Technische Hochschule Ingolstadt
Fakultät für Elektrotechnik und Informatik

Zukunft in Bewegung

Die Prozessicht

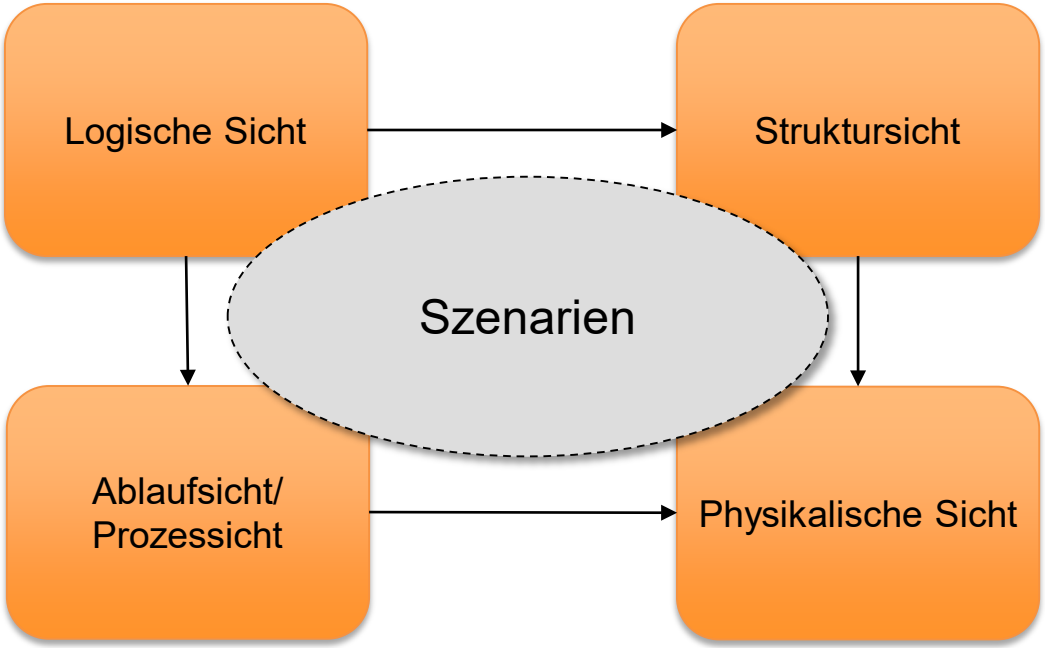
Architektur- und Entwurfsmuster der Softwaretechnik

Prof. Dr. Bernd Hafenrichter 06.03.2015

Die 4+1 Sichten der Softwarearchitektur

Es gibt keine allgemeine „Architekturdarstellung“.

Es müssen verschiedene Sichten (eines Systems) zu einer Gesamtarchitektur vereinigt werden



Philippe Kruchten, The 4+1 view model of architecture, IEEE Software, November 1995, 12(6), pp. 42-50

Prozesssicht Sicht der Architektur

Fokus:

Fokus: Abbildung des Produktmodells auf ein Verarbeitungsmodell.

Behandlung von Nebenläufigkeit und Synchronisation

Betrachtet werden folgende Aspekte

- Teilmenge der nicht-funktionalen Anforderungen
  - Performance
  - Availability
- Nebenläufigkeit
- Koordination/Synchronisation

Prozesssicht Sicht der Architektur

Grundlegende Eigenschaften bzw. Erkenntnisse

- Die Prozesssicht weist wenig Ähnlichkeiten mit der Code-Struktur auf
- Besteht aus einer vernetzten Menge kommunizierender Objekte
- Die Element sind Komponenten welche nur zur Laufzeit von Bedeutung sind (Prozesse, Threads, EJB‘s, Servlets, DLL‘s, Queues, …)
- Die Interaktion der Elemente variiert sehr stark abhängig von der verwendeten Technologie
- Diese Sicht ist wichtig für die Betrachtung qualitativer Merkmale wie z.B. Performance und Zuverlässigkeit
- Beim Entwurf der Prozessarchitektur sollte eine möglichst strikte Trennung zwischen der Technologie sowie der Anwendungsarchitektur selbst stattfinden

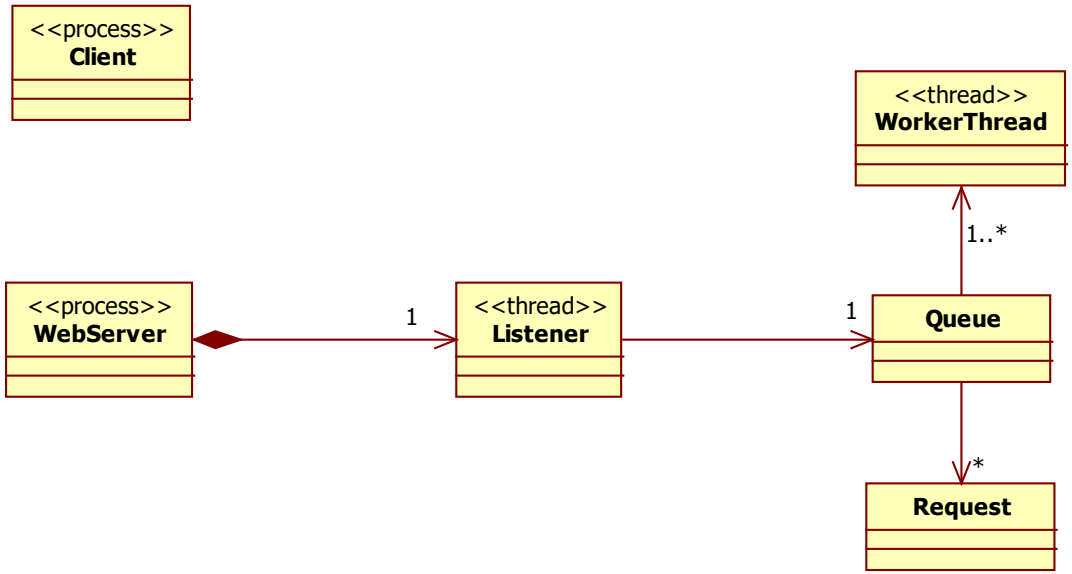
Prozesssicht Sicht der Architektur

Für das „statische Modell“ der Prozesssicht können Klassendiagramme und Objektdiagramme verwendet werden:

Diese dargestellten Klassen werden um folgende Stereotypen ergänzt:

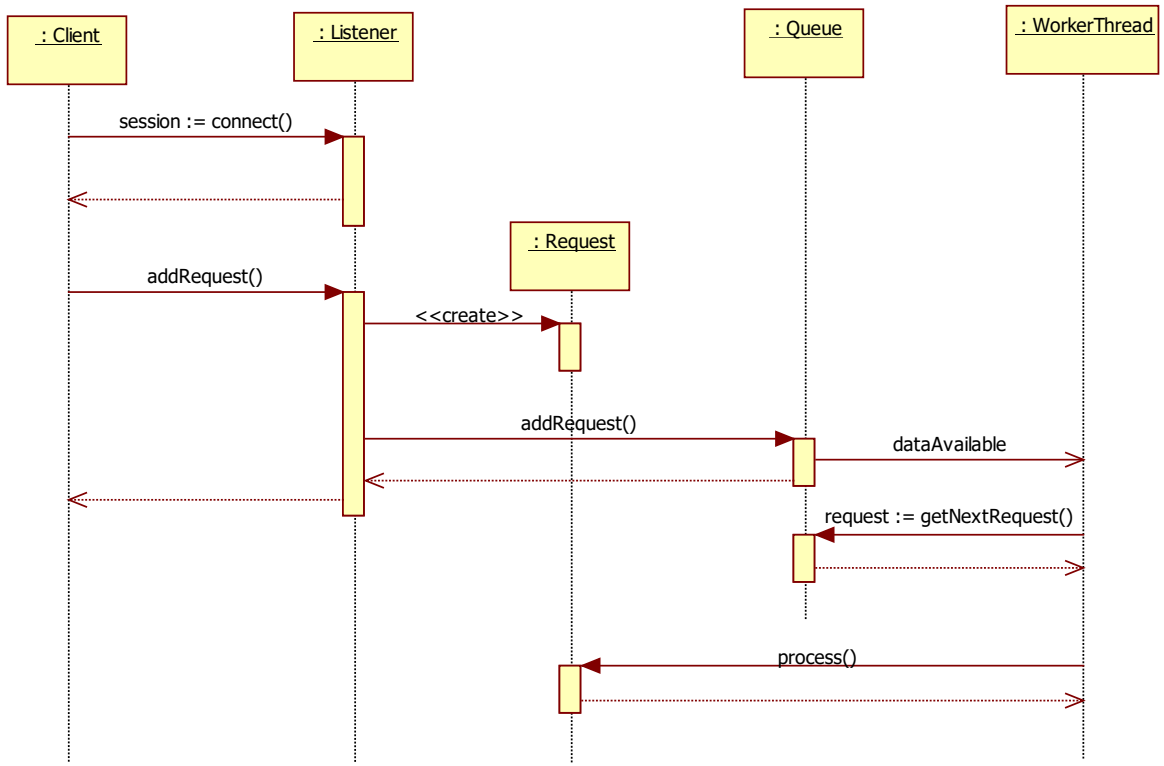
<<thread>>: „lightwight flow of control“

<<process>> „heavywight flow of control“



Prozesssicht Sicht der Architektur

Für das „dynamische Modell“ der Prozesssicht können Sequence und Aktivitätsdiagramme verwendet:



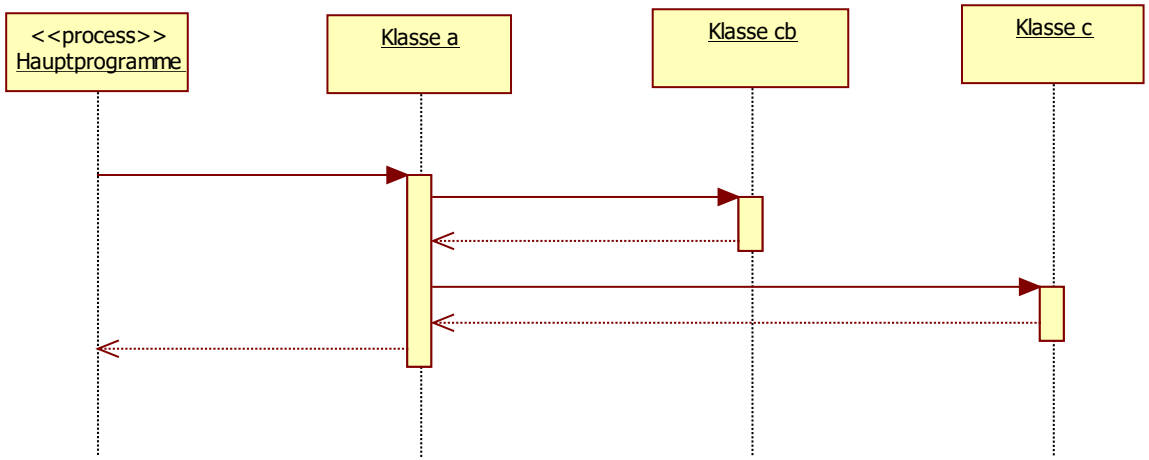
„Call-Return“

Prinzip:

- Es existiert ein Hauptprogramm welches Funktionen und Methoden der verschiedenen Subsystem aufruft
- Subsysteme dürfen selbst wiederum weitere Funktionen und Methoden aufrufen
- Diese Art der Strukturierung wird auch als „Top-Down-Architektur“ bezeichnet

„Call-Return“

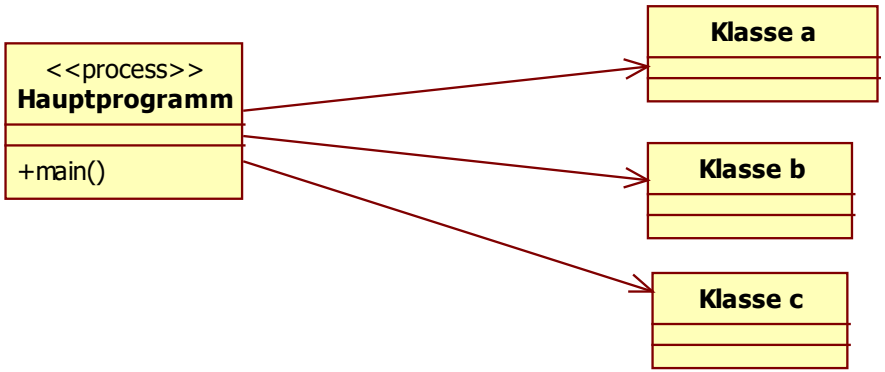
Dynamische Sichtweise



Das Hauptprogramm wird als Prozessgestartet und ruft verschiede Methoden auf

„Call-Return“

Statische Sichtweise



Anwendung & Vorteile:

- Einfache Programme mit klar definierter Struktur
- Keine Parallelität in der Verarbeitung
- Arbeitsschritte werden sequentiell ausgeführt

„Selective Boradcast“

Prinzip:

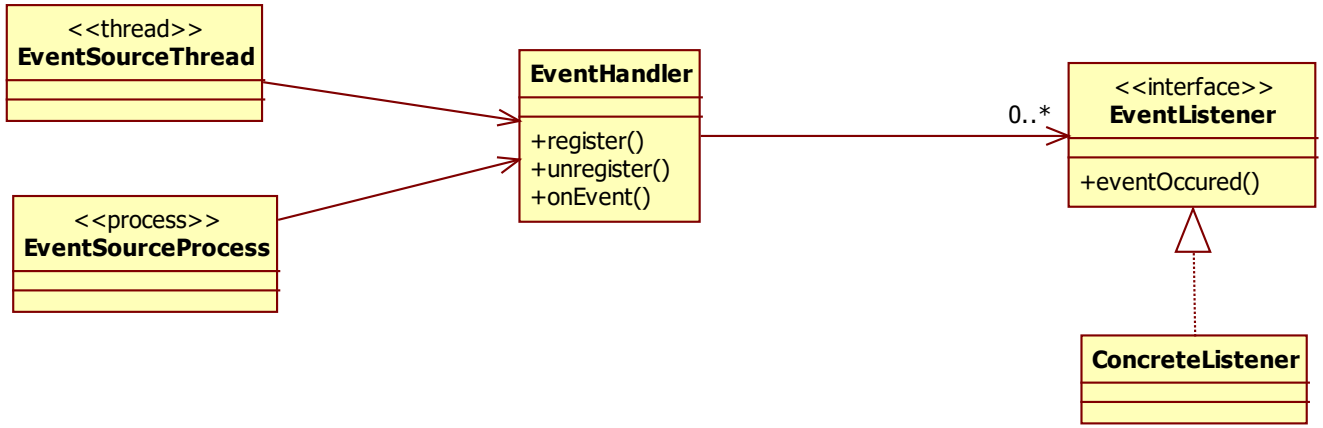
- Verwende ein Ereignisgesteuertes Modell anstelle von direkten Methodenaufrufen
- Komponenten können sich für den Empfang von Ereignissen registrieren.
- Tritt das Ereignis auf, können alle registrierten Komponenten von diesem Ereignis benachrichtig werden
- Auch bekannt als „Publish/Subscribe“ (im kleinen)

Anwendung:

- Verteilte System können auf Basis dieses Muster lose gekoppelte werden
- Erweiterbare, graphische Oberflächen (z.B. Eclipse)

„Selective Boradcast“

Statische Sichtweise

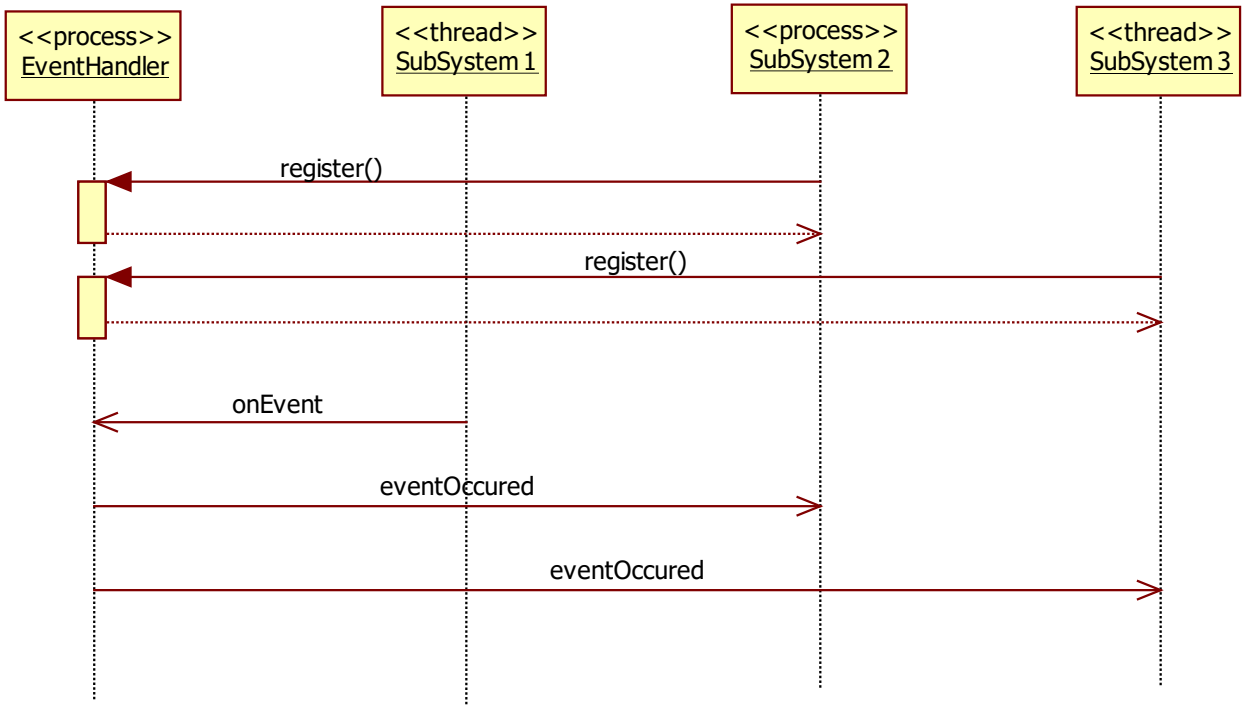


Idee:

- Wird ein Ereignis gemeldet (onEvent) wird dieses an die registrierten Listener weitergegeben
- Ein EventListener muss das Interesse an Events über die Methode „register“ bekannt geben

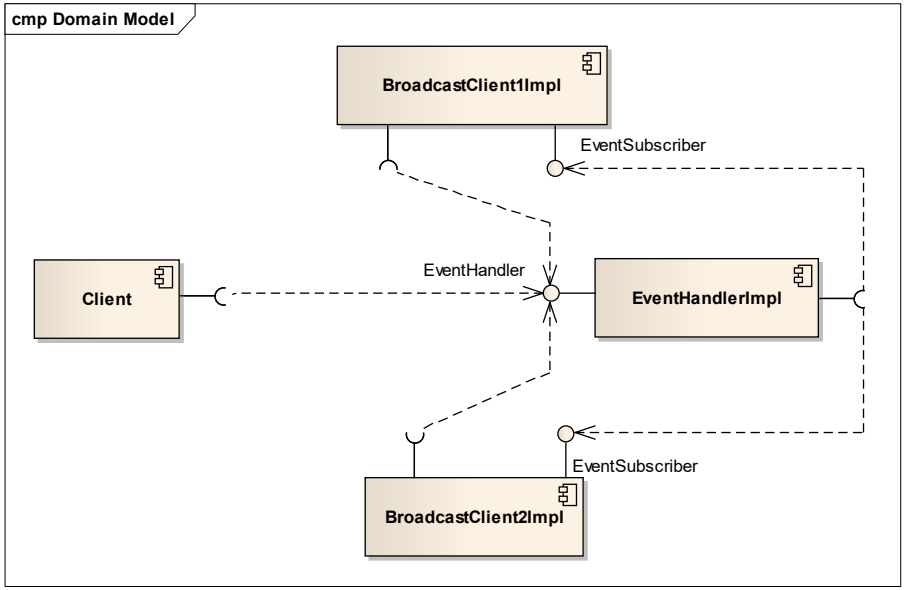
„Selective Boradcast“

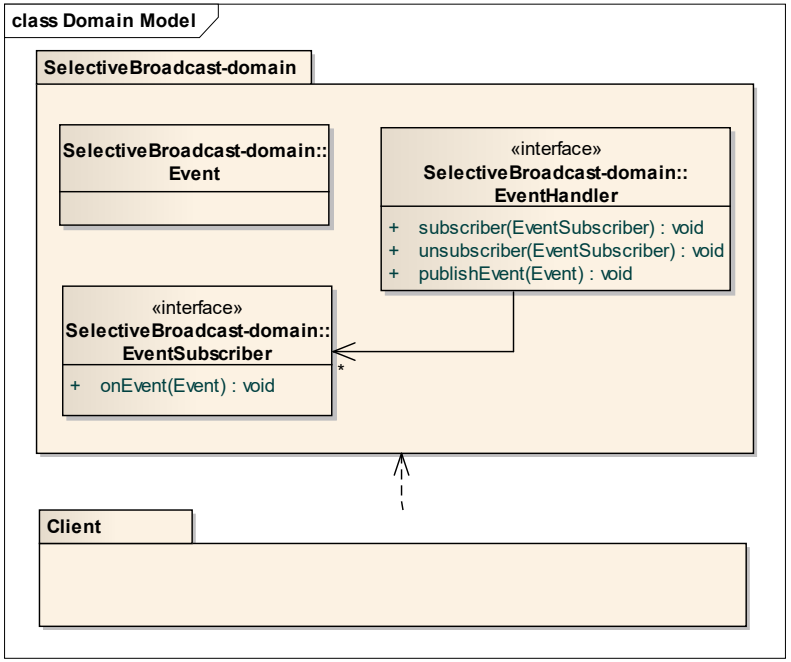
Dynamische Sichtweise



„Selective Boradcast“

Umsetzung als Komponentenarchitektur





„Selective Boradcast“

Vorteile:

- Systeme welche auf diesem Muster basieren sind einfach zu erweitern
- Geringer Kopplunge zwischen dem Kernsystem und den angebunden Komponenten
- Bestehende Komponenten können ausgetauscht werden

Probleme:

- Nach dem melden eines Events gibt die EventSource die Kontrolle über die Verarbeitung des Ereignisses ab
- Die Verarbeitungsreihenfolge über mehrere Listener ist nicht definiert
- Die Eventquelle weis nicht wann alle Events abgearbeitet sind

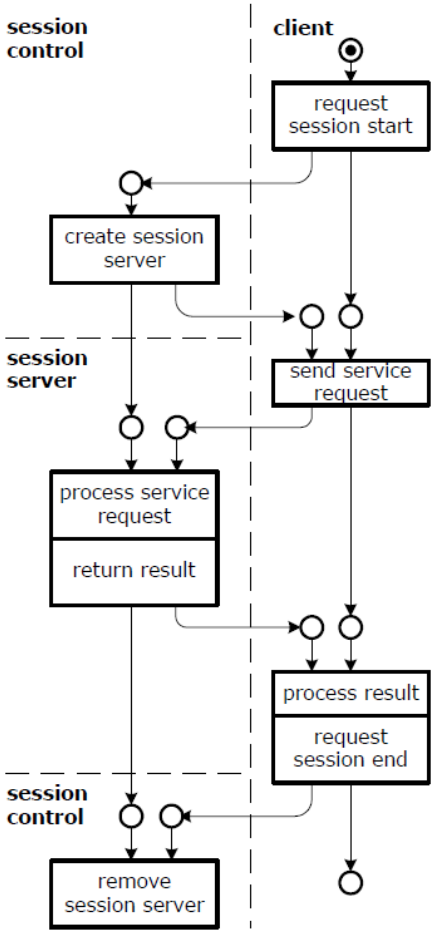
Konzeptionelle Serverpatterns

Motivation:

- Moderne Softwaresysteme sind unterteilt in einen Client- und einen Serverteil
- Die Serverkomponenten sind in der Regel Server welche auf Multitasking aufsetzen um eingehende Aufträge zu bearbeiten
- Nachfolgend wird ein Liste von Patterns für auftragsbasierte Server vorgestellt.

- Achtung: Die vorgestellten Patterns können auch auf andere Aufgabenstellungen übertragen werden. Sie können dann angewendet werden, wenn ein definierter Umgang mit Ressourcen notwendig ist

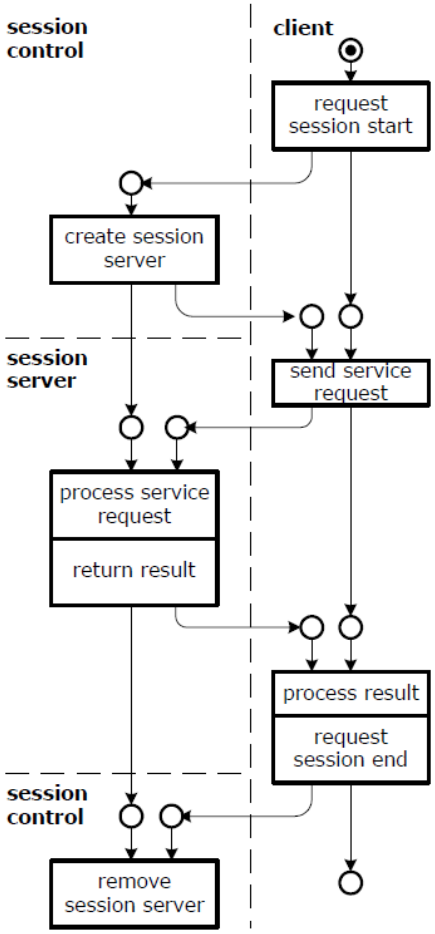
Konzeptionelle Serverpatterns



Single-Request-Session:

- Der Client baut eine Sitzung mit dem Server auf
- Der Client sendet den Auftrag an den Server
- Der Server verarbeitet den Auftrag
- Der Server sendet die Antwort zurück an den Client
- Die Verbindung zwischen Client uns Server wird wieder abgebaut

Konzeptionelle Serverpatterns



Folgende Zeiten sind relevant für den Client:

- Connect Time: Die Dauer zwischen Senden einer (TCP-) Verbindungsanfrage und Aufbau der Verbindung
- Response Time: Zeit für die Antwort auf den Auftrag nach Verbindungsaufbau

Konzeptionelle Serverpatterns – Sequentieller Server

Pattern: Sequentieller Server

Kontext :

Ein Server soll eingehende Client-Anfragen verarbeiten.

Konzeptionelle Serverpatterns – Sequentieller Server

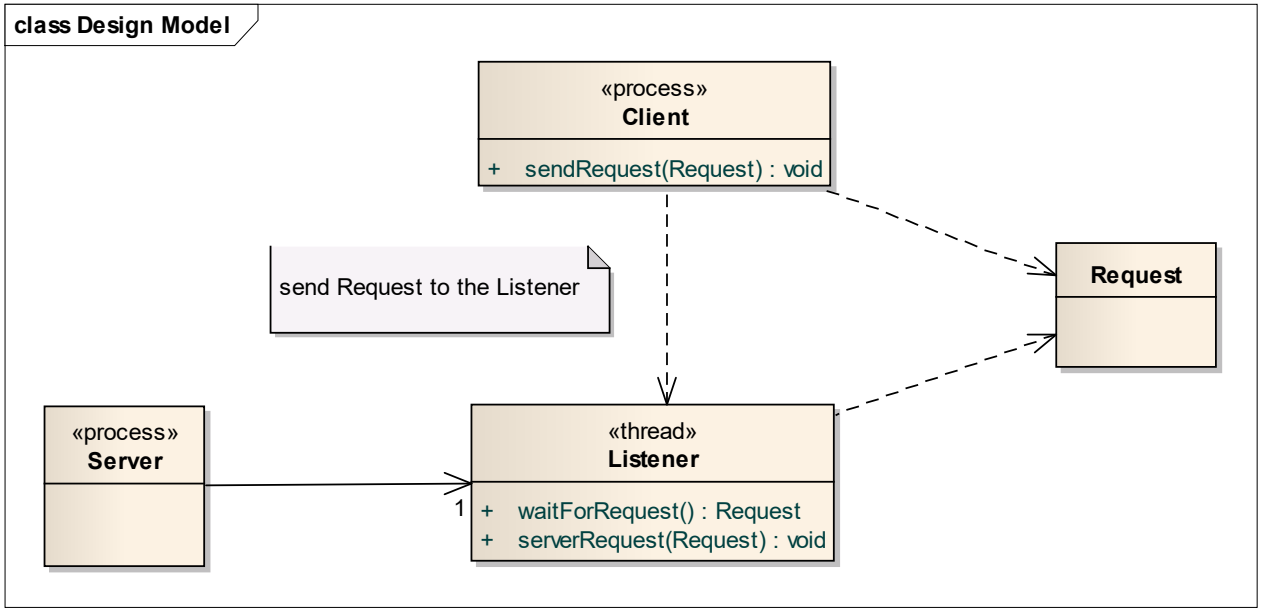
Lösung: Ein Server-Thread ist für die Abarbeitung der Anfragen zuständig

Server-Thread:

- Eine Task überwacht den oder die Server Ports
- Bei einem Connection-Request wird die Verbindung zum Client aufgebaut.
- Danach wird die Anfrage verarbeitet und die Verbindung geschlossen
- Danach können neue Anfragen bearbeitet werden

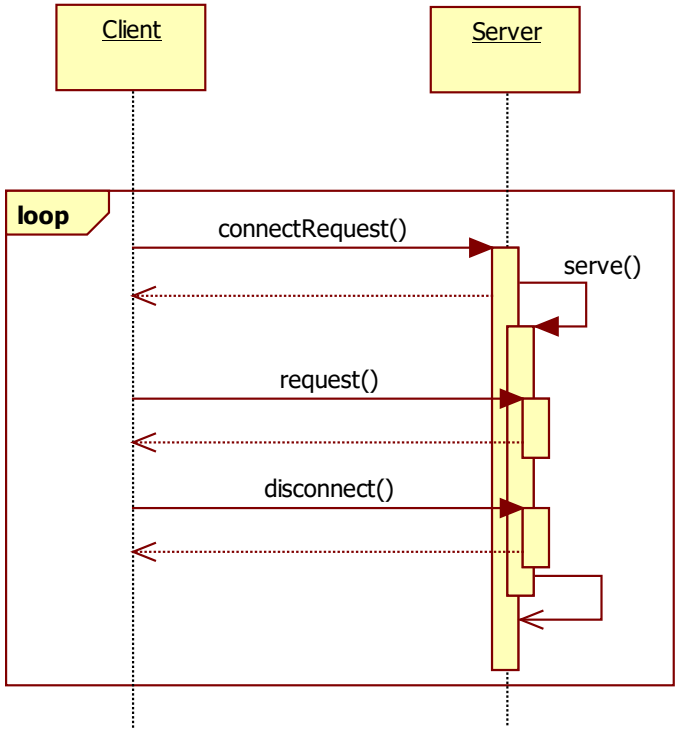
Konzeptionelle Serverpatterns - Sequentieller Server

Statische Sichtweise



Konzeptionelle Serverpatterns - Sequentieller Server

Dynamische Sichtweise



Konzeptionelle Serverpatterns – Sequentieller Server

Eigenschaften:

- Die Implementierung dieses Musters ist sehr einfach
- Es erfolgt eine Zwangsserialisierung der eingehende Anfragen
- Eine Synchronisation ist nicht notwendig
- Ressourcennutzung ist exakt definiert

Problem:

- Die vorhandenen Maschinenresourcen werden nicht optimal ausgenutzt

Konzeptionelle Serverpatterns – Sequentieller Server

Implementierungsbeispiele:

Siehe:

- Paket fileimport.sequential
- Paket tcpserver.sequential

Konzeptionelle Serverpatterns – Listener Worker

Pattern: Listener/Worker

Kontext :

Ein Server soll mit Hilfe von Multitasking Dienste für eine offene Anzahl an Clients nebenläufig anbieten. Es wird ein verbindungsorientiertes Netzwerkprotokoll (z.B. TCP/IP) verwendet.

Problem

Wie benutzt man Tasks (Prozesse oder Threads) zur nebenläufigen Bearbeitung von Connection– und Service–Requests?

Konzeptionelle Serverpatterns - Listener Worker

Lösung: Unterteile die Tasks im Server in Listener und Worker Tasks:

Listener:

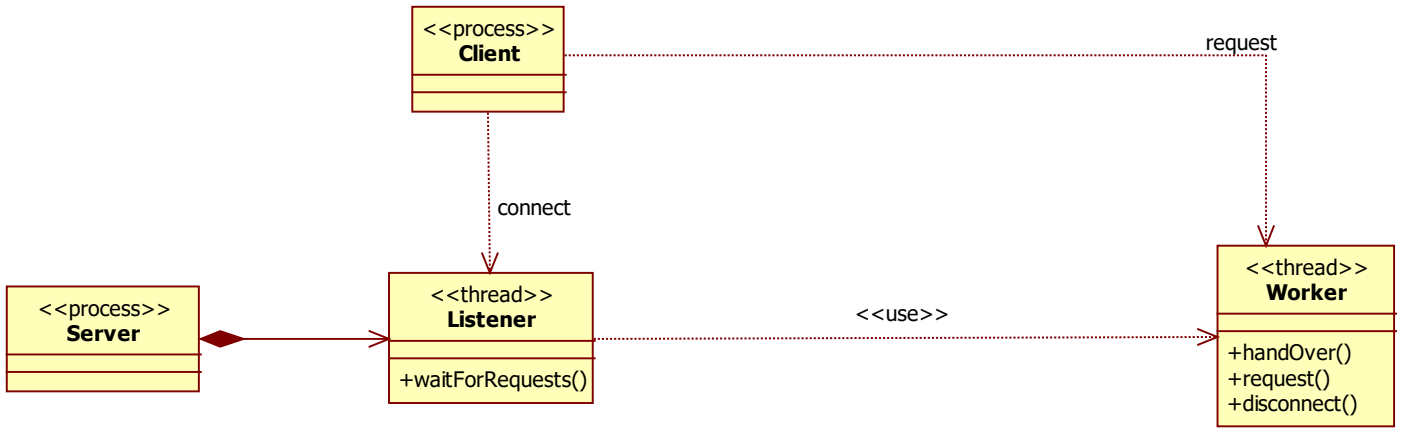
Eine Task überwacht den oder die Server Ports, baut bei einem Connection Request die Verbindung zum Client auf und übergibt diese Verbindung an einen Worker.

Worker:

Der Worker verarbeit eingehende Aufträge des Clients und liefert das gewünschte Ergebnis zurück. Idealerweise gibt es pro Client ein Worker Task, welcher ausschließlich für den Client aktiv ist.

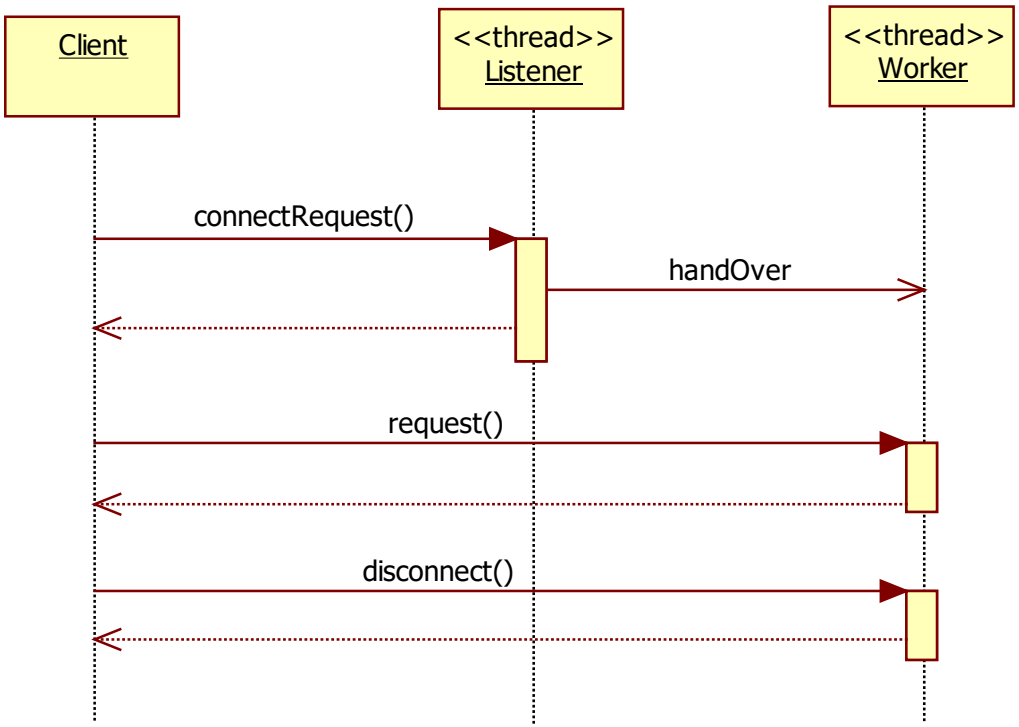
Konzeptionelle Serverpatterns - Listener Worker

Statische Sichtweise



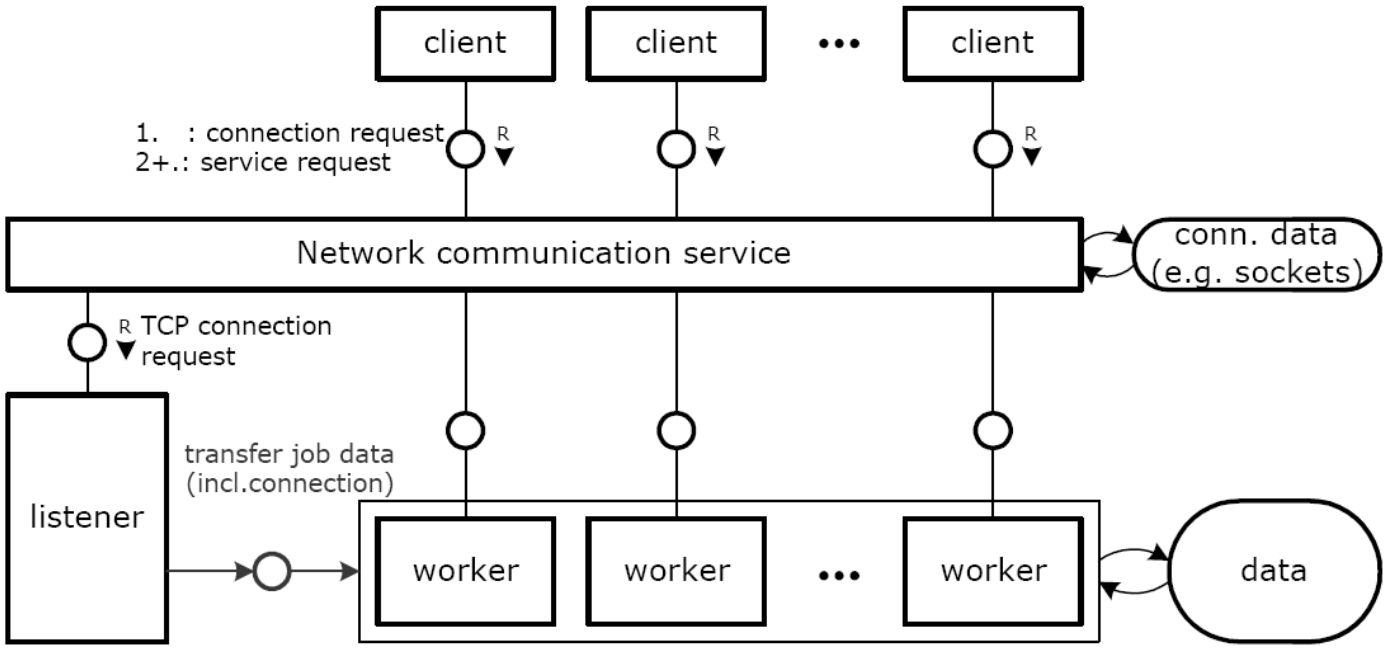
Konzeptionelle Serverpatterns - Listener Worker

Dynamische Sichtweise



Konzeptionelle Serverpatterns - Listener Worker

Statische Sichtweise



Konzeptionelle Serverpatterns - Listener Worker

Eigenschaften:

- Der Verbindungsaufbau erfolgt schnell, da der Listener nur die Aufgabe hat, auf Connection Requests zu reagieren.
- Synchronisation beim Zugriff auf Resourcen notwendig

Offene Fragen/Punkte

- Wann werden die Worker Tasks erzeugt?
- Wie übergibt der Listener die Verbindung zum Client an einen Worker?

Konzeptionelle Serverpatterns – Forking Server

Pattern: Forking Server

Ausgangssituation

Man wendet für einen auftragsbearbeitenden Server das LISTENER / WORKER Pattern an.

Problem

Wie handhabt man möglichst einfach die Erzeugung der Worker Tasks und die Übergabe der Client–Verbindung?

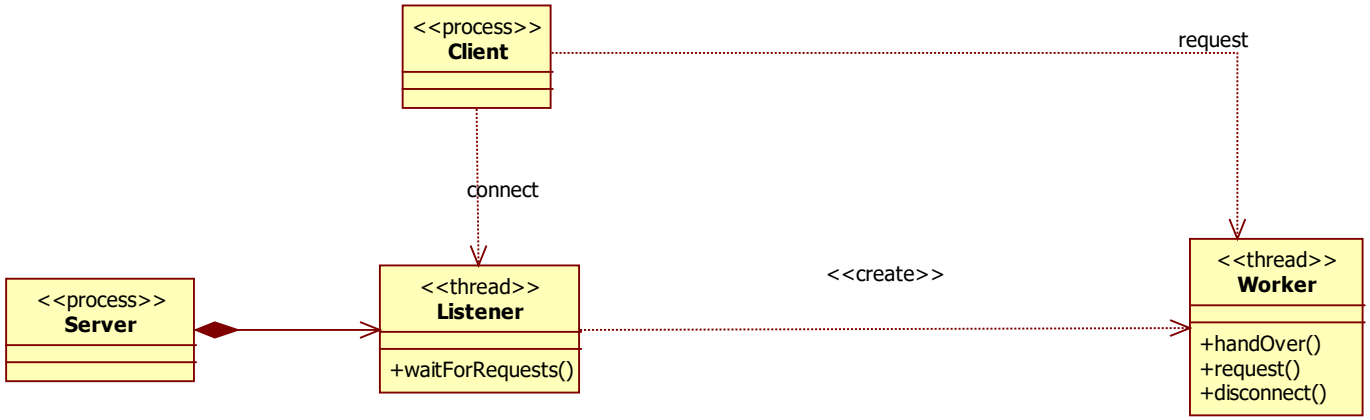
Konzeptionelle Serverpatterns – Forking Server

Lösung: Für jeden Connection-Request wird ein eigenen Worker gestartet.

- Starte eine Task als Master Server, die die Rolle des Listeners übernimmt
- Für jeden eingehenden Connection-Request wird ein neuer Worker-Task erzeugt
    - Unix: fork
    - Windows: CreateProcess
    - Alternativ: Thread
- Der Worker-Task bearbeitet die Service-Request und beendet sich nach getaner Arbeit

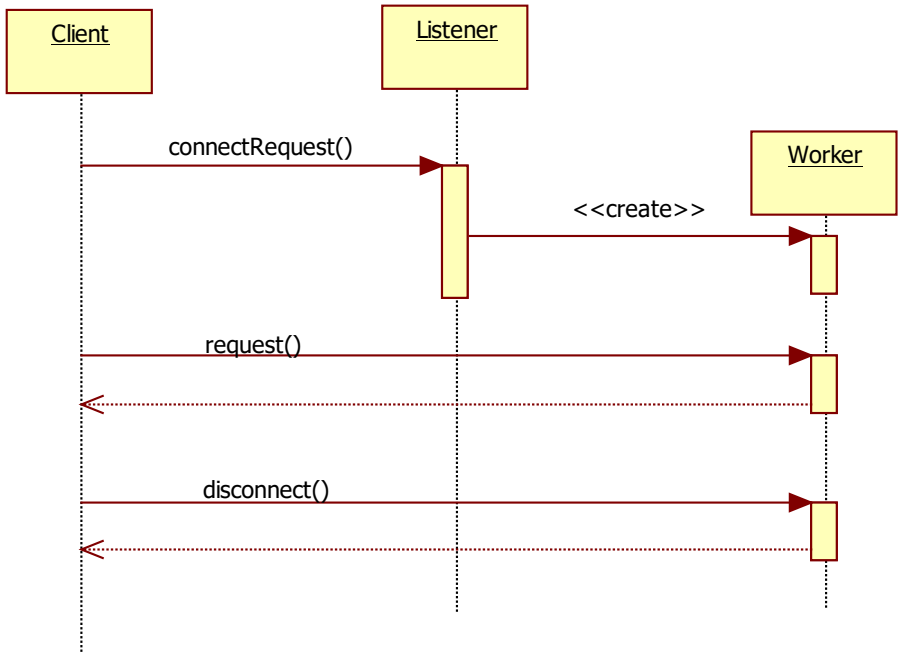
Konzeptionelle Serverpatterns – Forking Server

Statische Sichtweise



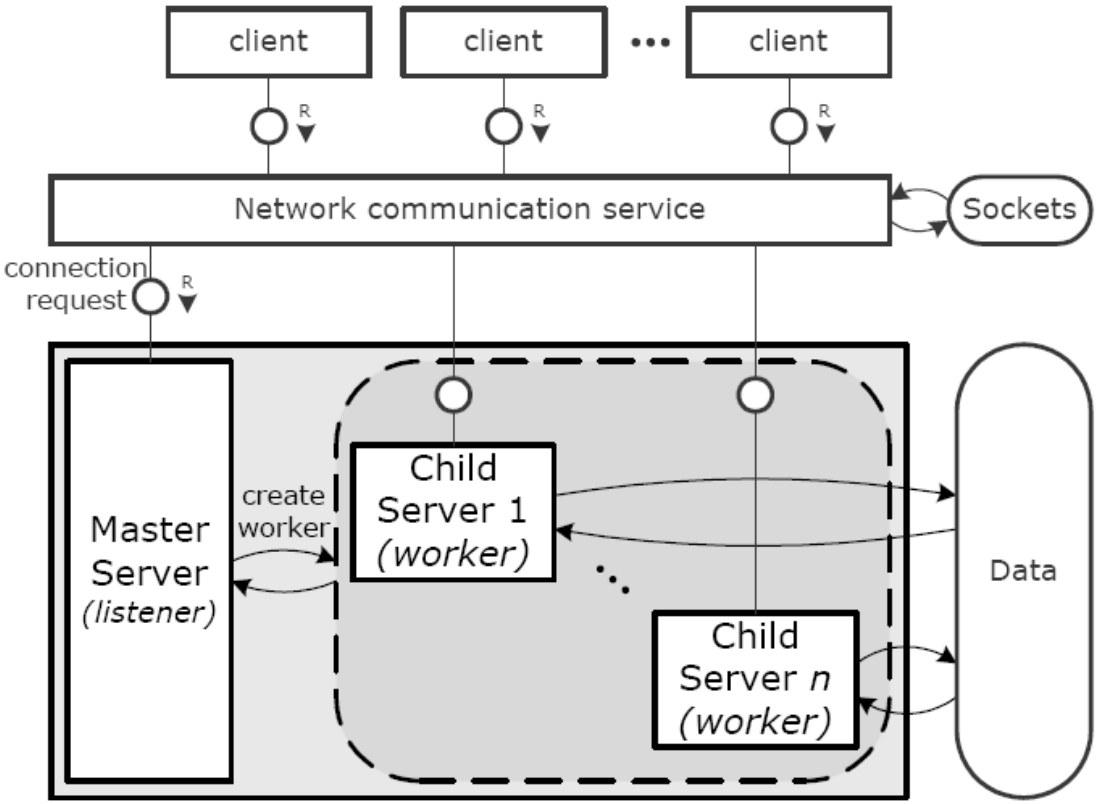
Konzeptionelle Serverpatterns – Forking Server

Dynamische Sichtweise



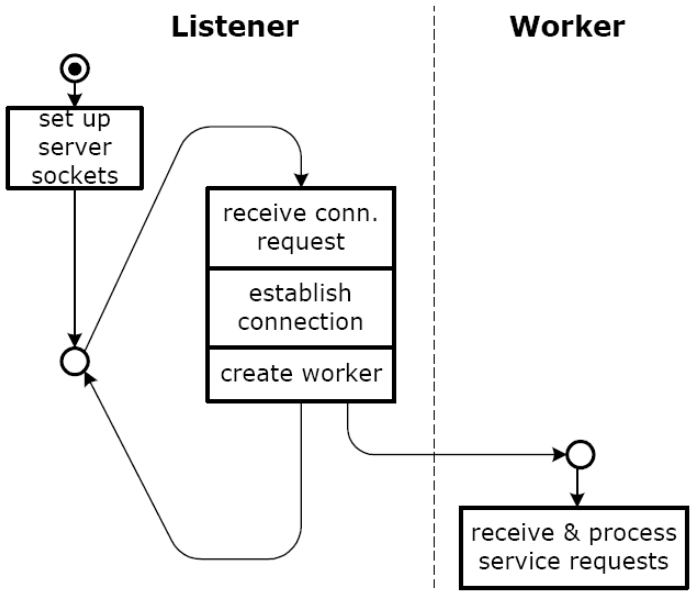
Konzeptionelle Serverpatterns – Forking Server

Statische Sichtweise



Konzeptionelle Serverpatterns – Forking Server

Dynamische Sichtweise



Konzeptionelle Serverpatterns – Forking Server

Eigenschaften:

- Die Übergabe der Client–Verbindung zwischen Listener und Worker funktioniert auch bei Verwendung von Betriebssystem–Prozessen.
- Die Anzahl der Tasks richtet sich genau nach dem aktuellen Bedarf
- Ressourcennutzung ist abhängig von der auftreten Last

Offene Fragen/Punkte

- Die Antwortzeit des Servers hängt von der Zeit ab, die das Erzeugen eines Workers erfordert
- Das ständige Erzeugen und Freigeben von Systemressourcen kostest Zeit und führt zu einer höheren Systembelastung
- Problematisch bei hoher punktueller Belastung

Konzeptionelle Serverpatterns – Forking Server

Implementierungsbeispiele:

Siehe:

- Paket fileimport.forking
- Paket tcpserver.forking

Konzeptionelle Serverpatterns – Worker Pool

Pattern: Worker Pool

Ausgangssituation

Man wendet für einen auftragsbearbeitenden Server das LISTENER / WORKER Pattern an.

Problem

Wie erreicht man eine möglichst kurze Antwortzeit?

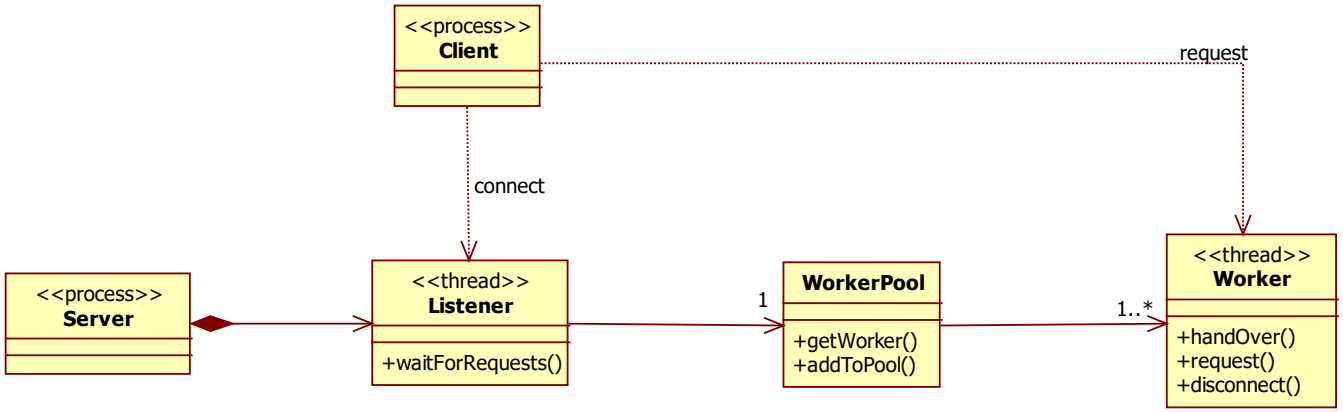
Konzeptionelle Serverpatterns – Worker Pool

Lösung:

- Erzeuge bei Start des Servers eine bestimmte Anzahl von Worker Tasks
- Solange keine Arbeit vorliegt sind diese Tasks inaktiv
- Eingehende Connect-Anforderungen werden auf einen inaktiven Task zugeteilt

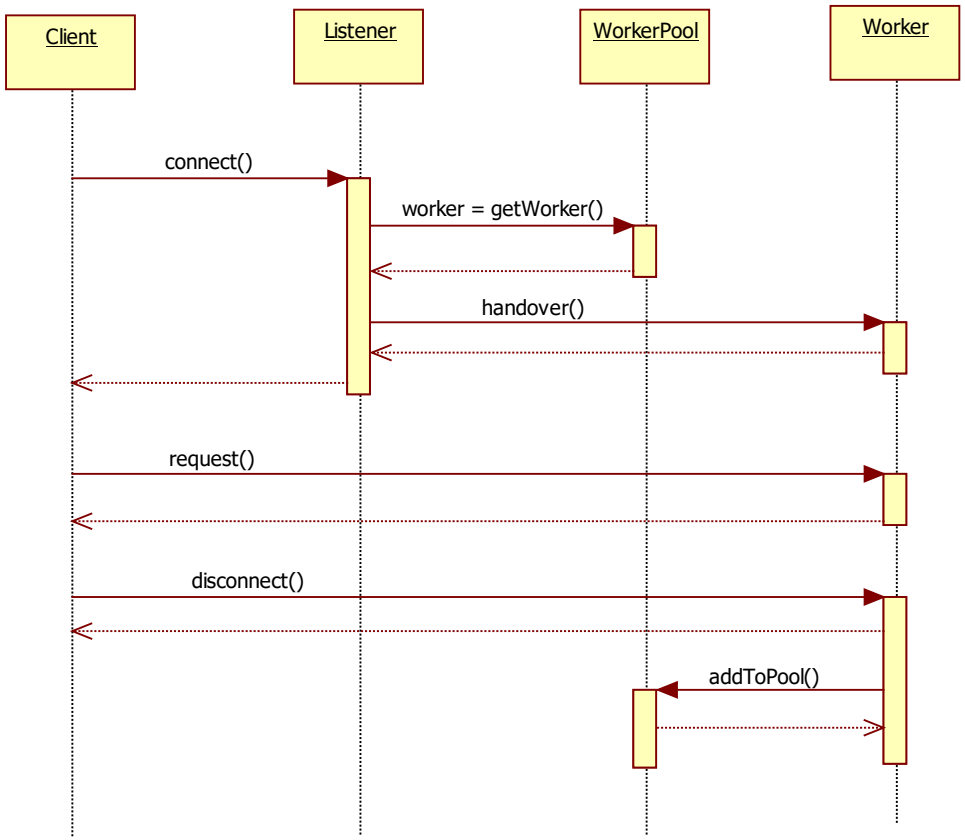
Konzeptionelle Serverpatterns – Worker Pool

Statische Sichtweise



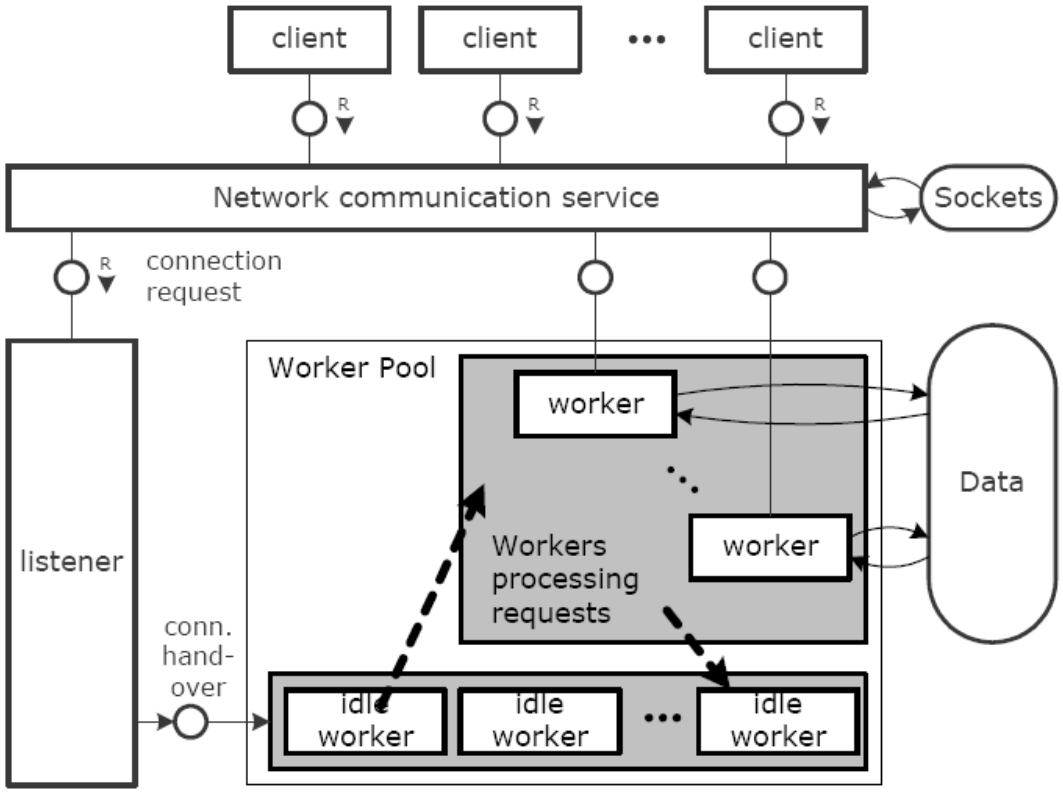
Konzeptionelle Serverpatterns – Worker Pool

Dynamische Sichtweise



Konzeptionelle Serverpatterns – Worker Pool

Statische Sichtweise



Konzeptionelle Serverpatterns – Worker Pool

Eigenschaften:

- Es geht bei der Annahme von Requests keine Zeit mehr für die Erzeugung von Worker Tasks verloren
- Die Anzahl der Worker Tasks kann begrenzt und damit für die Maschinen–Ressourcen optimiert werden
- Eingehende Anfragen müssen warten bis ein zugehöriger Worker zur Verfügung steht
- Synchronisation beim Zugriff auf Resourcen notwendig

Konzeptionelle Serverpatterns – Worker Pool

Implementierungsbeispiele:

Siehe:

- Paket fileimport.workerpool
- Paket tcpserver.workerpool

Konzeptionelle Serverpatterns – Job Queue

Pattern: Job Queue

Ausgangssituation

Man wendet das LISTENER / WORKER und das WORKER POOL Pattern an.

Problem

Wie übergibt der Listener eine Client–Verbindung an einen Worker

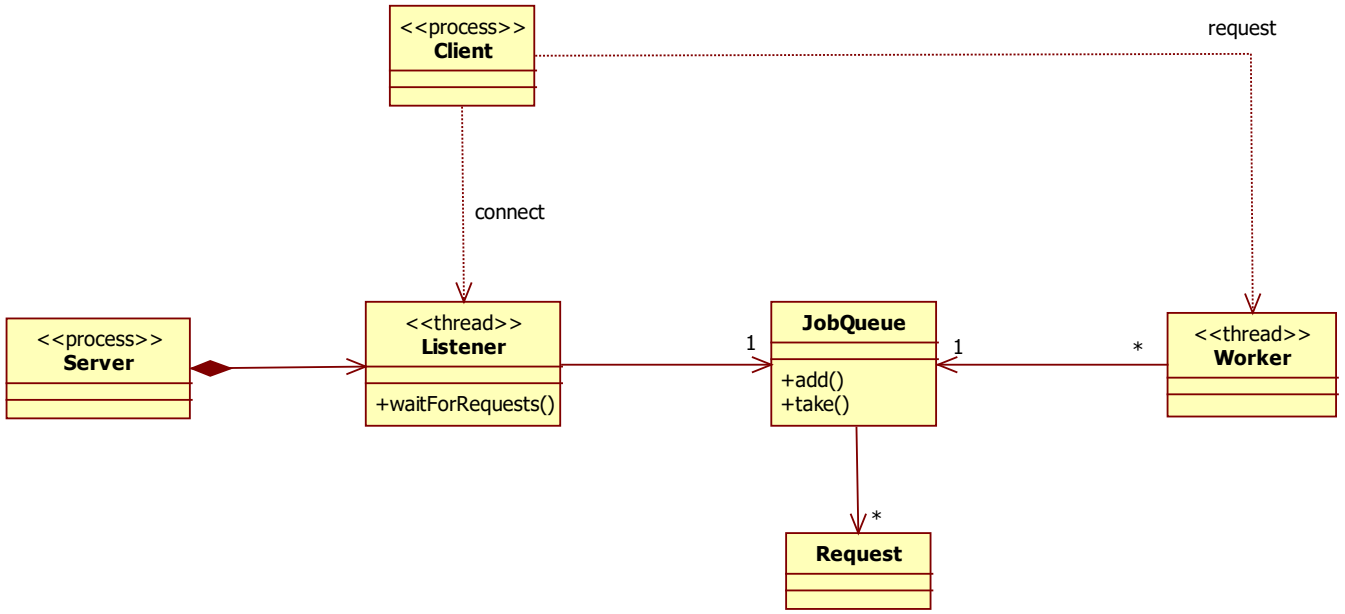
Konzeptionelle Serverpatterns – Job Queue

Lösung:

- Eine Job Queue dient zur Übergabe der Client–Verbindung zwischen Listener und Worker.
- Alle inaktiven Worker warten darauf, dass ein Job in die Queue kommt.
- Nach Annahme eines Connection Requests schiebt der Listener die Client–Verbindung als Job in die Queue und weckt damit den ersten inaktiven Worker
- Der Worker nimmt den Job aus der Queue und kommuniziert mit dem Client
- Falls keine inaktiven Worker Tasks warten, füllt sich die Queue mit jedem Connection Request.

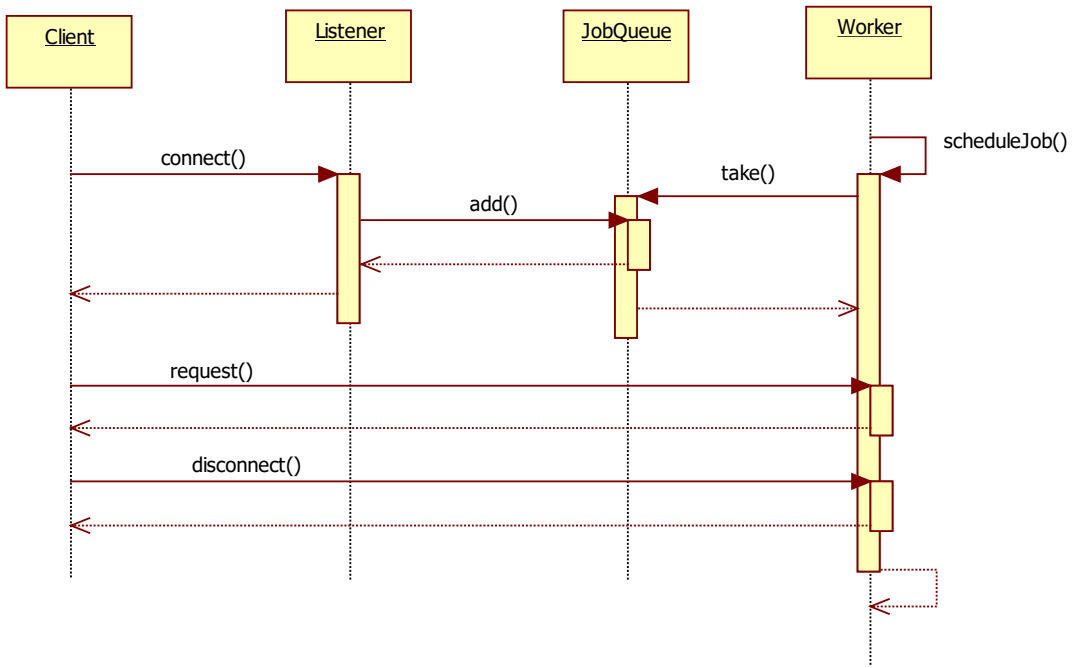
Konzeptionelle Serverpatterns – Job Queue

Statische Sichtweise



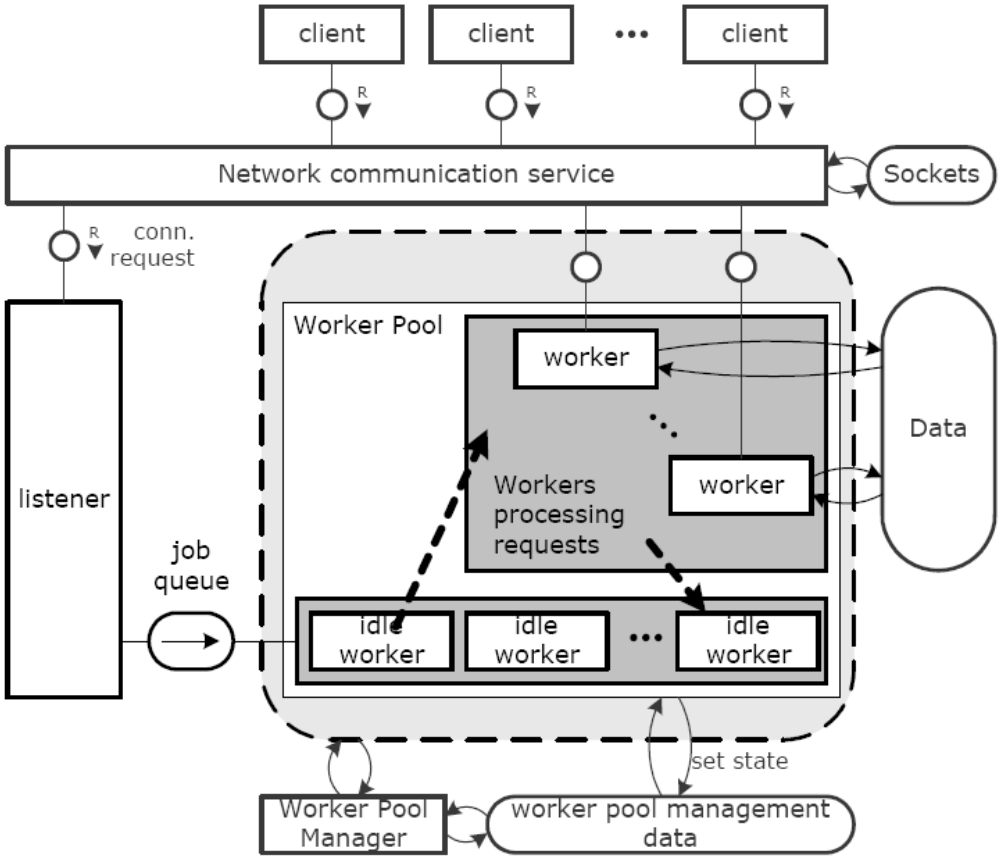
Konzeptionelle Serverpatterns – Job Queue

Dynamische Sichtweise



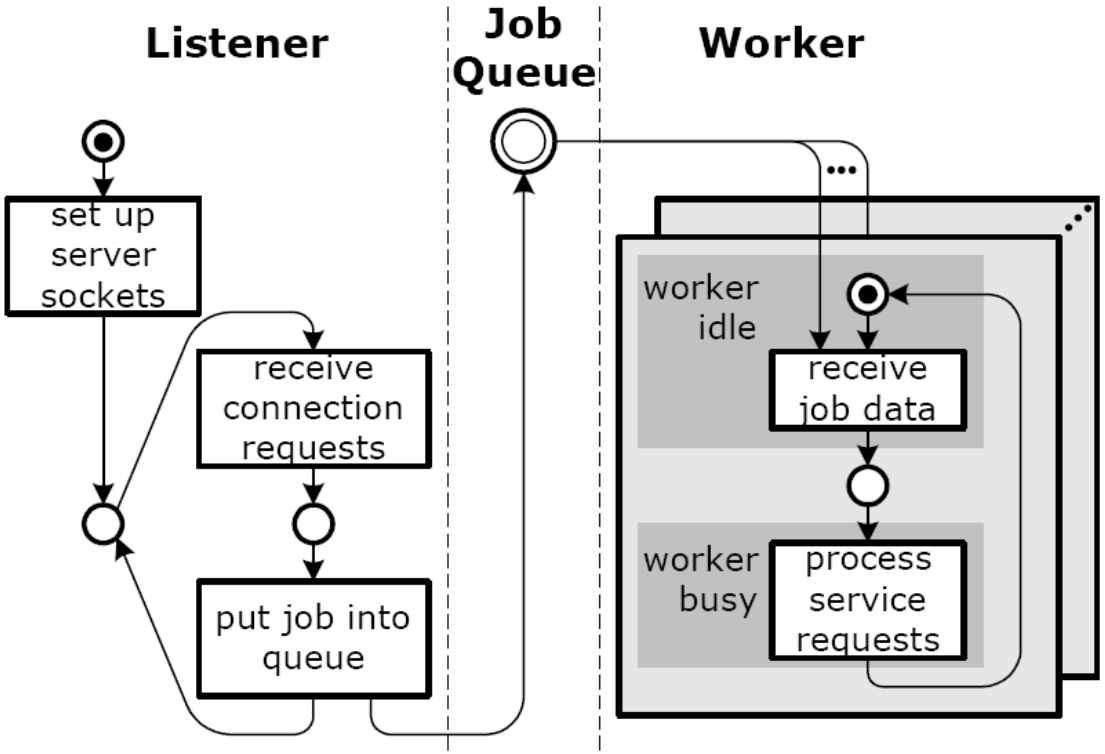
Konzeptionelle Serverpatterns – Job Queue

Statische Sichtweise



Konzeptionelle Serverpatterns – Job Queue

Dynamische Sichtweise



Konzeptionelle Serverpatterns – Job Queue

Eigenschaften:

- Der Listener nimmt mit der kürzest möglichen Antwortzeit Connection Requests an
- Durch die Job Queue sind auch Worker Pools mit statischer Anzahl an Workern effizient
- ein Client muss eventuell etwas warten, bis sein Service Request angenommen und bearbeitet wird
- Falls möglich kann die Queue auch persistent abgelegt werden

Offene Fragen/Punkte

- Wie kann die Auslastung der Queue begrenzt werden?

Konzeptionelle Serverpatterns – Worker Pool

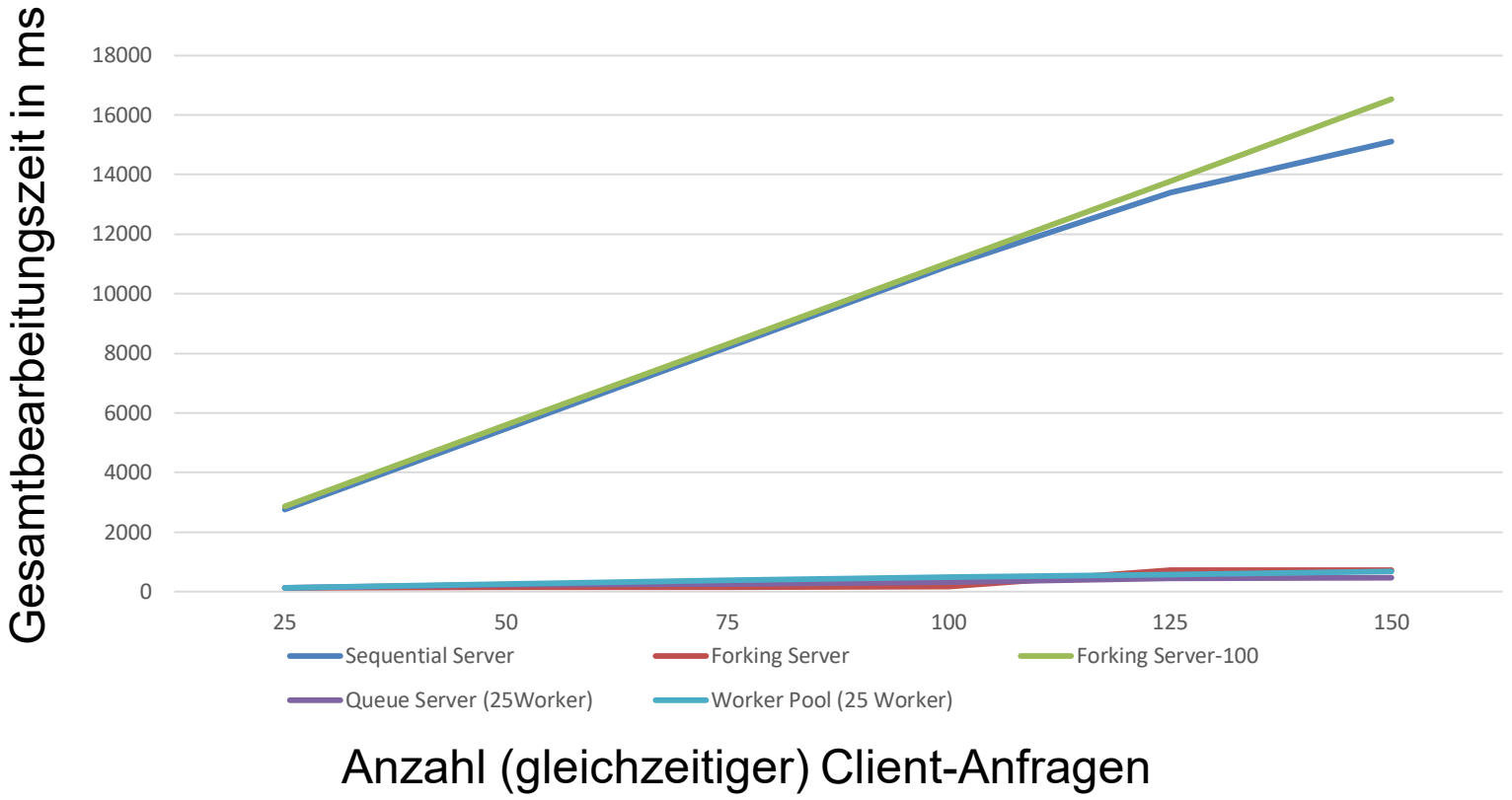
Implementierungsbeispiele:

Siehe:

- Paket fileimport.queue
- Paket tcpserver.queue

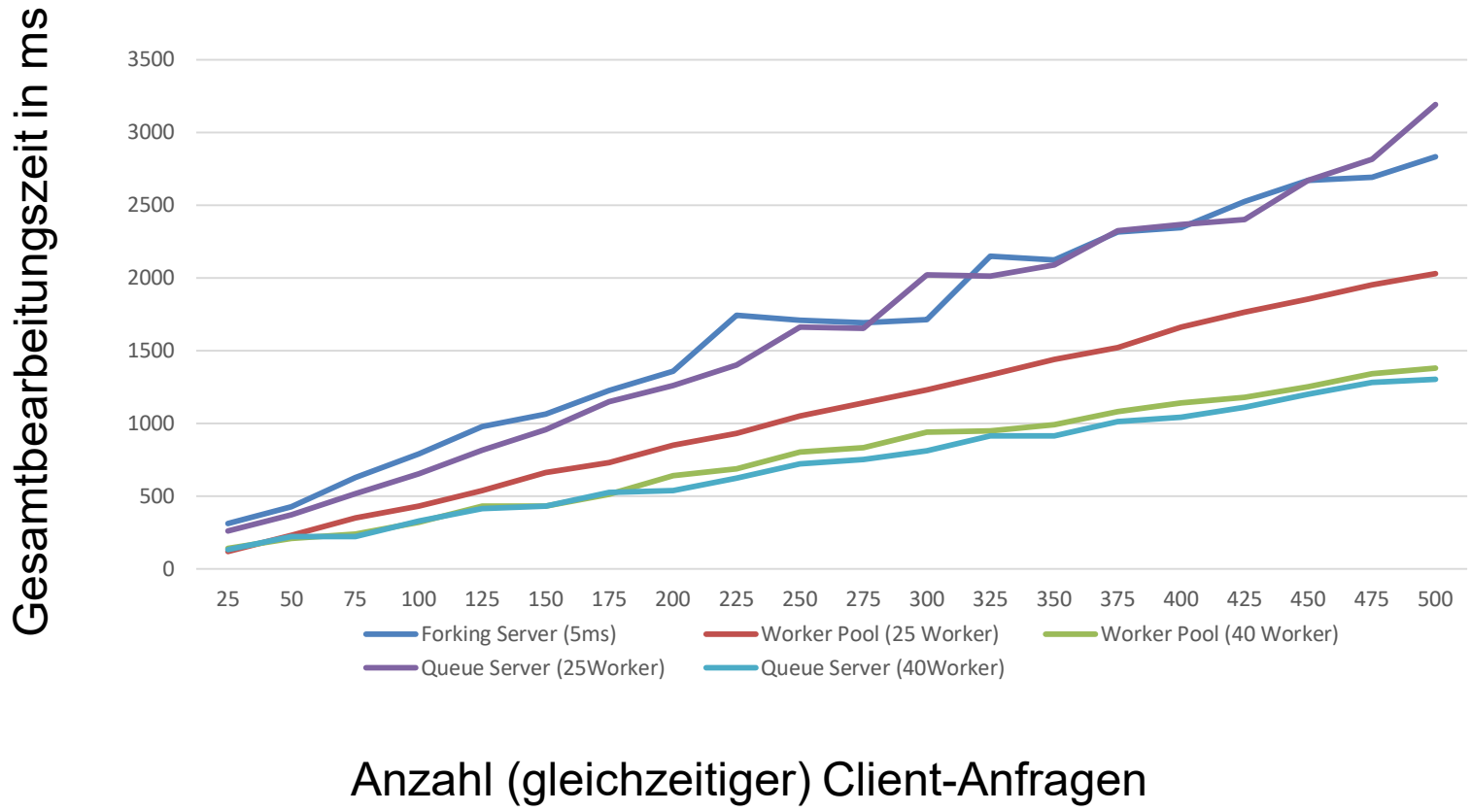
Konzeptionelle Serverpatterns – Vergleich

Performancevergleich der vorgestellten Patterns



Konzeptionelle Serverpatterns – Vergleich

Performancevergleich der vorgestellten Patterns



Prozess-/Ablaufsicht

Fazit

- Die vorgestellten Pattern erlauben die bewusste Steuerung der nicht-funktionalen Anforderungen
- Für jedes (Server-)basierte System ist die Verarbeitung von Aufträgen von zentraler Bedeutung.
- Eine definierte Resourcen-Auslastung ist Garant für eine stabiles und zuverlässiges System welches in definierten Rahmenparameter arbeitet
- Verschiedene Pattern können in einer Applikation kombiniert werden um das optimale Ablaufverhalten zu erzielen
- Die Element der Ablaufsicht können als Komponenten an die restliche Applikation exportiert werden

Prozess-/Ablaufsicht

Ausblick

- Die Zeitgesteuerte/Periodische Bearbeitung von Jobs ist ebenfalls der Ablaufsicht zuzuordnen.
- Die Zeitsteuerung kann in Kombination mit z.B. einer Work-Queue betrieben werden