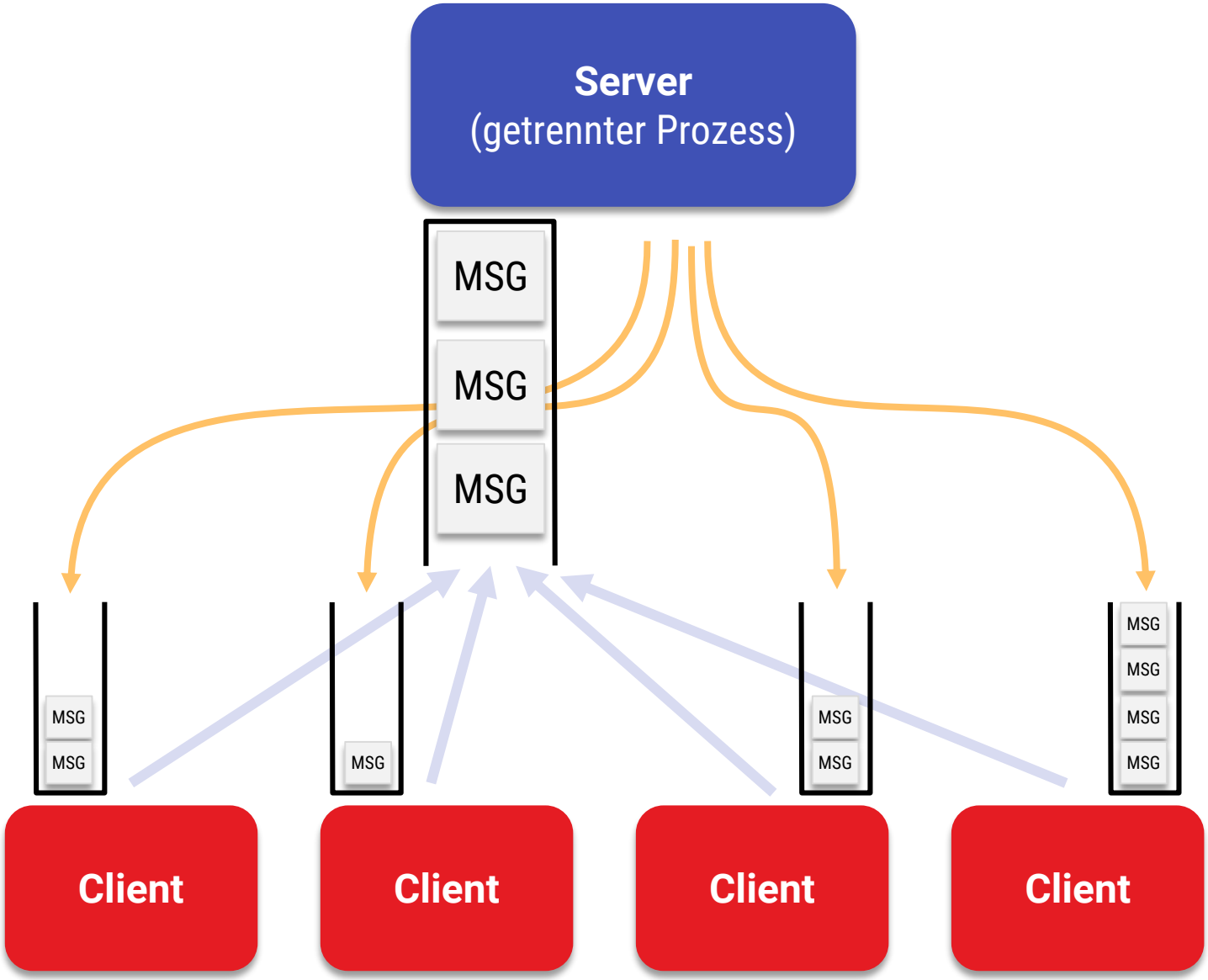
- Parallele Threads mit „shared memory“
  - Gesichert über Mutexe
- „Monitor“-Pattern
  - Objekt im Speicher, das nur von einem Task „betreten“ werden darf
  - Gesichert über Mutexe
  - JAVA „synchronized“ Methoden machen Objekt zum „Monitor“
- „Services“ / Server
  - Kommunikation via Nachrichten-Warteschlangen

(Client-) Server-Pattern

Struktur: Message-Queues

- Server (links) hat Nachrichtenwarteschlange
- Viele Clients können Nachrichten schicken
  - Jeder Client hat auch eine eigene Warteschlange
  - Server antwortet mit Nachrichten an Clients

(Client-) Server-Pattern



Server-Pattern

Eigenschaften

- Server kapselt eine „Ressource“
- Ordnung / Sortierung der Anforderungen über Warteschlange
  - Kein gemeinsamer Zugriff auf die gleiche Ressource
  - Nachrichtenverteilung intern
  - Parallelisierung / Lastverteilung möglich
- Funktioniert local wie auch in verteilten Systemen

Vorteile Server

Kapselung

- Zugriff auf Ressourcen wird arbitriert

Optimierungen

- Latenzen verdecken
  - Zugriff auf Geräte und Netzwerk oft mit hohen Latenzen
  - z.B. Festplatte (spinning-disk)
    ca. 15ms Latenz, 150MB/sec Durchsatz
  - z.B. Anweisung an GPU
    μsec Latenz, GB-TB/sec Durchsatz
- Mehrere nebenläufige Prozesse
  - Erhalten Nachrichten, wenn Ressource verfügbar

Server-Pattern

Beliebtes Muster

- Microservices im Internet, Web-Apps
- Data-Processing Software, z.B. Jupyter Notebooks
- Betriebssysteme: Microkernel-Architektur
  - Klassiker: „MINIX“, „AmigaOS“ mit „Libraries & Devices“
  - Aktuelle Beispiele: „L4“, „MACH“ (→ Mac OS/X)
    - Hybride Architekturen: „Darwin“ (Mac), WindowsNT Kern
  - „Services“ / „Devices“ kapseln
    nebenläufig arbeitende Ressource, z.B.
    - „Block-Device“ für Festplatte / Floppy-Disk
    - „FileSystem-Device“ für Dateisystem
    - „X-Server“, „Intuition-Process“ für GUI

Server-Pattern

Entwurfsprinzipien

- Parallele Prozesse mit verteiltem Speicher!
  - Grundsätzlich anspruchsvoll
  - „Profis nehmen doch Microservices“ – hat schon manches Projekt ruiinert
  - Wissen, worauf man sich einlässt
    - Mächtiges Werkzeug, wenn richtig genutzt
- Unabhängigkeit, Robustheit/Fehlertoloranz wichtig
  - Unabhängig testbar
  - Erwartet „fehlerhafte“ Eingaben, alte Protokollversionen, etc.
  - Scaling-Strategie: „Ein Team pro Microservice“
    - Globaler Entwurf vorher gut durchdacht (Bottleneck)

Nebenläufigkeit in Programmiersprachen

(aka. Diskussion Übungsaufgabe 7, ehemals **Programmierprojekt 2**)

Beispiel: Python

Python: Multithreading

- „Global Interpreter Lock“
  - Einzelne Befehle atomar
    - Multithreading wenig effektiv
  - Standardmodul `threading`

Python: parallele Prozesse

- Standardmodul `multiprocessing`*)
  - Klasse `Process` kapselt einen Prozess
    - Achtung: „Main-Guards“ erforderlich
  - Klasse `Pipe` kapselt eine Message-Queue
    - Python-Objekte direkt verschicken (intern via Pickle)

*) https://docs.python.org/3/library/multiprocessing.html

Beispiel: basic_io

Basic-IO

- Einfache Graphikbibliothek für „EIP“
  - Erlaubt synchrone Graphikausgabe via „`draw_ellipse`“ u.ä.
  - Erlaubt Abfrage von Tastatur/Maus ohne Events
    - „`get_current_mouse_position`“ statt „`mouseMoveEvent`“
- Technisch
  - Öffnet Qt-Fenster in extra Prozess
  - Kapselt Event-handling
- Client-Server Architektur
  - IO-Fenster ist ein Server
  - Kommunikation mit Anwendung (Client) über Nachrichten

Beispiel: basic_io

```
# Based on original source code, but slightly simplified

def __start_window_process() -> None:
    main_to_win_main_end, main_to_win_win_end = Pipe()
    win_to_main_win_end, win_to_main_main_end = Pipe()
    __connection_to_window = main_to_win_main_end
    __connection_from_window = win_to_main_main_end
    __window_process = Process(target=__window_process_loop, name="basicio_windowproc",
                               args=(main_to_win_win_end, win_to_main_win_end))
    __window_process.start()


def __window_process_loop(connection_client_to_window: Pipe, connection_window_to_client: Pipe) -> None:
    app = QApplication(sys.argv)
    win = IOWindow(connection_client_to_window, connection_window_to_client)
    win.show()
    app.exec()
```

Beispiel: basic_io

```
# Commands send to other processes (slightly simplified)
@dataclass
class IOMessage:
    msg_type: str
    params: dict


# Based on original source code, but slightly simplified
def draw_ellipse(x: int, y: int, radius_x: int, radius_y: int,
                 fill_color: Optional[RGBColor], border_color: Optional[RGBColor],
                 border_thickness: int) -> None:
    _try_post_msg(IOMessage(msg_type='draw_ellipse',
                            params={'x': x, 'y': y,
                                    'radius_x': radius_x, 'radius_y': radius_y,
                                    'fill_color': fill_color, 'border_color': border_color,
                                    'border_thickness': border_thickness
                            })
                  )

# Error handling removed (that was most of the code :-) )
def _try_post_msg(msg: IOMessage) -> None:
    can_post: bool = False
    while not can_post:
        __connection_to_window.send(msg)
```

Beispiel: basic_io

```
# Commands send to other processes (slightly simplified)
@dataclass
class IOMessage:
    msg_type: str
    params: dict


# Receiver side (simplified)
...
io_msg: IOMessage = self.connection_client_to_window.recv()
if io_msg.msg_type == 'draw_ellipse':
    painter: QPainter = ...
    x: int = io_msg.params['x']
    y: int = io_msg.params['y']
    radius_x: int = obj.params['radius_x']
    radius_y: int = obj.params['radius_y']
    painter.drawEllipse(x - radius_x // 2, y - radius_y // 2, radius_x, radius_y)
elif obj.msg_type == 'draw_text':
    ...
...
```

Beispiel: C++ und QT

Prozesse und Threads

- `QThread`
  - Abstrakte Methode
    ```
    protected: virtual int exec() = 0;
    ```
    enhält Code für Thread
  - Ableiten, um neue Thread-Objekte zu definieren
  - Methode `run()` startet Thread (von außen)
- `QProcess`
  - Kapselt neuen Prozess
  - Benötigt Dateinamen eines ausführbaren Programms (Windows Prozess Model ist an Dateien gebunden)
  - Message-Passing über Standard-Ein/Ausgabe möglich

Beispiel: C++ und QT

Synchronisationsprimitive in QT

- `QCriticalSection`
  - Methoden `lock()`, `unlock()`
- `QSemaphore`
  - Methoden `acquire()`, `release()`, `available()`
- Traditionelle message queues: `QLocalSocket`
- QT-Message queues mit Event-Modell
  - Abgebildet auf **`signals`** + **`slots`** (event-driven)
  - `QThread` kann **`signale`** + **`slots`** senden/empfangen
  - Jeder Thread hat eine eigene Event-Queue
  - Nur ein GUI-Thread erlaubt (Widgets sind nicht „thread-safe“)

Sockets: Message Queues für das Internet

(aka. Diskussion Übungsaufgabe 7, ehemals **Programmierprojekt 2**)

Verteilte Systeme

Kommunikation zwischen verschiedenen Rechnern

- Internet Protokoll („IP“)
  - Jeder Rechner hat eine IP-Adresse (z.B. „134.93.178.2“)
    - Unterschiede bei Version 4 vs. Version 6
  - Jeder Rechner kann bis zu 2^{16} (64K) Ports haben
  - Bidirektionale Punkt-zu-Punkt Verbindung (IP+Port zu IP+Port)
- Kommunikation auf gleichem oder zwischen verschiedenen Rechnern
  - Über Prozess- und Rechnergrenzen hinweg

TCP/IP vs. UDP/IP

Zwei Typen von IP-Sockets

- UDP – Packetorientiert
  - Einzele Datenpackete fester Größe
  - Ein Packet hat bis zu 64K Größe (minus Overhead)
  - Keine Garantien für Ankunft / Zeit / Reihenfolge
  - Beliebt für Games, Voice/Video u.ä.
- TCP – Datenstromorientiert
  - Serielle Datenverbindung
  - Garantierte Ordnung
  - verlorene Pakete werden automatisch nochmal angefordert
  - Höhere Kosten (Latenz, Overhead)
  - Standard für alles „nicht-Echtzeit“

Verteilte Systeme

Bibliotheken für IP-Kommunikation

- BSD-Sockets
  - Standardabstraktion für UNIX (Mac) & Windows
- Standardlibraries für C/C++
  - Linux/MacOS-X: `sys/socket.h`
  - Windows: `winsock2.h`
  - Cross-Plaform mit Qt: `QTcpSocket`
- Andere Sprachen
  - Python: Modul „`socket`“, Klasse `socket`
  - Java/Scala: `java.net.Socket`
- API im wesentlichen einheitlich

Verteilte Systeme

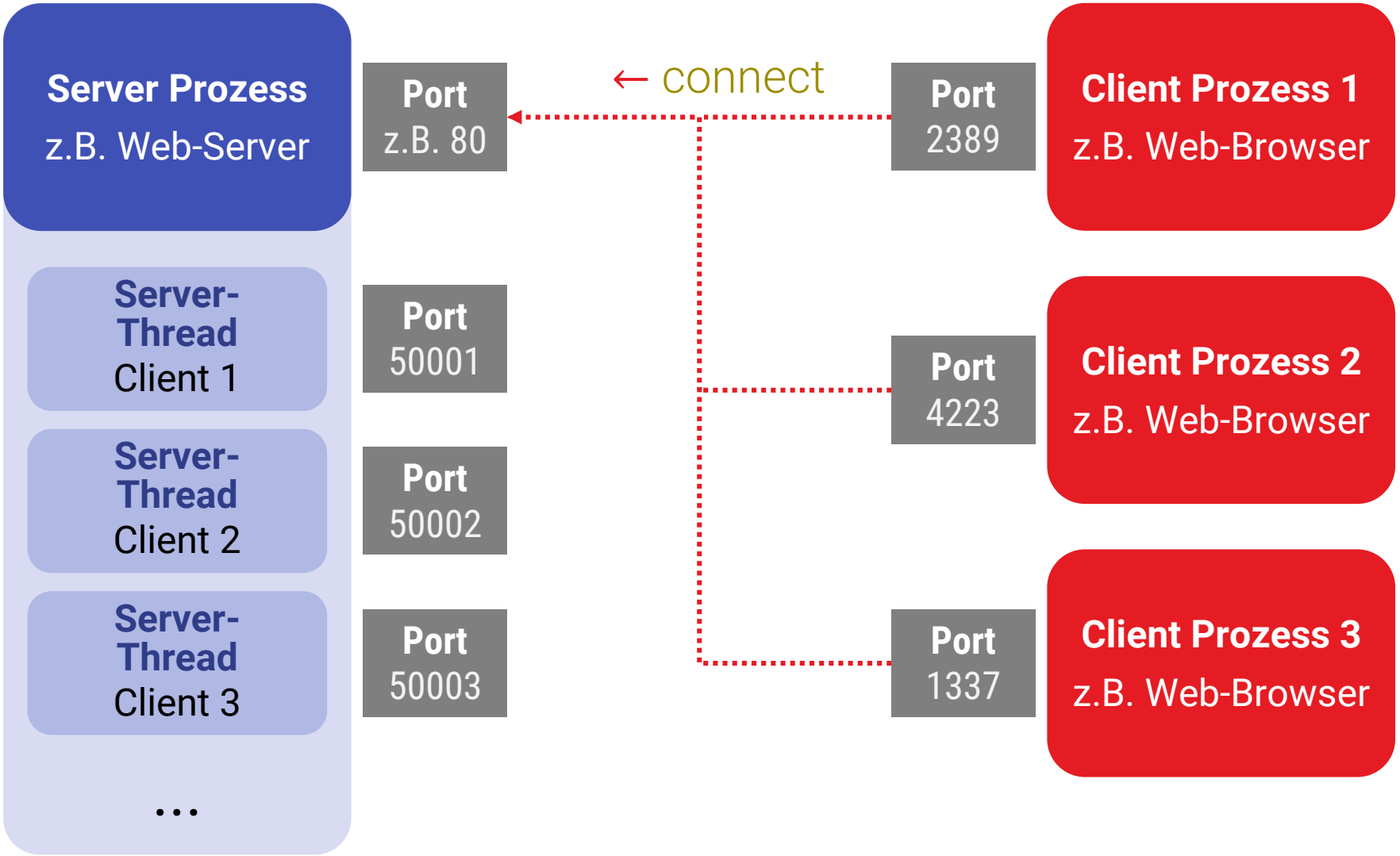
Server Kommandos

- `socket()` – erzeugt eine Socket
- `bind()` – setzt Parameter (Port, Protokoll, etc.)
- `listen()` – wartet auf Kontakt von außen (Task schläft)
- `accept()` – Verbindung akzeptiert: erzeugt neuen Socket!
- `read()`/`write()` – schickt/empfängt Daten (Binär)

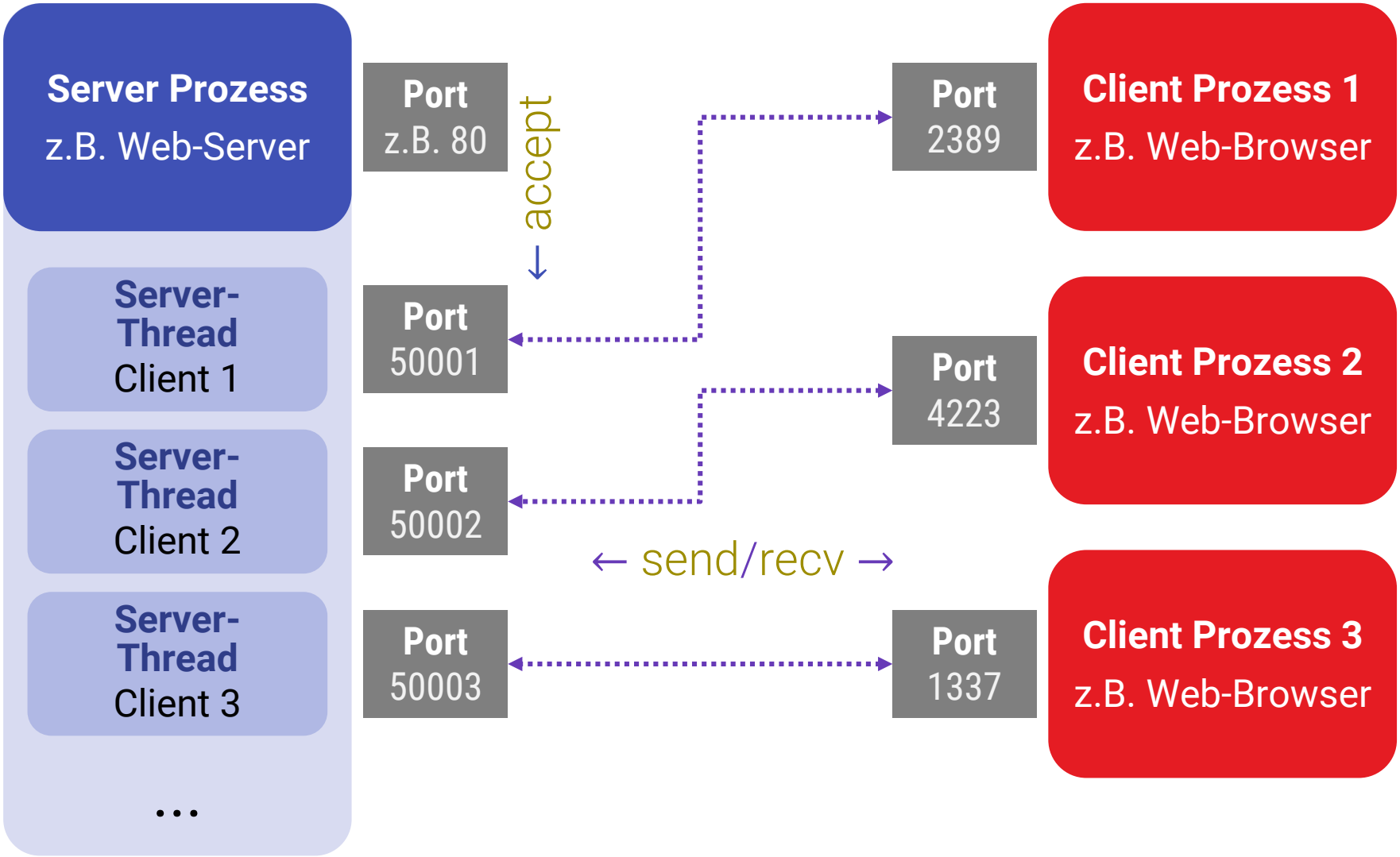
Client Kommandos

- `socket()` – erzeugt eine Socket
- `connect()` – Verbindet mit Server-Socket (setzt auch Par.)
- `read()`/`write()` – schickt/empfängt Daten (Binär)
- `close()` – schließt Verbindung (auch für Server)

Server Architektur



Server Architektur



Standard Client/Server Architektur

Server-Hauptprozess (Pseudo-Code!)

```
while (true) { // server runs forever
    socket = Warte auf Verbindung;
    Erzeuge neuen Thread für socket;
    Starte Thread (Parameter: Socket,
                   Sever-Daten);
}
```

Server-Thread (Pseudo-Code!)

```
while (true) {
    Warte auf Kommando(socket)
    falls commando-1:
      Empfange weitere Parameter;
      Hole Daten;
      Sende Daten an Client;
    falls commando-2:
      Ähliche Komunikation mit client;
    ...
    falls commando-ende: break;
}
Schließe Verbindung;
```

Server (Pseudo-Code!)

```
verbinde mit Server;

Schicke Kommando;
Schicke Paramter;
Schicke / Empfange Daten;

Schicke Kommando;
Schicke Paramter;
Schicke / Empfange Daten;

...

Schicke Ende-Commando;

Schließe Verbindung;
```

Standard Client/Server Architektur

Server-Hauptprozess (Pseudo-Code!)

```
Socket *sock
   = new Socket(80, TYPE_TCP_IP);
while (true) { // server runs forever
   sock.listen();
   Socket *clSock = sock.accept();
   ServerThread *newClientServ
      = new ServerThread(clSock, this);
   newClientServ->run();
}
```

Server-Thread (Pseudo-Code!)

```
while (true) {

   sock->read(command);

   if (command==FIRST) {send first}
   if (command==SECOND) {send second}
   if (command==END) {break;}

}

sock->close();
```

Client (Pseudo-Code!)

```
Socket *sock = new Socket();
sock->connect(
  80,
  nslookup("www.myserver.de"),
  TYPE_TCP_IP
);
sock->write(
  "GET /index.html HTTP/1.1\n“
  "Host: www.myserver.de\n");
sock->read(buffer);
displayPage(buffer)
```

Standard Client/Server-Architekur

Serverprozess

- Wartet auf Verbindungen
- Erzeugt (sofort) einen neuen Thread für jede Verbindung
- Jeder Thread bedient einen Client

Motivation

- Hohe Latenzen
  - Internet: 10ms–100ms realistisch
  - Klassische Festplatte: 10-20ms pro Zugriff
- Umschalten auf andere Threads um CPU auszunutzen

„Moderne“ Server

Es ist 2024…

Neue Welt

- Latenzen dramatisch gesunken
  - Schnelle I/O-Geräte
  - Schnelle (lokale) Netzwerke
- Tausende Verbindungen (Internet-Server)
  - Threads brauchen einige Ressourcen
  - Kann ein Bottleneck werden

(Zahlen von 2020)

Latenzen (IOPS = I/O Operationen/sec)

7200rpm HD: ca. 75-100 IOPS (10ms)

Consumer SSD: 100.000 IOPS (10µs)

Profi SSD-Array: bis zu 10.000.000 IOPS (100ns)

InifiniBand: Latenz ca. 1µs

Linux-Thread-Switch/IPC: ca. 1-10µs (ca. 10000 Takte)

- Experimente im Netz: Nachrichten via „Pipe“, Sockets, Semaphoren brauchen ca. 3-4µsec auf 2020er x86 CPU
- „Busy Waiting / Polling“ ca. 150nsec

Moderne Architektur

Moderne Server

- „Non-Blocking-IO“
  - Kein Warten auf Daten
  - Direkt nächsten Job anfangen
  - Alles in einem Thread
- Jeder Thread arbeitet mit vielen Clients
  - Event-Queues in jeden Thread
  - Ereignisse, falls neue Daten verfügbar werden

Beispiele

- Node.JS, Python `asyncio`, Boost `asio` (C++)

Zusammenfassung

Client-Server Architektur

- Nachrichtenaustausch zwischen Client und Server
  - Typischerweise Ereignis-orientierte Architektur in beiden
- Latenzen verstecken / multi-Core ausnutzen via Threading:
  - Server erzeugt neuen Thread für jede Anfrage
  - Thread „blocked“ bis Daten verfügbar (einfach zu prog.)
  - Schreiben auf „Datenbank“ via Mutex/Semaphoren gesichert
- Moderne Serverarchitekturen nutzen „non-blocking“-I/O
  - Effizienter bei niedrigen Latenzen bzw. sehr vielen Anfragen

Architekturmuster

Nur einige wenige Muster

- Klassische Client/Server-Architektur:
  - Threads erzeugen für Verbindungen
- Event-driven:
  - Ereignisse in Callbacks verwandeln
  - Verteilen an Komponentenbaum
  - Klassisches OOP-Design:
    - Eine Art von abstrakter „`processEvent`“-Methode
  - „Funktional“
    - Datenflussgraph (evtl. zyklisch = rekursiv)
  - Reactive programming
    - Events in Datenflussgraph

Moderner Server