Hochschule Karlsruhe
University of
Applied Sciences

Infrastruktur-minimales, paralleles Backend für Offline-Chartanalysen

Bachelorarbeit

Student:	Lennart Ruck	78861
Universität:	Hochschule Karlsruhe – Technik und Wirtschaft	
Studiengang:	INFB	
Semester:	Wintersemester 2024/2025	
Referent:	Prof. Dr. Thomas Fuchß	
Betreuer:	Dr. Jens Lemcke	
Bearbeitet am:	14. April 2025	

Danksagung

Ich möchte mich bei allen bedanken, die mich bei der Erstellung dieser Bachelorarbeit unterstützt haben. Ein besonderer Dank geht an meinen Betreuer Herrn Dr. Lemcke für seine fachliche Unterstützung und das konstruktive Feedback im Verlauf der Arbeit. Ebenso danke ich meinem Referenten Herrn Prof. Dr. Fuchß an der Hochschule Karlsruhe für die Unterstützung im Rahmen der gemeinsamen Abstimmungen. Außerdem möchte ich mich bei der PowMio GmbH für das Vertrauen und die Unterstützung bedanken, die diese Arbeit möglich gemacht haben.

Abstract

Technische Analysten im Aktienhandel stehen vor einer großen Herausforderung, sie benötigen die Möglichkeit viele Chartanalysen schnell und effizient auszuführen. Das Ausführen dieser Aufgaben ist langsam und aufwändig. Das Ziel der Arbeit ist es ein System zu entwickeln, welches einfach nutzbar ist und das Ausführen der Aufgabenmenge beschleunigt. Bewusst verzichten wir auf das Einsetzen einer großen Enterprise Lösung und fokussieren uns stattdessen auf eine leichtgewichtige Lösung. Die Lösung soll wenig komplex, ressourcenschonend und gut skalierbar sein um den Kosten-Nutzen Faktor des Systems zu maximieren.

Im Fokus der Arbeit steht eine konzeptionelle Lösung des Problems, dem Verarbeiten großer Aufgabenmengen auf verteilten Systemen. Durch das Abstrahieren der Aufgaben selbst lässt sich unsere Lösung auch auf andere Geschäftsdomänen anwenden. Zur Lösungsfindung beschäftigt sich die Arbeit mit technischen und methodischen Ansätzen die für solche Fragestellungen geeignet sind. Insbesondere im Kontext der Lastenverteilung auf verteilten Systemen und cloud-basierten Architekturen.

Basierend auf dieser Forschung werden drei mögliche Lösungsalternativen entworfen, bewertet und miteinander verglichen. Eine dieser drei Alternativen wird umgesetzt und erläutert. Die Implementierung zeigt dass wir ein effizientes und performantes System zum Verarbeiten von Aufgabemengen realisieren konnten. Die gezeigte Lösung ist in ihrer Art auch auf Probleme anderer Geschäftsdomänen anwendbar. Das Ergebnis der Arbeit ist eine fundierte Übersicht über relevante Konzepte und Techniken und dient als Einstiegspunkt für die Entwicklung leichtgewichtiger verteilter Verarbeitungssysteme.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem einleitenden Kapitel wird der Misstand den die Bachelorarbeit löst, in der ersten Sektion hergeleitet und in der zweiten Sektion ausformuliert. Im direkten Anschluss wird das Ziel, Abgrenzung und Vorgaben der Lösung behandelt bevor das Kapitel nach einer kurzen Erklärung zum Aufbau der BA endet.

1.1 Herleitung des Problems und Motivation

In dieser Arbeit wollen wir uns mit einem Problem der technischen Analysten (TA), eine Personengruppe, die im Aktienhandel tätig ist, befassen, wessen Lösung dessen Geschäft wirtschaftlich macht und in seiner Art aber genug Abstraktion beinhaltet, um auf ähnliche Probleme angewendet werden zu können.

Im Rahmen unserer Arbeit ist der Aktienhandel das Handeln von Aktien gegen Geld. Das Ziel ist es Profit mit diesem Handel zu erwirtschaften.

Der Aktienhandel funktioniert weitestgehend so, ein Teilnehmer des Handels, z.B. eine Person, kann eine Bestellung zum Kauf einer Aktie aufgeben, besitzt der Teilnehmer die Aktie kann er die Aktie(n) im Besitz stattdessen zum Verkauf anbieten. Die Bestellungen werden bei einem Broker aufgegeben, dieser führt Paare geeigneter Bestellungen gegen eine Gebühr aus, es kommt zum Geschäft. Den Preis dieser Geschäfte kann man zeitlich zu einem Preisverlauf, oder auch Chart genannt, festhalten. Ein Händler spekuliert über den zukünftigen Verlauf des Charts, denn so kann er Einkaufen bevor der Preis stark ansteigt und Verkaufen bevor der Preis wieder stark fällt. Der Begriff stark ist hier bewusst vage gewählt, da das Wählen der Preise der Einkaufs- und Verkaufsbestellungen, sowie der Zeitpunkt des Aufgebens der Bestellungen, ja genau die Kunst des Handelns ist.

Es gibt verschiedene Personen und Personengruppen welche am Aktienhandel beteiligt sind, wir wollen uns die technischen Analysten und dessen Probleme genauer ansehen.

Technische Analysten betreiben Aktienhandel und legen ihren Entscheidungen, im Handeln, Charts über vergangene Preisverläufe und andere Statistiken der Aktienmärkte zu Grunde [1, Seite 21]. Das Wählen des Bestellpreises und des Bestellzeitpunktes modellieren TAs in Form von Handelsstrategien. Eine einzelne Handelsstrategie enthält Regeln für das “Wann geben wir eine Bestellung auf?” und Berechnungsformeln, sog. Indikatoren, für “Wie hoch ist der Preis der Bestellung?”. Diese Regeln und Indikatoren sind auf Zahlenwerten des Charts definiert. Der TA definiert eine gute Handelsstrategie, als eine welche gute Bestellungen ausgibt und damit zu profitablen Geschäften führt. Um beurteilen zu können ob eine Strategie gut ist, betrachtet er dessen Performanz wenn er sie in der Vergangenheit angewandt hätte.

Dies ermöglicht ihm ein Algorithmus, der sog. Backtest. Overfit, ein internes Projekt, bietet eine Implementierung dieses Algorithmus die es erlaubt Handelsstrategien zu konfigurieren und auf vergangenen

Charts von Aktienmärkten auszuführen. Der Backtest simuliert einen Händler der seine Strategie befolgt, d.h. er gibt Bestellungen auf, wenn die zugehörige Regel zutrifft und der Backtest simuliert dann anhand der Markthistorie welche Geschäfte für den Händler stattgefunden hätten. Der Backtest arbeitet sich so durch die Historie, das Ergebnis ist ein zeitlicher Handelsverlauf für die Strategie, d.h. wir bekommen eine Auflistung aller getätigten Geschäfte die die Strategie ausgelöst hat. Diesen Handelsverlauf evaluiert der TA anschließend, z.B. durch Betrachten des Profits über einen Zeitraum, um dessen Performanz zu bewerten.

1.2 Misstand

Das Problem liegt darin, dass der TA, um eine wirtschaftliche Handelsstrategie zu entwickeln, einen Bedarf an Backtests pro Tag hat, welche das Overfitprojekt nicht decken kann.

Ein einzelner Backtest braucht lange, das Auswerten einer Handelsstrategie auf den Daten der letzten 15 Jahren kann in unserem Fall mehrere Minuten dauern, wir rechnen zur Einfachheit mit 2min pro Backtest. Um eine Handelsstrategie zu finden, die wirtschaftlich rentabel sein kann, müssen wir sehr viele Konfigurationen austesten. Pro Tag sollten mind. 50 verschiedene Konfigurationen ausgeführt werden können, das sind 100min pro Tag. Die Handelsstrategie wird auf verschiedenen Aktienmärkten, welche anhand verschiedener Aspekte relevant sind, getestet, derzeit 20. Auf diesen Märkten wollen wir immer alle Konfigurationen testen. Insgesamt resultiert das in 2000min an Backtest die pro Tag ausgeführt werden müssen, das sind über 33h. Diese Zahl ist höher als der Tag Stunden hat, also muss die Ausführung all dieser Chartanalysen beschleunigt werden.

Um die Backtests mit Overfit auszuführen, muss der Händler am Computer sitzen, die Analyse manuell starten und die Ergebnisse ebenfalls manuell abspeichern. Da der Händler nicht an beliebigen Stellen am Tag, für eine beliebige Dauer, Zeit vor seinem PC hat, schränkt dies das verfügbare Zeitfenster weiter ein.

Ein weiteres Bedenken des TA sind die laufenden Kosten sowie Wartungskosten für eine Lösung des Problems. Durch höhere Kosten ist es noch schwieriger eine wirtschaftliche Handelsstrategie zu entwickeln.

1.3 Ziel

Es folgt eine kurze Beschreibung der Lösung.

Das Starten einer Chartanalyse kann, auch unterwegs, durch Verschieben einer Google Docs Datei, welche die Konfiguration enthält, in einen Google Drive Ordner ausgelöst werden. So kann der TA die Analysen über den Tag starten und sie dann am Abend auswerten. Für die Ausgabe der Chartanalysen werden Google Sheets Dateien verwendet, welche der Analyst, zu Hause oder ebenfalls unterwegs, einsehen kann. Chartanalysen verschiedener Konfigurationen werden parallel ausgeführt, indem mehrere Prozessorkerne eines Servers benutzt werden und insgesamt mehrere Server benutzt werden. Das parallele Ausführen verkürzt den Zeitbedarf der täglichen Analysen bei Nutzen von 6 Prozessorkernen auf 6h. Die Server verwenden eine relationale Datenbank um sich zu koordinieren. Damit wird mehrfaches Ausführen einer einzelnen Chartanalyse verhindert welches den Zeitbedarf erhöht und Ressourcen verschwendet. Das System verhindert den Verlust von Chartanalysen, bei Ausfall eines Servers.

Der Titel der BA beschreibt diese Lösung in seinen grundlegenden Zügen. Anhand des Titels wird im folgenden nochmal die Lösung erläutert.

“Infrastruktur-minimales paralleles Backend für Offline-Chartanalysen”

Infrastruktur-minimal bedeutet, dass die Komplexität in Implementierung und Umfang von Ressourcen und Komponenten minimal bleiben soll. Eine weniger komplexe und günstigere Lösung ist einer anderen zu bevorzugen. Dies hält vermeidbare Kosten auf einem Minimum um die Wirtschaftlichkeit des Tools zu erhalten.

Offline-Chartanalysen bedeutet, dass der Nutzer während der Analyse nicht mit der Applikation interagieren muss, deswegen offline.

Paralleles Backend bedeutet, dass anstatt eine lokale Applikation zum Starten und Beenden eine kontinuierlich laufende Applikation auf einem oder mehreren entfernten Servern läuft. So ist die Applikation aus der ferne, quasi immer zu erreichen. Zu dem führt das Backend die Chartanalysen nicht nur sequentiell aus, sondern besitzt die Fähigkeit mehrere Chartanalysen parallel auszuführen.

1.3.1 Abgrenzung, Vorgaben, Rahmenbedingungen

Der Arbeit liegen folgende Vorgaben vor, welche vorab aus Präferenz und dem bestehenden Umfeld der Arbeit hervorgingen.

Der Ort für die Eingabe ist ein Google Drive Ordner. Durch das Verschieben einer Google Doc Datei, mit Konfiguration einer Chartanalyse in YAML-Format, erfolgt die Eingabe. Nachdem eine Chartanalyse bearbeitet wurde wird das Ergebnis, in Form einer Google Sheets Datei, in einen Ausgabeordner gelegt. Die zugehörige Eingabedatei wird ebenfalls nach Bearbeitung in einen Ausgabeordner abgelegt. Die Ausgabeordner müssen nicht zwingend der selbe Ordner sein. Die Google Drive Ordner werden in der Applikation konfiguriert.

Es besteht bereits eine relationale Datenbank, welche genutzt werden kann. Bei der Wahl gebrauchter Infrastrukturkomponenten wird diese damit ähnlichen Lösungen bevorzugt.

1.4 Aufbau der Arbeit

Im ersten Einleitungskapitel finden wir Herleitung des Problems, eine Formulierung des Misstands, Definition des Ziels und Vorgaben der Arbeit.

Im zweiten Kapitel, dem Hintergrund, findet man Fachwissen, Information über genutzte, oder ebenfalls relevante, technische Frameworks. Ebenso enthalten sind Erklärungen zu Begriffen der Performanz & Parallelität, auf welche wir uns dann im dritten Kapitel beziehen. Das zweite Kapitel stellt seine Themen nicht in vollem Umfang dar, stattdessen bietet es kurze Erklärungen und fokussiert sich auf die Punkte auf welche in späteren Kapiteln aufgegriffen wird.

Im dritten Kapitel, Stand der Technik & Forschung, werden Wissensbereiche in der Informatik im Bezug auf unsere Problemstellung vorgestellt. Das Kapitel bietet Probleme und Lösungsansätze aus dem Bereich der Wissenschaft. Auf Grundlage dieser Forschung können wir dann in den folgenden Kapiteln die gewählten Lösungsalternativen begründen und bewerten. Das dritte Kapitel endet mit Vorüberlegunen und Hinleitung zu den Lösungsalternativen. Hier werden Aspekte und Fragestellungen, welche die Lösungsalternativen beachten müssen definiert. Ebenso werden hier wenige grundlegende Entscheidungen getroffen, in welchen sich die Lösungsalternativen nicht unterscheiden werden.

Kapitel vier, fünf und sechs, beinhalten jeweils eine Lösungsalternative, welche dort, ohne Beachtung der exakten technischen Umsetzung, vorgestellt und erläutert werden.

In Kapitel Sieben werden die Lösungsalternativen gegenübergestellt, bewertet und die Entscheidung für die Lösung aus Kapitel Fünf begründet.

In Kapitel Acht wird die Implementierung der Lösung beschrieben. Hier sind Auffälligkeiten und Shortcomings der gewählten Implementierung beschrieben.

Im Kaptiel Neun wird das Ergebnis bewertet und ein Ausblick gegeben.

Kapitel 2

Hintergrund

2.1 Aktienhandel & Chartanalysen

Der Aktienhandel funktioniert weitestgehend wie folgt, ein Teilnehmer des Handels, z.B. eine Person, kann eine Bestellung zum Kauf einer Aktie aufgeben, besitzt der Teilnehmer die Aktie kann er die Aktie(n) im Besitz stattdessen zum Verkauf anbieten. Die Bestellungen werden bei einem Broker aufgegeben, dieser führt Paare geeigneter Bestellungen aus, es kommt zum Geschäft.

2.1.1 Kerzen & Charts

Charts halten den Preis der ausgeführten Geschäfte eines Aktienmarktes zeitlich fest. Wenn man jedes einzelne ausgeführte Geschäft des Marktes grafisch darstellt, dann erhält man einen Punktgraphen, siehe Abbildung 2.1.

Da bei großen Aktien, wie bspw. Amazon, dutzende Millionen Geschäfte pro Tag ausgeführt werden [2], wäre die Datenmenge schwer handhabbar und teuer. So werden die Geschäfte über eine Zeitspanne zu einer sog. Kerze aggregiert. Eine Kerze besitzt im wesentlichen 4 Datenpunkte, Open, High, Low, Close. Open ist der Preis zu dem das erste Geschäft in der Zeitspanne ausgeführt wurde, Close der Preis des letzten Geschäfts. High ist der höchste Preis aller Geschäfte der Zeitspanne, Low ist der niedrigste Preis aller Geschäfte der Zeitspanne. Ist Open höher als Close spricht man von einer fallenden oder auch negativen Kerze, ist Open niedriger als Close spricht man von einer steigenden oder auch positiven Kerze.

In folgender Abbildung 2.1 ist ein Punktgraph einiger theoretischer Geschäfte innerhalb einer Zeitspanne aggregiert und als Kerze dargestellt. Auf der X Achse ist die Zeit abgetragen, auf der Y Achse der Preis. Die aggregierte Kerze befindet sich rechts auf der Abbildung.

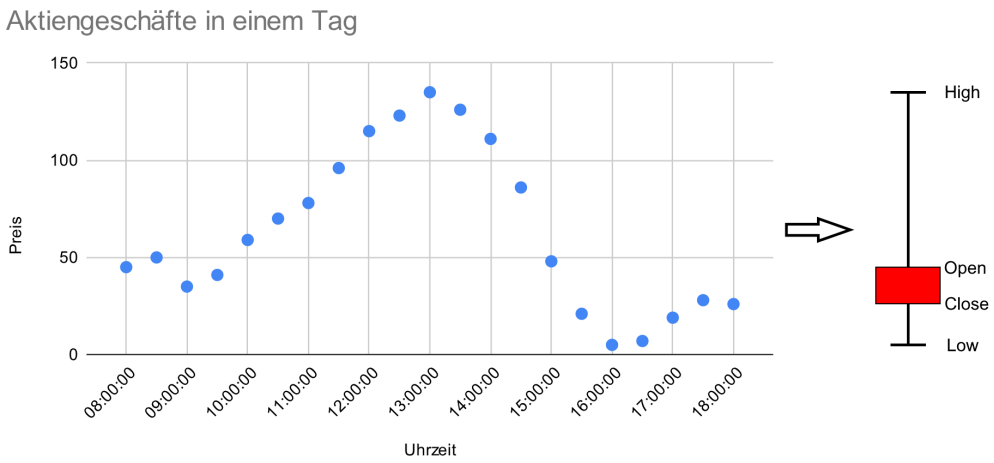

Abbildung 2.1: Aktiengeschäfte in einem Tag zu einer Tageskerze aggregiert

Reiht man mehrere dieser Kerzen aneinander erhält man einen Kerzengraph, das ist ebenfalls ein Chart. Häufig anzutreffen sind Daycharts, bei welchen die Zeitspanne für jede Kerze ein voller Tag ist.

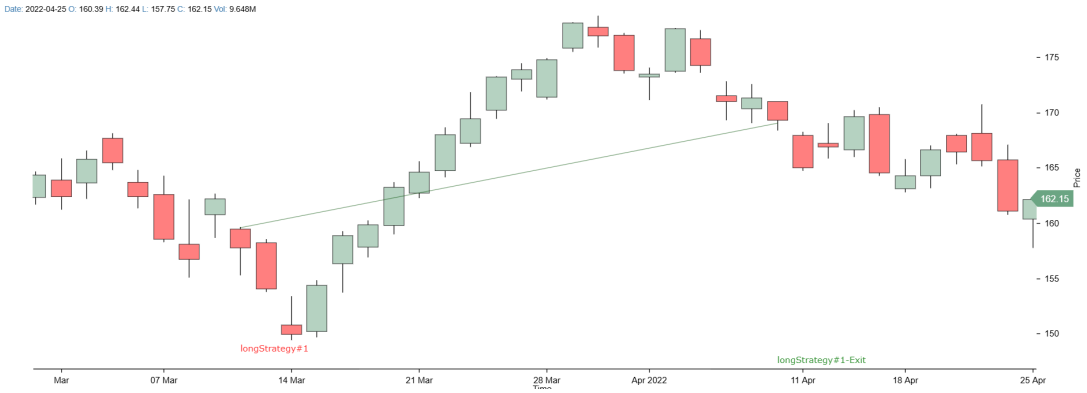

Abbildung 2.2: Kerzenchart einer Aktie

2.1.2 Technische Analysten (TA) & Handelsstrategie Overfit Modellierung

TAs betreiben Aktienhandel und legen ihren Entscheidungen im Handeln, Charts über vergangene Preisverläufe und andere Statistiken der Aktienmärkte zu Grunde. Das Wählen des Bestellpreises und des Bestellzeitpunktes modellieren TAs in Form von Handelsstrategien. In einer einzelnen Handelsstrategie finden wir sog. Regeln für "Wann geben wir eine Bestellung auf?" und Berechnungsformeln, sog. Indikatoren, für "Wie hoch ist der Preis der Bestellung?". Die Regeln und Indikatoren sind auf Zahlenwerten des Charts definiert.

Das interne Projekt Overfit stellt TAs bereits Mechanismen zum Entwickeln und Auswerten von Handelsstrategien bereit. Dabei sind in Overfit Algorithmen implementiert zum Ausführen verschiedener Chartanalysen. Overfit bietet ebenfalls eine eigene Modellierung der Handelsstrategie. Die Handelsstrategie in Overfit besteht aus einem konfigurierbaren Gerüst. Das bedeutet es ist vorgeschrieben wie einzelne Regeln und Indikatoren logisch ineinandergreifen, jedoch sind die Regeln und Indikatoren austauschbar und durch das kombinieren von konkreten Regeln und Indikatoren erreicht man so große Vielfalt.

Der TA definiert eine gute Handelsstrategie als eine, welche Bestellungen ausgibt die zu profitablen Geschäften führen.

In Overfit wird die Strategie durch eine Datenklasse repräsentiert, welche abstrakte Indikatorklassen und abstrakte Regelklassen beinhaltet. Der Nutzer fügt dann konkrete Indikatorklassen und konkrete Regelklassen ein und erhält so eine konfigurierte Handelsstrategie, welche Eingabe für Overfits Chartanalysen ist. Overfit bietet ebenfalls einen Deserialisierungsmechanismus, welcher aus einer geeigneten YAML-Datei eine Instanz einer solchen Strategieklasse erzeugen kann.

2.1.3 Backtest & Chartanalysen

Um beurteilen zu können ob eine Handelsstrategie gut ist, betrachtet ein TA dessen Performanz wenn er sie in der Vergangenheit angewandt hätte. Dies ermöglicht ihm ein Algorithmus, der sog. Backtest. Overfit bietet eine Implementierung dieses Algorithmus die es erlaubt konfigurierte Overfitstrategien auf vergangenen Kerzengraphen von Aktienmärkten auszuführen.

Der Backtest simuliert einen Händler der seine Strategie befolgt, d.h. er gibt Bestellungen auf, wenn die zugehörige Regel zutrifft und der Backtest simuliert dann anhand der Markthistorie welche Geschäfte für den Händler stattgefunden hätten. Der Backtest arbeitet sich so durch die Historie, das Ergebnis ist ein zeitlicher Handelsverlauf für die Strategie, d.h. wir bekommen eine Auflistung aller getätigten Geschäfte welche die Strategie ausgelöst hat. Diesen Handelsverlauf evaluiert der TA anschließend, z.B. durch Betrachten des Profits über einen Zeitraum, um dessen Performanz zu bewerten.

Weitere Chartanalysen von Overfit umfassen die Parameteroptimierung und Hyperparameteroptimierung.

Die Parameteroptimierung eröffnet gegenüber einem Backtest die Möglichkeit die Overfitstrategie parametrisiert zu konfigurieren. Statt expliziten Zahlenwerten innerhalb eines Indikators oder Regel, kann eine Spanne angegeben werden. Die Parameteroptimierung, vereinfacht formuliert, führt den Backtest für alle Parameterkombinationen aus und gibt die beste Kombination zurück. Die Parameteroptimierung beinhaltet neben zusätzlicher fachlicher Logik auch Optimierungen, wie Zwischenspeichern bereits ermittelter Indikator- und Regelwerten. Dennoch braucht die Parameteroptimierung deutlich mehr Rechenzeit.

Die Hyperparameteroptimierung besitzt nun neben Parametern in der Strategie noch Parameter für verschiedene logische Aspekte der Parameteroptimierung, dementsprechend ist die benötigte Laufzeit für so eine Chartanalyse nochmals höher.

2.2 Technische Frameworks

2.2.1 Spring Framework

Das Overfitprojekt und ebenso das entwickelte Task-Processing Projekt dieser Arbeit ist in Java17 mit dem Spring Framework implementiert. In diesem Kapitel wird der Kern des Spring Frameworks so wie benötigte Begrifflichkeiten erläutert.

Eine genutzte Erweiterung des Spring Frameworks ist Spring Boot, mit dieser kann man “ganz einfach eigenständige, produktionsreife Spring-basierte Anwendungen erstellen, die [man] „einfach ausführen“ [kann]” [3]. Spring Boot übernimmt viel Konfiguration und Boilerplate Code, den man als Entwickler zuerst selbst verstehen und generieren müsste.

Die wohl wichtigste und zentrale Funktionalität die das Spring Framework uns bietet ist die Dependency Injection in Verbindung mit dem IoC Container (Inversion of Control).

Mit Hilfe der Dependency Injection kann eine Komponente, wie beispielsweise eine Klasse welche ein Stück Businesslogik ausführt, seine benötigten Abhängigkeiten, meist weitere ähnliche Komponenten, deklarieren. Konkret wird dies mit der Java Annotationen `@Autowired` an Feldern, Konstruktoren oder Setter-Methoden erreicht. Das Spring Framework sorgt während der Laufzeit dafür dass alle diese Abhängigkeiten instanziert werden und der Komponente zur Verfügung stehen. Durch das Annotieren solcher Komponenten mit `@Component` wird dem Spring Framework vermittelt, dass Spring sich eine Instanz dieser Klasse erstellt und in seinem IoC Container hält, um sie für die Dependency Injection in einer anderen Komponente zu verwenden. Dadurch besteht die Inversion, eine Komponente muss ihre Abhängigkeiten nicht selbst besorgen sondern stattdessen übernimmt das Spring Framework diese Aufgabe.

Neben Komponenten mit Businesslogik, kann Spring ebenfalls sog. "Property Values", wir nennen sie Konfigurationswerte, von verschiedenen Quellen auflösen und Klassen bereitstellen. Dieses Feature nennt Spring Externalized Configuration [4]. Ein Konfigurationswert besteht immer aus einem Schlüssel und dem Wert. Diese Konfigurationswerte können Verhalten der Applikation beeinflussen oder Daten zur Anbindung an externe Dienste und Ähnliches beinhalten. Die Konfigurationswerte umfassen meist all die Werte die abhängig von der Installation der Applikation und dessen Umgebung sind. Bspw. gibt es oftmals einen Schlüssel mit der URL einer Datenbank welche die Applikation verwendet, dessen Wert enthält dann die tatsächliche Adresse des Servers. Diese Konfigurationswerte können der Applikation in verschiedenen Formen bereitgestellt werden. Bspw. als Kommandozeilenparameter, Umgebungsvariablen oder durch eine oder sogar mehrere Konfigurationsdateien. Konfigurationswerte kann man sich durch Annotieren eines Konstruktorparameters oder Klassenfelds mit `@Value` direkt von Spring auflösen lassen. Spring schaut in seinen Quellen in einer bestimmten Reihenfolge nach den gesuchten Schlüsseln und setzt den Wert in der Klasse dementsprechend.

Durch das Verwenden der `@Value` Annotation hat man in der Klasse eine statische Referenz auf den Schlüssel und Spring löst der Klasse den zugehörigen Wert automatisch zur Laufzeit auf. Möchte man den Wert ändern reicht es die application.yaml anzupassen, so muss der Code nicht angepasst werden und folglich nicht erneut kompiliert werden.

Unser Projekt verwendet ebenfalls dieses Feature in Form einer Konfigurationsdatei, die application.yaml, welche die Schlüssel-Wert Paare in YAML-Syntax enthält.

2.2.2 Spring Data

Spring Data ist eine Familie an Produkten welche ein auf Spring basiertes Programmiermodell für den Datenzugriff bereitstellt. Es vereinfacht die Nutzung von Datenzugriffstechnologien wie bspw. relationalen Datenbanken [5].

Unser Projekt verwendet dabei die Produkte Spring Data JDBC und Spring Data JPA in kombinierter Form.

Spring Data JPA (Java Persistence API) zielt darauf ab, die Implementierung von Datenzugriffsebenen deutlich zu verbessern, indem der Aufwand auf das tatsächlich erforderliche Maß reduziert wird [6].

Spring Data JDBC (Java Database Connectivity) bietet erweiterte Unterstützung für JDBC-basierte Datenzugriffsebenen. [7].

Mit den zwei Produkten können wir mit Leichtigkeit JPA Repository Interfaces anlegen und in diesen Methoden definieren. Ein JPA Repository ist ein Interface welches unter anderem CRUD Operationen für Datenzugriff bereitstellt. Diese Methoden spiegeln aus Sicht unserer Anwendung die Datenzugriffe dar, also finden oder modifizieren von Daten in der dahinter liegenden Datenbank. Für Parameter und Rückgabe dieser Methoden geben wir eigene Javaklasen an, so dass wir gekapselt vom Datenmodell der Datenbank sind. Diese Methoden können wir mit `@Query` annotatieren, in welcher wir ein gewünschtes

SQL statement als Stringparameter angeben können. In dem SQL statement können wird durch JDBC dann z.B. direkt die Parameter der Methode referenzieren.

Die Anwendung kann so den Datenzugriff rein deklarativ vorschreiben und Spring Data JPA und Spring Data JDBC übernehmen für uns die Arbeit Code zu generieren sowie technische Details zu klären.

2.2.3 Google API Client library

Um sehr einfach aus Javacode Google APIs diverser Produkte, wie bspw. Google Drive, zu benutzen integriert man die von Google bereitgestellte Google API-Clientbibliothek und die produktspezifischen Clientbibliotheken.

"Die Google API-Clientbibliothek für Java bietet gängige Funktionen aller Google APIs, z. B. HTTP-Transport, Fehlerbearbeitung, Authentifizierung, JSON-Parsing, Mediendownload/-upload und Batchverarbeitung. Die Bibliothek enthält eine leistungsstarke OAuth 2.0-Bibliothek mit konsistenter Schnittstelle, einfachen, effizienten XML- und JSON-Datenmodellen, die jedes Datenschema unterstützen, und Protokollzwischenspeicher" [8].

Zum identifizieren der Dateien für die einzelnen Googleprodukte besitzt jede Datei eine File ID, diese Identifikatoren sind einmalig und ändern sich nicht über den Lebenszyklus der Datei [9]. Wir schreiben die File ID als FileId.

Google Drive API

Für das Google Produkt Google Drive gibt es mehrere Clientbibliotheken für verschiedene Usecases, darunter das hier vorgestellte Google Drive API. Die Google Drive API kann "in Google Drive gespeicherte Dateien hochladen, herunterladen, freigeben und verwalten." [10]

Für die Dateien im Google Drive hat die API ein eigenes Datenmodell, darunter die bereits vorgestellte FileId so wie andere Metainformationen. Für uns relevant ist hier das `parents` Attribut, welches das Elternverzeichnis enthält. Ebenfalls das `mimeType` Attribut der Datei ist relevant, er enthält den MIME-Typen der Datei.

Zwei hier relevante Funktionen der API ist das suchen nach Dateien in einem Ordner und das Verschieben von Dateien.

Zum Suchen von Dateien gibt man Filter in Form von Queries an. Zurück kommen dann alle Dateien des Nutzers welche den Filtern entsprechen [11]. Damit können wir uns bspw. alle Google Docs Dateien eines Drive Ordners ausgeben lassen. Dazu muss man im Filter nur angeben, dass die Datei im Elternverzeichnis die FileId des Drive Ordners hat und der MIME-Typ dem einer Google Docs Datei entspricht.

Zum Verschieben einer Datei holt man sich zuerst dessen Metadaten über die FileId, modifiziert dann das Attribut `parents` lokal und schickt zuletzt ein Update mit den Modifikationen ab [12]. Aus dem parents Attribut werden zuerst das alte Elemente entfernt und danach der neue Ordner als Element eingefügt.

Google Docs API

Mit der Google Docs API kann man auf Google Docs Dateien zugreifen und diese verändern. Für uns wichtig ist die Funktion des Auslesens des Inhalts einer Google Docs Datei [13].

Die Datenstruktur einer Google Docs Datei ist komplex [14]. Die Komplexität der Datenstruktur stammt von den Features die eine Google Docs Datei über einer einfach Textdatei anbietet. Darunter die Dokumententabs, die Möglichkeit Medien und Tabellen einzubetten etc. So ist es nicht ganz trivial den gesamten Text aus einer Google Docs Datei zu extrahieren. Google bietet jedoch viele Beispiele die man zur Hilfe zuziehen kann [15].

Google Sheets API

Mit der Google Sheets API kann man Google Sheets Dateien erstellen und bearbeiten. Für uns wichtig sind die Funktionen eine neue Google Sheets Datei zu erstellen [16] und ebenso Inhalt in die Tabelle zu schreiben [17].

Google Cloud Webhook

"Mit Webhook wird die schlanke, eventgesteuerte Kommunikation bezeichnet, die automatisch Daten zwischen Anwendungen über HTTP sendet. Webhooks werden von bestimmten Events ausgelöst und automatisieren die Kommunikation zwischen APIs" [18]. Die Google Cloud Webhooks, ein Produkt von Google. Mit diesen kann man sich eventbasierte Push-Nachrichten, als Reaktion auf veränderte Ressourcen in der Google Drive Cloud einrichten. Dafür gibt man einen eigenen API-Endpunkt an, an welche die Google Drive API dann die Nachricht sendet und die Ressource welche man beobachten möchte, z.B. einen Google Drive Ordner [19].

2.2.4 Java Threads

Java Threads sind ein in Java integrierter Mechanismus um neue Threads in der JVM zu erstellen welche von der JVM verwaltet werden. Für den Entwickler ist dieses Feature ein einfacher, schneller Weg sein Programm nebenläufig zu machen. Die Arbeit beschränkt sich in der Erklärung auf die für uns relevanten Aspekte. Darunter der Aspekt wie man einen Thread korrekt unterbricht und beendet.

Wir gehen jedoch trotzdem kurz darauf ein wie man einen Javathread erstellt und ausführt. Der Entwickler bereitet ein `Runnable` vor, und gibt dieses dem Thread bei der Instanziierung. Anschließend kann der Thread mit `Thread.start()` gestartet werden, dann wird das `Runnable` nebenläufig ausgeführt. Ist das `Runnable` fertig ausgeführt wird der Thread automatisch wieder geschlossen. Mit `Thread.wait()` kann man auf das Ende des `Runnables` warten.

Um einen anderen Thread zu unterbrechen sendet man dem Thread ein Signal. Der Thread muss dann selbst auf dieses Signal reagieren, meistens bricht der Thread sein Tun ab und endet. Die Threadklasse bietet bereits mit der Instanzmethode `Thread.interrupt()` ein Mechanismus um ein Signal zum unterbrechen des Threads zu senden. Dabei wird im Thread die `interrupted` Flag gesetzt. Der Thread kann mit `Thread.interrupted()` den Status dieser Flag abfragen [20]. Da der unterbrochene Thread selbst auf die Flag reagiert hat er die Möglichkeit noch Code auszufüren bevor er endet. Ein Thread sollte vor dem beenden von ihm gesperrte Ressourcen freigeben, Verbindungen zu externen Systemen schließen und keinen inkonsistenten Zustand in Daten hinterlassen.

In unserem Projekt verwenden wir zum Stoppen eines Threads nicht die interrupted-Flag der Threads selbst, sondern imitieren den Mechanismus durch das Verwenden eines `AtomicBoolean` als Flag. `AtomicBoolean` ist Teil des `concurrent.atomic` packages von java und gibt uns eine lockfreie, thread-safe Implementierung für Boolean [21]. Dieser Aspekt ist wichtig für uns, da ein Thread die Flag setzen wird und ein anderer Thread die Flag prüfen wird. In unserer Logik verwenden wir diese Flags um Threads zu signalisieren, dass sie ihr Tun stoppen sollen. Wir verwenden nicht den Unterbrechungsmechanismus der Javathreads selbst sondern unsere eigene Implementierung um nicht von der Handhabung der interrupted-Flag in anderem Code abhängig zu sein. Ein weiterer Grund ist dass im JDBC-Entwicklerhandbuch von der Nutzung von interrupted abgeraten wird [22].

Eine eigene solche Flag zu implementieren scheint generell ein relativ akzeptierter Ansatz zu sein. Ein

gezwungenes Ende, wie durch die deprecated `Thread.stop()` Methode, ist wie erwähnt wegen Freigeben von gesperrten Ressourcen etc. nicht empfohlen.

Alternativen zum manuellen Erstellen und Starten von Threads ist durch das Verwenden von `ThreadPoolExecutor` oder anderen Executor-Services möglich.

Der `ThreadPoolExecutor` ist vom `concurrent` Package von Java und bietet einen einfachen Weg einen Threadpool zu erstellen welcher gegebene Aufgaben auf den Threads des Pools erledigt. Der `ThreadPoolExecutor` ist eine Erweiterung der Java `Executor` Klasse [23]. Ein Executor bietet die Methode `void Executor.execute(Runnable r)` mit welcher sehr einfach Aufgaben asynchron ausgeführt werden können. Im Falle des `ThreadPoolExecutor`s laufen diese Aufgaben auf den Threads des Pools.

2.2.5 ActiveMQ

ActiveMQ ist ein Produkt von Apache, ein OpenSource java-basierter Nachrichten-Broker [24]. Beim Kommunizieren mehrerer Systeme untereinander kann ActiveMQ als Kommunikationsbrücke zwischengeschalten werden [25].

In ActiveMQ können Nachrichtenwarteschlangen angelegt werden. In eine Nachrichtenwarteschlange können Nachrichten eingefügt werden und einzeln herausgenommen werden [25]. ActiveMQ dient so als Puffer für die Nachrichten. Wenn die Konsumenten langsamer im Bearbeiten der Nachrichten sind, als Nachrichten produziert werden, dann sammeln sich die Nachrichten im Puffer. So gehen keine Nachrichten verloren, wenn der Puffer groß genug ist. ActiveMQ kümmert sich um konsistenten Zugriff auf die Nachrichtenwarteschlangen, unterstützt Acknowledgments der Nachrichten und kümmert sich ebenfalls um die Persistenz der Nachrichten.

Alternativ zu Nachrichtenwarteschlangen können Themen angelegt werden um einen Publish-Subscribe Pattern umzusetzen [25]. Dabei dient das Thema lediglich dazu um den Nachrichtenkanal zu identifizieren. Anwendungen können sich auf Themen abonnieren. Eine Anwendung kann eine Nachricht zu einem Thema veröffentlichen und diese wird an alle Abonnenten des Themas weitergeschickt.

2.2.6 PostgreSQL Datenbank

"PostgreSQL ist ein leistungsfähiges, objektrelationales Open-Source-Datenbanksystem, das die SQL-Sprache verwendet und erweitert, kombiniert mit vielen Funktionen, die eine sichere Speicherung und Skalierung der kompliziertesten Daten-Workloads ermöglichen" [26, übersetzt mit DeepL]. Die relationale Datenbank die dem Projekt zur Verfügung gestellt wird ist eine PostgreSQL Datenbank. Im Verlaufe der Arbeit wird PostgreSQL erwähnt, jedoch gehen wir nicht auf Details von PostgreSQL ein.

2.3 Performanz & Parallelität

Performanz ist ein Bewertungskriterium, um Lösungen oder Systeme miteinander zu vergleichen. Um Performanz zu quantisieren gibt es verschiedene Performanzmetriken. In unserer Analyse haben wir die drei folgenden Performanzmetriken aufgestellt.

2.3.1 Performanzmetriken

- **Durchsatz:** Der Durchsatz errechnet sich aus der Menge an Arbeit welche in einer Zeitspanne abgearbeitet wurde. Hat ein System einen höheren Durchsatz als ein Alternatives, so kann dieses

System mehr Arbeitsmenge in der gleichen Zeit abarbeiten oder die gleiche Arbeitsmenge in weniger Zeit abarbeiten hat.

- **Latenz:** Latenz beschreibt die Dauer die ein System braucht zum Bearbeiten einer Arbeitseinheit. Eine Arbeitseinheit kann bspw. ein Task sein. Beginnend beim Zeitpunkt zu dem die Arbeit an das System übergeben wurde, endend wenn die Ausgabe fertig ist.
- **Ressourcenauslastung:** Ressourcenauslastung beschreibt das Verhältnis zwischen Ressourcen welche aktiv sind gegen die Ressourcen die inaktiv sind. Ressourcen die inaktiv sind verrichten keine Arbeit.

Meist möchte man die Ressourcenauslastung eines Systems möglichst hoch halten, jedoch kann sich eine niedrigere Ressourcenauslastung lohnen, wenn es den Durchsatz oder die Latenz verbessert.

2.3.2 Parallelität und Nebenläufigkeit

Unter Nebenläufigkeit und Parallelität versteht man die Eigenschaft eines Systems mehrere Aufgaben, Berechnungen oder Anweisungen gleichzeitig ausführen zu können [27]. Dabei bezeichnet Parallelität eine Nebenläufigkeit bei der zeitgleich mehrere Anweisungen, auf mehreren Prozesorkernen ausgeführt werden. Auf einem einzelnen Prozessorkern kann durch Multitasking ebenfalls Nebenläufigkeit erreicht werden kann.

2.3.3 Amdahl's Law

Amdahl's Law beschreibt die maximale Verkürzung der Ausführungszeit eines Problems durch parallele Ausführung. Je mehr Ausführungszeit eingespart werden kann, desto größer die Beschleunigung.

Dabei teilt man das Problem in einen sequentiellen und parallelen Teil. Je größer der parallele Teil ist, desto größer die Beschleunigung. Der parallele Teil besteht aus Teilen des Problems welche getrennt voneinander, ohne gegenseitige Abhängigkeiten, parallel, ausgeführt werden können.

In folgender Abbildung 2.3 sieht man für unterschiedliche Größen des parallelen Teils die potentielle Beschleunigung nach Amdahl's Law.

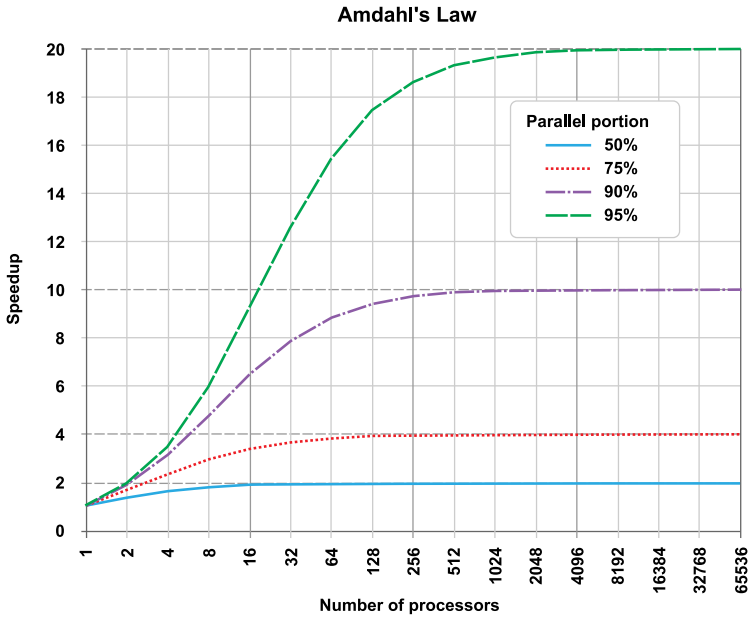

Abbildung 2.3: Amdahl's Law [28]

Kapitel 3

Stand der Technik & Forschung

In diesem Kapitel werden verschiedene Wissensbereiche der Informatik vorgestellt. Die Wissensbereiche sind eine von vier Themenbereichen eingegliedert. Die Themenbereiche dabei dienen zur Struktur und dem Einordnen verschiedener Wissensbereiche und sind selbst keine wissenschaftlich fundierten Themengebiete. Jeder Themenbereich beschäftigt sich mit einer zentralen Frage bzw. einer Problematik und enthält die relevanten Lösungskonzepte der Informatik. Die vorgestellten Wissensbereiche werden mit Fokus auf unseren Anwendungsfall behandelt. Die BA beschäftigt sich mit paralleler Programmierung einer Anwendung, welche auf verteilten System eingesetzt werden kann.

3.1 Wie wird höhere Performanz erreicht?

Die Frage dieser Sektion zielt auf den Themenbereich der Parallelverarbeitung ab. Parallelverarbeitung bezeichnet parallele Datenverarbeitung. Datenverarbeitung bezeichnet dabei ein Programm oder mehrere Programme, welche Operationen auf Datenmengen ausführen und Ergebnisse erzielen. Mit Parallelverarbeitung kann höhere Performanz in Form von kleinerer Latenz und höherem Durchsatz erreicht werden.

Zentral ist dabei, dass wir mehrere Threads haben, welche parallel auf mehreren Prozessorkernen ausgeführt werden.

Wir wollen uns mit der Parallelverarbeitung nicht aus Sicht eines Betriebssystementwicklers oder Hardwareingeneur betrachten. Wir setzen moderne Prozessorarchitektur und moderne Betriebssysteme vorraus.

Um parallele Verarbeitung zu erreichen muss ein Gesamtproblem zunächst in kleinere parallel ausführbare Teilprobleme zerlegt werden. Das bedeutet anstatt einen durchgehenden einzelnen Ausführungsstrang, teilt sich dieser an manchen Stellen in viele Ausführungsstränge auf, welche jeweils ein Teilproblem lösen. Diese Ausführungsstränge müssen nun noch zeitgleich auf verschiedenen Prozessorkernen laufen um eine parallele Verarbeitung zu erreichen. Mit Ahmdahl's Law (2.3.3) kann man dann die maximale Beschleunigung durch die Parallelverarbeitung bestimmen. Die Beschleunigung spiegelt sich in der Performanz durch höheren Durchsatz und niedrigere Latenz wieder.

Die Ausführungsstränge kann man entweder auf den Prozessorkernen eines einzelnen Systems verteilen, daraus wird dann ein paralleles System [29]. Oder man verteilt die Ausführungsstränge auf verschiedene eigenständige Systeme ein sog. verteiltes System [29]. Eine Kombination beider Varianten ist ebenso möglich, verteilte parallele Systeme in welchem jedes verteilte System wieder ein paralleles System ist. Diese hybride Variante verbindet beide Architekturen um die Stärken beider auszunutzen.

Parallele Systeme fokussieren sich auf hohe Nebenläufigkeit, effiziente Koordination zwischen Prozessorkernen und effiziente Ausnutzung geteilter Ressourcen [29]. Parallele Systeme haben drei Kategorien, Systeme mit geteiltem Hauptspeicher, Systeme mit verteiltem Hauptspeicher und hybriden Systemen welche die zwei anderen Kategorien verbindet [30, Seite 33]. Die Threads tauschen Informationen aus, um sich zu koordinieren und in allen drei Kategorien ist dieser Informationsaustausch im Vergleich zu verteilten Systemen sehr schnell und weniger komplex. Praktische Anwendungsbeispiele für solche parallele Systeme findet man z.B. im Bereich wissenschaftlicher Simulationen und Bildverarbeitung.

Verteilte Systeme legen neben paralleler Verarbeitung ihren Fokus auf Robustheit, Skalierbarkeit und effiziente Ressourcennutzung [29]. Verteilte Systeme tauschen Informationen zur Koordination über das Netzwerk aus, was langsamer ist, aber erlaubt die Systeme ebenfalls geografisch zu verteilen. Diese Verteilung erlaubt es ebenso auf einzelne Systemausfälle zu reagieren und das System zu skalieren.

3.1.1 Skalierung in verteilten Systemen

Skalieren erlaubt es ein System der Arbeitslast anzupassen, so dass die Performanz immer möglichst hoch gehalten wird. Als Hochskalieren bezeichnet man im Allgemeinen ein Erhöhen der eingesetzten Ressourcen. Durch die extra Ressourcen erzielt man eine höhere Arbeitsleistung. Durch Runterskalieren, also Reduzieren der eingesetzten Ressourcen, kann man in den meisten Fällen überflüssige Ressourcen einsparen.

Es gibt zwei grundlegende Dimensionen zu skalieren, horizontale Skalierung und vertikale Skalierung.

Man kann ein System horizontal skalieren, das bedeutet meistens dass man die Anzahl an Prozessorkernen im System erhöht. Genereller bezeichnet man mit horizontaler Skalierung das Replizieren einzelner Komponenten, wodurch sich die Arbeit auf die replizierten Komponenten verteilt [31].

Vertikale Skalierung bedeutet eine einzelne Komponente leistungsfähiger zu machen, wodurch sie mehr Arbeit in weniger Zeit leisten kann. Als Beispiel könnte man Prozessorkerne gegen leistungsfähigere Prozessorkerne austauschen [31].

In verteilten Systemen skaliert man in den meisten Fällen durch das Hinzufügen und Entfernen weiterer gleicher Systeme. Ein gutes Beispiel für ein verteiltes System an welchem man Skalierung anschaulich darstellen kann kommt aus der Natur, die Honigproduktion von Bienen. Eine Biene stellt ein System dar und die Aufgabe des gesamten Bienenstocks ist die Honigproduktion. Jede Biene geht an eine andere Stelle um Pollen zu sammeln und bringt sie zum Bienenstock, jedes System arbeitet also eigenständig. Der Bienenstock ist ein zentraler Ort, wo alle Bienen ihre Pollen ablegen, der dann zu Honig verarbeitet wird. Die Systeme koordinieren sich über diese zentrale Instanz, das spiegelt ein Client-Server System wieder, welches man häufig in der Realität antrifft [29]. Möchte man mehr Honig produzieren, kann man weitere Bienen hinzufügen, welche dann insgesamt mehr Pollen sammeln und letztendlich mehr Honig herstellen. Analog gilt weniger Bienen, weniger Honig. Das Beispiel zeigt auch auf, dass eine unendliche horizontale Skalierung Probleme aufwirft. Der Bienenstock mit der Bienenkönigin kann nur eine begrenzte Anzahl an Bienen haben, bevor der Platz zum Honigmachen ausgeht. In diesem Fall muss der Bienenstock vertikal skaliert werden, in unserem Beispiel z.B. räumlich vergrößert werden. Alternativ könnte man auch den Bienenstock horizontal skalieren, also einen weiteren Bienenstock hinzufügen. Bei dieser alternativen Lösung muss man dann jedoch andere Probleme lösen, wie bspw. die Frage wo eine einzelne Biene seine Pollen hinbringt, auch analog in der Informatik kein triviales Belangen.

Horizontal zu skalieren ist einfach, man muss bestehende Systeme nicht verändern sondern man fügt modular Ressourcen hinzu oder entfernt sie. Durch Fortschritte in der Virtualisierung und Cloud-Computing ist der Prozess vom Ressourcen hinzufügen oder entfernen erst möglich und stark vereinfacht worden [32]. Cloud-Computing macht es trivial parallele Programme auszuführen, durch die verschiedenen Servicemodelle werden niedrigere technologische Schichten für den Nutzer komplett wegabstrahieren. So kann man bspw. durch Nutzen eines PaaS (Platform as a Service) mit Leichtigkeit von einer Applikation viele Instanzen starten und hat so ein leistungsstarkes verteiltes System.

3.1.2 Modelle der Parallelverarbeitung

Modelle der Parallelverarbeitung dienen zum Einordnen eines konkreten Problems in ein passendes Modell. Die Modelle betrachten verschiedene Arten wie ein Problem in Teilprobleme aufgeteilt und dann parallel verarbeitet werden kann. Da wir hier im Bereich der Parallelverarbeitung sind, eine Spezialform der Datenverarbeitung, ist ein Problem für uns eine Datenmenge auf welcher eine Reihe von Operationen ausgeführt werden muss.

Ein “Embarrassingly Parallel” Problem, bezeichnet ein Problem, bei welchem eine Unterteilung in Teilprobleme möglich ist, welche auf unterschiedlichen Daten unterschiedliche Operationen ausführen. Die Teilprobleme sind dabei völlig unabhängig voneinander, so dass keine Kommunikation zwischen parallelen Prozessen stattfinden muss. Dadurch kann eine maximale Beschleunigung durch die Parallelität erreicht werden [30, Seite 80]. Der Vorteil bei “embarrassingly parallel” Problemen ist, dass man die Daten so wie Programmcode komplett verteilen kann. Ein reales Beispiel wäre ein Webserver, welcher die Anfragen vieler Clients beantworten muss. Hierbei muss der Webserver je nach Anfrage andere Daten und Schritte ausführen um die Anfrage zu beantworten. [30, Seite 98]

Der “Task-Parallelismus” beschreibt ein Problem, bei welchem die Teilprobleme die Ausführung verschiedener Operationen auf der selben Datenmenge oder verschiedenen Teilmengen beschreibt. Es muss die Konsistenz und der konsistente Zugriff auf die Daten in allen Fällen abgesichert werden, da die Teilprobleme durchaus auf der gleichen Datenmenge zeitgleich arbeiten. Ein reales Beispiel könnte das Analysieren einer großen Datenmenge sein, wobei die Teilprobleme das Ausrechnen verschiedener Statistiken über die Datenmenge darstellen könnten [33].

Der “Daten-Parallelismus” beschreibt Probleme, bei denen gleiche Operationen auf verschiedenen Daten ausgeführt werden müssen. Dafür werden die Daten in Blöcke aufgeteilt und parallel verarbeitet. Ein reales Beispiel ist hier das Rasterisieren eines Bildes in der Computergrafik, bei welchem für jeden Pixel auf dem Bildschirm der gleiche Code zum Farbwert bestimmen ausgeführt wird [33].

Der “Pipeline-Parallelismus”, oder nur Pipeline, ist ein Modell für eine Teilmenge der Probleme welche sich in den Daten-Parallelismus einordnen. Auch hier werden gleiche Operationen auf verschiedenen Daten ausgeführt. Der Unterschied ist jedoch, dass bei der Pipeline die Operationen in einzelne Sequenzen, Pipelineschritte, aufgeteilt werden. Die komplette Datenmenge wird dann nach und nach durch die Pipeline geschickt. So werden auf jedem Datenpunkt die gleichen Operationen aufgerufen, jedoch ist die Ausführung auf die Pipelineschritte verteilt. Der Vorteil ist, dass der Ort an dem ein einzelner Pipelineschritt ausgeführt wird sich speziell auf seine Sequenz von Operationen konzentrieren kann und dadurch meist effizienter und schneller wird. Ein reales Beispiel findet sich in der Bildverarbeitung, bei welcher viele Transformationen über Bildabschnitte nacheinander passieren. Hier stellt eine einzelne Transformation einen Pipelineschritt in der gesamten Verarbeitungspipeline dar.

Das “Fork-Join Model” beschreibt die Art ein Problem in Teilprobleme aufzuteilen und wieder zusammenzuführen. Beim Fork-Join Modell beschreibt ein Fork-Schritt ein Problem in parallele Teilprobleme aufzuteilen, ein Join-Schritt beschreibt das wieder zusammenführen der Teilprobleme um sequentiell

fortzusetzen. Auf ein Fork-Schritt können weitere Fork-Schritte folgen um die Granularität des Problems weiter zu erhöhen. Durch das Fork-Join-Modell kann man potentiell Probleme mit hoher sequentieller Laufzeitkomplexität stark beschleunigen. Ein reales Beispiel findet man bei manchen Arten der Parameteroptimierung. Hier kann eine Auswertung auf allen Daten einer einzelnen Parameterwertekombination den ersten Fork-Schritt darstellen. Das Auswerten einer Parameterwertekombination auf einem einzelnen Datenpunkt wäre dann der zweite Fork-Schritt. Nach der Auswertung fügt man die Ergebnisse in zwei Join-Schritten wieder zusammen. Die potentielle Beschleunigung durch anwenden des Fork-Join-Modells ist hoch. Bei n vielen Datenpunkten und m vielen Parameterwertekombinationen hat das Problem eine sequentielle Laufzeit von $O(n \cdot m)$ hat, mit Anwenden des Fork-Join-Modells, mit $x > 1$ vielen Prozessorkernen wäre die Laufzeit dann, also in $O(n \cdot m/x)$, ein Bruchteil der sequentiellen Laufzeit.

3.2 Wie verteilt das System die Tasks?

Im Rahmen dieser Arbeit ist das Ziel ein System zu erstellen welches Tasks, also Arbeitspakete, erkennen und verarbeiten soll. In diesem Kontext stellen wir drei Fragen.

- Wie erkennt ein System neue Tasks, welche Möglichkeiten gibt es in unserem Kontext auf Nutzerinteraktion aufmerksam zu werden?
- Wo, auf welcher Ressource im System wird ein Task ausgeführt?
- Wann wird der Task ausgeführt?

Anhand eines kleinen Beispiels einer lokalen Anwendung widmen wir uns kurz diesen drei Fragestellungen. Eine Anwendung erkennt eine UI Nutzerinteraktionen und löst danach eine Kette an Methodenaufrufen aus. Diese verarbeiten die Eingabe und zeigen dem Nutzer potentiell ein Ergebnis an. In diesem trivialen Beispiel sind die drei gestellten Fragen leicht zu beantworten. Ein UI Event-listener erkennt die Eingabe und ruft einen Callback des Frontends auf, in welchem dann sofort an Ort und Stelle die Verarbeitung der Eingabe stattfindet.

Bei größeren Anwendungen auf verteilten Systemen sind die Antworten auf die drei Fragen deutlich komplexer. Einzelne Arbeitsschritte oder Verantwortlichkeiten des Systems sind auf verschiedene Anwendungen verteilt. Von den einzelnen Anwendungen gibt es oftmals mehrere Instanzen. Da die Arbeitsschritte auf potentiell replizierten Systemen ausgeführt werden, muss das gesamte System sich in einer Weise koordinieren um die gestellten Fragen zu beantworten.

3.2.1 Erkennen neuer Tasks

Wenn ein Nutzer eine Aktion ausführt und damit eine Eingabe tätigt, dann muss es im System einen Punkt geben, welcher diese Eingabe erkennt. Das Klicken eines UI Elements auf einer Benutzeroberfläche oder das Ablegen einer Datei an einen bestimmten Ort, können beides Aktionen sein mit welchem ein Nutzer eine Eingabe tätigt.

Zur Erkennung einer Eingabe gibt es im grundlegenden zwei unterschiedliche Methoden. Ein eventbasierter Ansatz in welchem der Empfänger eine passive Rolle spielt und der “Polling” Ansatz in der der Empfänger eine aktive Rolle spielt.

Im eventbasierten Ansatz gibt es einen passiven Empfänger und dieser wird von der Event Quelle aktiv mit neuen Events benachrichtigt. Ein Event bedeutet hier die Interaktion des Nutzers um eine Eingabe zu tätigen. Umsetzbar ist das mit dem publish-subscribe Pattern, auch oftmals Observer Pattern genannt [34, Seite 35]. Bei diesem meldet sich ein Empfänger bei der Eventquelle an. Die Empfänger

geben bei der Anmeldung i.d.R. einen Callback an die Eventquelle. Über diesen Callback ruft dann die Eventquelle den Empfänger direkt auf. Im Callback verarbeitet der Empfänger dann das neue Event. Bspw. bei einer Webanwendung ist ein Event der Klick auf ein UI-Element und die UI ruft dann den Callback des Frontends auf. Die UI ist dabei die Eventquelle und das Frontend der Empfänger.

Gegenüber dem event basierten Ansatz spielt bei Polling der Empfänger eine aktive Rolle und schaut aktiv nach neuen Eingaben. Wenn der Nutzer eine Eingabe tätigt muss sich der Zustand im System irgendwo ändern, eine neu abgelegte Datei oder Nachricht oder veränderte Werte in der Nutzeroberfläche. Der Empfänger erkennt dann Zustandsänderungen durch konsekutive Anfragen und erkennt damit die Nutzereingaben. Das System schaut periodisch in einen Dateiordner und merkt sich die enthaltenen Dateien, dann erkennt das System wenn ein Nutzer eine Datei ablegt wenn es das nächste mal nachsieht.

3.2.2 Lastenverteilung

Die Lastenverteilung(LV) behandelt die Frage, wo ein Task verarbeitet wird. Das bedeutet auf welcher Rechenressource wird der Programmteil, der den Task verarbeitet ausgeführt. Das Ziel der Lastenverteilung ist es die Tasks auf die verarbeitenden Ressourcen so zu verteilen dass ein möglichst hoher Durchsatz und eine möglichst hohe Ressourcenauslastung erreicht wird [35, Seite 50].

Im Kontext der Arbeit beschäftigen wir uns konkret damit, wie Lastenverteilung und im nächsten Teil Task-Scheduling in modernen Cloudumgebungen umgesetzt wird. Unsere Software wird auf verteilten Systemen parallel ausgeführt. Es ist nicht Teil der Arbeit wie ein Betriebssystem seine Prozesse und Rechenressourcen verwaltet, diese Low-Level Konzepte sind für uns weitestgehend transparent. Wir nehmen vereinfachend an auf einem System mit n Prozessorkernen, dass n viele Threads unserer Anwendung parallel ausgeführt werden. Ein solcher Thread / Prozessorkern ist eine Rechenressource für uns.

Die Arbeitslast die ein System verteilen muss teilt sich in drei Kategorien, Hauptspeicherlast, CPU-Last und Netzwerklast [36, Seite 11]. Aufgrund unserer lang laufenden Algorithmen der Chartanalysen ist in unserem Kontext die Verteilung der CPU-Last die Wichtigste. Die Hauptspeicherlast einer Chartanalyse stellt in unserem Kontext kein großes Problem dar. Wir nehmen an dass unsere Systeme mit ausreichend Hauptspeicher ausgestattet sind, so dass selbst die Last mehrerer Chartanalysen auf einem System nicht zu einem Flaschenhals führt. Die Netzwerklast ist durch die niedrige Anzahl an zu verarbeitenden Arbeitspaketen, den Tasks, ebenfalls nicht problematisch in unserem Kontext.

Unser Fokus liegt also darauf, welcher Prozessorkern in unserem verteilten System führt welchen Task aus und wie wird diese Entscheidung getroffen. Wir betrachten im folgenden verschiedene Klassifikationen und Eigenschaften von Lastenverteilungsverfahren.

Lastenverteilung kann statisch oder dynamisch erfolgen [30, Seite 202].

Bei statischer Lastenverteilung wird die Verteilung der Tasks auf die Rechenressourcen, bereits bevor ein Task ausgeführt wird, durchgeführt [30, Seite 202]. Ein Beispiel für statische Lastenverteilung ist das Round-Robin Verfahren, bei welchem die Tasks rundherum an die Rechenressourcen verteilt werden. Statische Lastenverteilungsverfahren können nur auf den verfügbaren Informationen vor der Ausführung aufbauen. Um eine gute Lastenverteilung zu erreichen brauchen sie detailiertes Wissen über die Arbeitslast, wie lange einzelne Tasks in der Ausführung brauchen, wie leistungsfähig die einzelnen Rechenressourcen sind usw. Da statische Verfahren keine Möglichkeit zur Verwendung von Laufzeitdaten haben, wie aktuelle Ressourcenauslastung können statische Verfahren nicht auf Veränderungen während der Laufzeit reagieren. Aktuelle Netzwerkauslastung, darunter Netzwerklatenz ist z.B. vor Laufzeit kaum vorhersehbar. Ebenso kann durch andere Prozesse auf einem System die Leistung von manchen Rechenressourcen einbrechen,

worauf ein statisches Verfahren nicht reagieren kann. Auch die erwartete Laufzeit der Tasks muss sehr akkurat sein um eine gute statische Verteilung zu ermöglichen. Viele Algorithmen, z.B. Suchalgorithmen, haben eine unbekannte Anzahl an auszuführenden Schritten was eine statische Lastenverteilung in diesen Fällen ineffektiv macht. Im Kontext der Arbeit, ist uns die Tasklänge unbekannt und wir benötigen deshalb für eine gute Lastenverteilung einen dynamischen Ansatz, so betrachten wir im Folgenden dynamische Lastenverteilungsverfahren.

Gegenüber der statischen Lastenverteilung haben wir die dynamische Lastenverteilung, welche während der Ausführung der Tasks stattfindet. Dadurch kann das Lastenverteilungsverfahren Informationen über aktuelle Auslastung der Ressourcen miteinbeziehen und andere Laufzeitdaten verwenden. Trotz des extra Overheads in der Laufzeit um die Lastenverteilung zusätzlich auszuführen, ist die dynamische Lastenverteilung oftmals effektiver [30, Seite 203].

Dynamische Lastenverteilung kann entweder zentralisiert oder dezentralisiert ausgeführt werden, auch nicht-verteilt und verteilt genannt [30, Seite 203] [37, Seite 155]. Bei zentraler / nicht-verteilter Lastenverteilung ergibt sich ein Master-Worker Pattern. Der Master hält die Menge an offenen Aufgaben und verteilt diese an seine Worker und ist alleine für eine gute Lastenverteilung verantwortlich [30, Seite 204]. Der Master wird auch manchmal Dispatcher (deutsch Verteiler) genannt, da dieser oftmals die Tasks aktiv an seine Arbeiter verteilt. Im Gegensatz dazu, führt bei der dezentralen / verteilten Lastenverteilung jedes System den Algorithmus zur Lastenverteilung aus und die Verantwortlichkeit für die Lastenverteilung ist zwischen allen geteilt [37, Seite 155]. Bei verteilter Lastenverteilung agiert jedes System selbst und holt sich offene Aufgaben und kann mit anderen Systemen interagieren. Verteilte Lastenverteilung ist am effektivsten, wenn die die Systeme möglichst unabhängig voneinander arbeiten können und wenig Interaktion zwischen den Systemen stattfindet [37, Seite 155]. Gute Lastenverteilung wird im dezentralen Ansatz durch die Bemühungen aller Systeme zusammen erreicht durch kooperativen Austausch oder lokaler Optimierungen.

Ein Nachteil bei zentraler Lastenverteilung ist dass das System das die Lastenverteilung ausführt ein Single Point of Failure ist [35, Seite 53]. Bei zentraler Lastenverteilung kann nur ein Task an einen Arbeiter zeitgleich verteilen, was in einem Szenario mit vielen Arbeitern und vielen Tasks zu einem Flaschenhals führen kann [30, Seite 205]. Ein Nachteil bei der verteilten Lastenverteilung ist dass in bei dieser potentiell mehr Nachrichtenaustausch notwendig ist, als bei einer zentralen Variante [37, Seite 155].

Bei der verteilten dynamischen Lastenverteilung können die Systeme kooperativ oder nicht kooperativ agieren. Bei einer kooperativen Variante arbeiten die Systeme auf ein gemeinsames Ziel hinzu wie bspw. die Verkürzung der durchschnittlichen Antwortzeit aller Systeme [37, Seite 155]. Bei einer nicht kooperativen Variante arbeitet jedes System auf die Verbesserung eines lokalen Ziels wie bspw. lokale Antwortzeit hin [37, Seite 155]. Während ein kooperativer Ansatz eine globale Optimierung anstreben kann, fällt sicherlich auch mehr Nachrichtenverkehr durch das Austauschen an Informationen zwischen den Systemen an, was potentielle Performanzgewinne kaputt machen kann. Bei einem nicht kooperativen Ansatz können zwar die Systeme nur lokal ihre Lastenverteilung optimieren, dafür können die einzelnen Systeme potentiell komplett unabhängig von den anderen Systemen handeln.

Bei der zentralisierten dynamischen Lastenverteilung gibt es den “Work-Pool” Ansatz. Bei diesem Ansatz hält der Master die Menge an offenen Tasks. Die Arbeiter fragen den Master nach einem Task, führen diesen aus und wiederholen diese Schritte [30, Seite 204]. Der Work-Pool Ansatz eignet sich für Mengen an Tasks die sich in ihrer Größe unterscheiden oder wenn Tasks mit der Zeit hinzukommen [30, Seite 204]. Dies ist ein sehr interessanter Ansatz im Kontext unserer Arbeit. Der Work-Pool ist die Menge an offenen Tasks die der Master verwaltet. Arbeiter fragen beim Master einen neuen Task an, dann kann

der Master auswählen, welchen Task er dem Arbeiter zuweist. Im einfachsten Fall setzt der Master die Menge durch eine einfache FIFO Warteschlange um. Falls Tasks Prioritätswerte haben, kann der Master anhand dieser die Reihenfolge anpassen in welcher er die Tasks herausgibt.

Besitzen Tasks Prioritäten und kommen Tasks während der Laufzeit hinzu, macht es Sinn ein "preemptive" Lastenverteilungsverfahren zu verwenden. Preemptive, zu deutsch präventiv, "bedeutet, dass eine laufende Ausführung zwangsweise gestoppt wird, um eine Aufgabe mit höherer Priorität zu bedienen" [36, Seite 21 übersetzt mit DeepL].

Eine Variante der dynamischen verteilten Lastenverteilung ist das "Work Stealing" [38]. Bei diesem haben einzelne Systeme eigene Warteschlangen mit potentiell mehreren Tasks die das System abarbeitet. Hat ein System keine Tasks mehr, kann es bei anderen Systemen nach Tasks suchen und diese "stehlen". Dadurch übernehmen weniger ausgelastete Systeme Arbeit von hoch ausgelasteten Systemen und es wird so eine bessere Lastenverteilung erreicht.

Um Algorithmen und Implementierungen von Lastenverteilungsverfahren zu bewerten benötigt man Metriken für diese. Im folgenden werden 9 Metriken vorgestellt entnommen aus eine der Literaturquellen, „A systematic literature review for load balancing and task scheduling techniques in cloud computing," welche sich mit Lastenverteilung speziell im Bereich des Cloud-Computing befasst [36, Seiten 17 bis 21]. Diese sind passend, da Softwarearchitektur und Installation für die Cloud präferiert werden.

- **Throughput,** zu deutsch Durchsatz, bezeichnet die Menge an Arbeit die in einem Zeitraum erledigt wird. Für Problemstellungen bei der ein existierendes System "zu langsam" für die Größe seiner Arbeitslast ist, ist oftmals der Durchsatz des Systems gemeint. So wie in dieser Arbeit muss die Lösung den Durchsatz erhöhen, so dass das System seine Arbeit schnell genug erledigen kann.
- **Makespan,** zu deutsch Produktionsdauer, bezeichnet die benötigte Laufzeit zur Ausführung einer bestimmten Menge an Arbeit. Die Produktionsdauer ist mit dem Durchsatz verbunden, beide Metriken messen die Menge an Arbeit pro Zeit. Für die Produktionsdauer ist es jedoch wichtig, dass für eine bestimmte Menge Arbeit die benötigte Zeit verkürzt wird. Für manche Probleme ist es wichtiger ein gewisses Arbeitspaket möglichst schnell abzuarbeiten, auch wenn das bedeutet über lange Zeit insgesamt weniger Arbeit zu erledigen.
- **Response Time,** zu deutsch Antwortzeit, ist die Zeitspanne zwischen Anfrage an das System durch Nutzer, bis der Nutzer eine Antwort erhält. Diese Metrik ist in Systemen die auf Nutzerinteraktionen reagieren besonders wichtig, da hier eine lange Antwortzeit zu unzufriedenen Nutzern führt.
- **Reliability,** zu deutsch Zuverlässigkeit, bezeichnet die Fähigkeit des Systems Fehler zu erkennen, mit diesen umzugehen, sowie Ausfallzeiten vorzubeugen. Geringe Zuverlässigkeit bedeutet das System hat längere Ausfallzeiten, erfordert mehr Wartung wegen Fehlern. Mehr Ausfallzeiten senken wiederrum andere Metriken wie bspw. den effektiven Durchsatz. Je nach Einsatzgebiet der Anwendung ist die Zuverlässigkeit von hoher Bedeutung, da in Ausfallzeiten abhängige Systeme wahrscheinlich ebenfalls negativ beeinflusst sind.
- **Migration time** bezeichnet die benötigte Dauer um Arbeitslast von einem System auf ein anderes System zu übertragen. Hierzu ist der Kontext jedoch, dass sich die Cloudinfrastruktur ändert und spielt nicht auf Mechanismen die im Work-Stealing wiedergefunden werden an. Sich ändernde Infrastruktur während das System läuft wollen wir im Rahmen dieser Arbeit nicht betrachten, so spielt auch diese Metrik für uns keine Rolle und wird im weiteren nicht betrachtet.

- **Bandwith,** zu deutsch Bandbreite, "stellt die Kapazität oder den verfügbaren Kanal für die Datenkommunikation dar. Sie bezieht sich auch auf die maximale Datenkapazität, die innerhalb eines bestimmten Zeitraums über eine Netzverbindung übertragen werden kann" [36, Seite 20 übersetzt mit DeepL]. Wie zu Beginn von 3.2.2 erwähnt spielt die Netzwerklast in unserem Kontext kaum eine Rolle, somit können wir auch diese Metrik später vernachlässigen.
- **Ressource utilization,** zu deutsch Ressourcenauslastung, "bezieht sich auf die effiziente Zuweisung und Verwaltung von Rechenressourcen, Ressourcen innerhalb einer Cloud-Infrastruktur, um die Anforderungen unterschiedlicher Arbeitslasten zu erfüllen. Dazu gehört die Optimierung der Nutzung von Servern, Speicherplatz, Netzwerkbandbreite und anderen Ressourcen, um die Leistung zu maximieren und die Verschwendung zu minimieren" [36, Seite 20 übersetzt mit DeepL]. Wie zu Beginn von 3.2.2 erwähnt ist für uns die Rechenressource der CPU-Zeit klar zu priorisieren. So ist eine gute Ressourcenauslastung im Sinne der Prozessorauslastung für uns am wichtigsten.
- **Energy consumption,** zu deutsch Energieverbrauch, "kann definiert werden als die Fähigkeit einer Cloud-Infrastruktur, den ihren Stromverbrauch zu optimieren und dabei eine optimale Leistung beizubehalten." [36, Seite 20 übersetzt mit DeepL]. Ein Lastenverteilungsverfahren kann diese Metrik durch Herunterfahren inaktiver Ressourcen beeinflussen, sich verändernde Infrastruktur ist jedoch außerhalb des Scope der Arbeit.
- **Fault tolerance,** zu deutsch Fehlertoleranz, bezeichnet die Fähigkeit des Gesamtsystems ohne Aussetzer weiter fortzuführen trotz aufgetretener Fehler und Teilausfällen. Zuverlässigkeit und Fehlertoleranz haben Überschneidungen, beide Metriken befassen sich mit den Auswirkungen von Fehlern und Teilausfällen auf das System. Ein System mit hoher Fehlertoleranz bleibt selbst unter schlechten Bedingungen noch funktionsfähig.

Für die spätere Auswahl eines Lastenverteilungsverfahren werden wir die Metriken Durchsatz und Ressourcenauslastung priorisieren, jedoch Zuverlässigkeit und Fehlertoleranz nicht außer acht lassen. Die anderen Metriken haben im Kontext der Arbeit weniger Bedeutung und werden auch wegen der Einfachheit nicht weiter betrachtet.

3.2.3 Task Scheduling

Das Task Scheduling befasst sich mit der Fragestellung, wann wird ein Task ausgeführt. Scheduling ist das Ordnen von auszuführenden Teilen der Arbeit auf die verfügbaren Rechenressourcen, in unserem Falle sind diese Teile der Arbeit die Tasks [39, Seite 7]. Ein gutes Scheduling zeichnet sich in parallelen verteilten Systemen über eine gute Lastenverteilung und kurze Ausführungszeit aus [39, Seite 11].

Wir befassen uns mit Schedulingverfahren in parallelen, verteilten Systemen. Dabei hat Scheduling in so einer Umgebung vier Komponenten [39, Seite 8]. Die Zielmaschinen, die parallel ausführbaren Tasks, der generierte Ausführungsplan und Performanzkriterien. Als Zielmaschinen verstehen wir unsere verfügbaren Rechenressourcen, also Prozessorkerne auf verteilten Systemen. Die parallel ausführbaren Tasks sind unsere Tasks selbst, also die Chartanalysen. Der generierte Ausführungsplan ist die Reihenfolge in welcher die Tasks letztendlich auf den Rechenressourcen ausgeführt werden. Dabei ist der Ausführungsplan als kein im Vorhinein festgeschriebenes Artefakt zu verstehen, sondern als Funktion welche einem Task seine Rechenressource und Startzeit angibt [39, Seite 10]. Die Performanzkriterien ähneln den Kriterien der Lastenverteilung, es geht um die Zeit in welcher alle Tasks verarbeitet wurden, die sog. Turnaround-Time. Ist diese Minimal ist die Performanz am größten. [39, Seite 11] [36, Seite 11].

In unserer Arbeit haben wir keine Priorisierung der Tasks durch unsere Fachlichkeit gegeben, ebenso können wir die Tasklänge nicht vorhersagen. Aus diesen Gründen kann unser Scheduling anhand dieser Kriterien keine beste Reihenfolge der Tasks ableiten. Der Fokus für die betrachteten Scheduling-Verfahren ist deshalb auf dessen Mehrwert für die Lastenverteilung. Scheduling-Algorithmen, welche eine möglichst gleiche Verteilung auf Rechenressourcen erreichen, können der Lastenverteilung zugeordnet werden [35, Seite 54].

Wir wollen zunächst mehrere Schedulingansätze welche für uns nicht relevant sind nennen.

- **First Come First Served,** oder auch Warteschlange genannt, führt als nächsten Task den aus, der bereits die längste Zeit auf Ausführung wartet [39, Seite 20].
- **Priority Scheduling** gibt jedem Task einen Prioritätswert. Ein Task mit höherer Priorität wird vor anderen Tasks mit niedrigerer Priorität ausgeführt [39, Seite 21].
- **Shortest Job First,** führt zuerst Tasks mit der geringsten Laufzeit aus um die durchschnittliche Antwortzeit der Tasks zu verringern [39, Seite 21].

Um eine bessere Lastenverteilung zu erreichen betrachten wir nun verschiedene Eigenschaften und Klassifikationen von Schedulingverfahren. Anhand dieser Eigenschaften und der Klassifikation eines Schedulingverfahren wollen wir dieses als potentiell für uns geeignet oder nicht einstufen können.

Ob ein Schedulingverfahren seinen Fokus auf die Lastenverteilung legt oder nicht wird als Eigenschaft eines Schedulingverfahrens angesehen [39, Seite 16]. Bei Schedulingverfahren mit dieser Eigenschaft geht es darum Informationen über die aktuelle Auslastung der Systeme in die Schedulingentscheidungen miteinzubeziehen und damit die Lastenverteilung zu verbessern.

Dabei können Schedulingverfahren mit dieser Eigenschaft nochmals in zwei weitere Kategorien, zentralisiertes und verteiltes Scheduling, aufgeteilt werden. Bei zentralem Scheduling ist ein zentrales System verantwortlich die Informationen über die Auslastung der anderen Systeme zu pflegen und Tasks bei schlechter Lastenverteilung zu übertragen [39, Seite 16]. Bei verteiltem Scheduling, muss jedes System die Informationen über seine Auslastung selbst pflegen und muss andere Systeme nach deren Auslastung fragen können. Genau wie bei zentralem Scheduling muss bei schlechter Lastenverteilung beim verteilten Scheduling die Gemeinschaft der Systeme so kooperieren, dass sie Tasks von hoch ausgelasteten Systemen auf weniger ausgelastete Systeme übertragen werden [39, Seite 16].

So wie bei der Lastenverteilung unterscheidet man zwischen statischen und dynamischen Schedulingverfahren [39, Seite 13]. Auch hier können bei statischem Scheduling nur die Informationen herangezogen werden welche bereits vor dem Start der Ausführung vorhanden sind, so dass ein gutes statisches Scheduling nur möglich ist, wenn das Wissen über die Tasks und Rechenressourcen vorhanden und akkurat ist. Bei dynamischen Schedulingverfahren geht man von einer schlechten Wissenslage über die Ressourcenanforderung eines Tasks und über die Ressourcenauslastung des Systems zur Laufzeit aus. Deshalb werden die Schedulingentscheidungen erst zu Beginn der Ausführung des Tasks ausgeführt [39, Seite 13].

Ein Schedulingverfahren kann entweder einmal vor Ausführung ausgeführt werden, sog. “One-time assignment” oder dynamisch Neuzuweisungen ausführen, sog. “dynamic reassignment” [39, Seite 17+18]. Bei dynamic reassignment können alte Schedulingentscheidungen überholt werden. Bspw. liegen neue Informationen vor, welche zu einem besseren Schedule führen, dann kann mit dynamic reassignment ein Task einer anderen Ressource und oder Zeit eingeteilt werden um ein kürzeres Schedule zu erreichen.

Eine weitere Eigenschaft ist ob das Scheduling "preemptive" oder nicht ist. Ist ein Schedulingverfahren preemtive, dann können Tasks während dessen Verarbeitung pausiert werden und später fortgesetzt werden. Bei non-preemptive Schedulingverfahren lässt sich die Ausführung eines Tasks nicht unterbrechen bevor diese endet [36, Seite 21]. Preemptive-Scheduling findet z.B. Anwendung beim prioritätsbasierten Scheduling. Dabei werden Tasks niedriger Priorität unterbrochen um Tasks höherer Priorität auszuführen. Anschließend kann die Bearbeitung des anderen Tasks wieder fortgeführt werden.

Ein dynamisches Schedulingverfahren kann seine Entscheidungen entweder auf einem einzelnen System treffen oder auf mehrere Systeme verteilen. Das Schedulingverfahren ist dann "distributed" oder "non-distributed" [39, Seite 15]. Diese Klassifizierung ist ähnlich zu der von Scheduling mit Fokus auf Lastenverteilung die wenige Paragraphen zuvor erläutert wurde. Bei Scheduling mit Fokus auf Lastenverteilung spricht man von zentralisiert und verteilt. Scheduling mit Fokus auf Lastenverteilung legt den Fokus bei der Unterscheidung auf das Überwachen der Ressourcenauslastung und Übertragen von Arbeit zwischen Systemen. Bei der Unterscheidung zwischen distributed und non-distributed Scheduling liegt die logische Authorität im Fokus [39, Seite 15]. So wie bei der Lastenverteilung 3.2.2 auch schon erläutert hat ein zentraler Ansatz für Entscheidungsauthorität gegenüber einem verteilten Ansatz Vor- und Nachteile. Ein zentraler Scheduler muss die Arbeitslast des gesamten Schedulings tragen können, gilt als Single Point of Failure, dafür ist bei verteilten Ansätzen potentiell mehr Nachrichtenaustausch notwendig.

Ein verteiltes dynamisches Schedulingverfahren kann "cooperative" oder "non-cooperative" agieren. Im nicht kooperativen Fall, treffen die autonomen Systeme Schedulingentscheidungen für ihre eigenen Ressourcen ohne dabei auf die Auswirkung ihrer Entscheidung auf das restliche System zu berücksichtigen [39, Seite 15]. Im kooperativen Fall muss jedes System zusätzlich zum eigenen Scheduling noch etwas für das Gesamtsystem beitragen. Bei einem nicht kooperativen Ansatz ist der Vorteil, dass die einzelnen Systeme weniger voneinander abhängen und reduziert meist die Komplexität des Scheduling. Da beim nicht kooperativen Scheduling jedes System auf seinem lokalen Bereich optimiert, kann es vorkommen, dass nicht das globale Optimum erreicht wird. Das bedeutet ein kooperatives oder zentralisiertes Scheduling hätte ein besseres Schedule erzeugt.

Die Quelle *Topics in parallel and distributed computing* enthält in Kapitel 11.3 noch weitere Klassifikationen, welche aufgrund fehlender Relevanz hier ausgelassen wurden.

Ein Schedulingverfahren kann auf seine Fairness untersucht werden. Dabei beschreibt die "Unfairness" die maximale Abweichung der tatsächlich ausgeführten Arbeit einer Rechenressource gegen die theoretisch ideale Menge an Arbeit einer Rechenressource im System [40, Seite 310]. Die theoretisch ideale Menge an Arbeit A_{ideal} für jede Rechenressource ergibt sich durch die gesamt verfügbare Arbeit A_{gesamt} und die verfügbaren Rechenressourcen $R_{\text{verfügbar}}$.

$$A_{\text{ideal}} = \frac{A_{\text{gesamt}}}{R_{\text{verfügbar}}} \tag{3.1}$$

Der Fairnesswert ergibt sich dann durch die errechnete Unfairness, ein kleinerer Wert bedeutet höhere Fairness. Erreicht man eine hohe Fairness, erreicht man auch eine hohe durchschschnittliche Ressourcenauslastung, eine vorteilhafte Eigenschaft für die Performanz eines Systems. Dabei ist zu beachten, dass die Rückrichtung nicht stimmt. Es kann eine hohe durchschschnittliche Ressourcenauslastung im System erreicht werden trotz hoher Unfairness, weil der Durchschnitt nicht beeinflusst wird, wenn eine bestimmte Rechenressource mehr Arbeit verrichtet als andere.

Eine Technik die ein Schedulingverfahren umsetzen kann ist das Prinzip des "Work-Stealing". Work-

Stealing beschreibt, dass Rechenressourcen welche nicht mehr ausgelastet sind, sich Arbeit von Rechenressourcen die ausgelastet sind "klauen", also übernehmen [41, Seite 720]. Dadurch kann eine höhere Ressourcenauslastung auf allen Rechenressourcen erreicht werden und damit ein besseres Scheduling.

Ein Schedulingverfahren kann "greedy" sein. Ein Greedy-Scheduler weißt einem Prozessorkern sobald der Prozessor frei wird und eine Aufgabe verfügbar ist eine Aufgabe zu. Dabei liegt die erreichte Laufzeit innerhalb des Faktors zwei der optimalen Laufzeit [42, Seite 1].

In manchen Fällen kann es sinnvoll sein anstatt jeden Task einzeln zu schedulen, mehrere Tasks zu einem Paket zu schnüren und dies als einzelnes Element zu schedulen, dem sog. Batch-Scheduling. Dieses kann sinnvoll sein, wenn bei einem zentralen Ansatz die zentrale Komponente einen Flaschenhals darstellt. Hat man kleine Mengen an Tasks, kann Batch-Scheduling aber auch zu schlechterer Ressourcenauslastung und Durchsatz führen, wenn eine Rechenressource mehrere Tasks zur Verarbeitung hat und eine andere Rechenressource keine. Je nach Situation kann man diesen negativen Nebeneffekt wieder durch das Anwenden von Work-Stealing ausgleichen, dann sollte man jedoch abwägen ob das Batch-Scheduling immernoch einen Mehrwert bringt.

3.3 Wie koordinieren sich verteilte Systeme?

Diese Sektion wird sich mit Transaktionen und Sperrbasierter Synchronisation von relationalen Datenbanken beschäftigen, wird jedoch dieses Wissensgebiet nicht vollständig durchleuchten oder erklären. Im Kontext unserer Arbeit benötigen wir einen Mechanismus, so dass bei zwei konkurrierenden Schreibanfragen auf einen Datensatz, eine Anfrage gewinnt und die andere fehlschlägt.

Verteilte Systeme laufen unabhängig voneinander, trotzdem greifen diese oftmals auf geteilte Ressourcen zu. Wir betrachten hier den Zugriff auf einen gemeinsamen Datenbestand mittels einer relationalen Datenbank. Passiert der Zugriff auf Daten zeitlich nacheinander getrennt entsteht kein Problem. Da die verteilten Systeme aber unabhängig voneinander agieren ist dies nicht garantiert. Greifen zwei Systeme zeitgleich auf die gleichen Daten zu und modifizieren diese, dann kann es zu einem inkonsistenten Zustand in den Daten kommen. Die verteilten Systeme müssen sich also in einer Weise koordinieren um einen korrekten Umgang mit dem Datenbestand zu versichern.

Zur Anschauung mit dem Beispiel einer Bankapplikation. Ein System möchte eine Überweisung von Konto A auf Konto B tätigen, zeitgleich möchte ein anderes System eine Einzahlung auf Konto A tätigen. Beide dieser Tätigkeiten beinhalten Schritt eins, Lesen des Kontostands von A, Schritt zwei, Modifizieren des Kontostands und Schritt drei, Speichern des neuen Kontostands. Beide Systeme führen jeweils Schritte eins bis drei nacheinander aus. Jenachdem in welcher Reihenfolge die Schritte der zwei Systeme zeitlich ausgeführt werden ergeben sich verschiedene Probleme [43]. Diese Probleme, wenn man sie unbeachtet lässt führen zu einem sog. inkonsistenten Zustand, einem unlogischen Endergebnis. In unserem Beispiel könnte z.B. der Fall auftreten, dass Konto A nicht von der Überweisung an Konto B belastet wird, also das Geld nicht abgezogen wird. Diese Probleme nennt man Probleme des konkurrierenden Zugriffs [44] oder auch Mehrbenutzerkonflikte [45, Seite 337].

Datenbanken bieten uns gegen diese Probleme Sperrverfahren oder auch Sperrbasierte Synchronisation [45]. "Sperrverfahren in Datenbanken folgen dem ACID-Prinzip (atomicity, consistency, isolation, durability), wonach eine Transaktion unteilbar und von anderen Transaktionen isoliert ist, nach Abschluss der Transaktion die Daten in einem konsistenten Zustand sind und die Änderungen dauerhaft bestehen." [43].

Ein Klient eröffnet implizit oder explizit eine Transaktion, wenn er auf Daten der Datenbank zugreifen möchte. Eine Sperre gehört immer einer Transaktion und bezieht sich auf einen Teil der Datenbank, wie bspw. ein einzelner Datensatz. Es gibt zwei Arten von Sperren "shared, read lock" und "exclusive,

write lock" [45, Seite 324]. Ist ein Datensatz mit einem read lock belegt, können andere Transaktionen ebenfalls ein read lock anfordern, so können beide Transaktionen den Wert des Datensatzes auslesen. Ist ein Datensatz mit einem read lock belegt, kann keine weitere Transaktion ein write lock auf den Datensatz anfordern. Ist ein Datensatz mit einem write lock belegt, kann keine andere Transaktion ein read lock oder ein write lock auf den Datensatz anfordern [45, Seite 324]. Diese "Verträglichkeit von Sperranforderungen" [45, Seite 324] verhindern die Mehrbenutzerkonflikte.

Sperrbasierte Synchronisation kann pessimistisch oder optimistisch erfolgen [45, Seite 337].

"Synchronisationsmethoden werden als pessimistische Verfahren bezeichnet, [wenn] sie von der Prämisse ausgehen, dass Mehrbenutzerkonflikte auftreten werden" [45, Seite 337]. Bei der pessimisten Synchronisation muss eine Transaktion, bevor es Datensätze ließt oder modifiziert, Sperren auf die Datensätze anfordern. Nur wenn die Transaktion eine Sperre auf den Datensatz besitzt, darf die Transaktion den Datensatz verändern.

"Bei der optimistischen Synchronisation geht man davon aus, dass Konflikte selten auftreten und man Transaktionen einfach mal ausführen sollte und im Nachhinein (à posteriori) entscheidet, ob ein Mehrbenutzerkonflikt aufgetreten ist oder nicht" [45, Seite 337]. Optimistische Synchronisation vermeidet Overhead im Vorhinein, muss jedoch am Schluss validieren ob eine Kollision aufgetreten ist. Tritt eine Kollision auf muss die getane Arbeit komplett rückgängig gemacht werden.

Die Verwendung einer Datenbank, welche Transaktionen und Sperren verwaltet ist zentralisierte Sperrung für unsere verteilten Systeme [46]. Alternativ zu diesem zentralen Ansatz gibt es auch verteilte Ansätze.

Die konsensbasierte Sperrung, diese hat einen Algorithmus, wie Paxos oder Raft, um einen Anführer in den verteilten Systemen auszuwählen welcher dann die Aufgabe der Sperrsynchronisation übernimmt [46]. Dieser Ansatz unterscheidet sich nicht so stark vom zentralisierten Ansatz, nur dass kein dediziertes System mehr verantwortlich ist, sondern einer der Teilnehmer. Der Vorteil bei diesem Ansatz ist, bei Ausfall des Anführersystems wird einfach ein neuer Anführer ausgewählt. In der zentralen Alternative würde das System beim Ausfall nicht weiter funktionieren und blockiert werden.

Die dezentrale Sperrung ist ein Ansatz welcher ohne zentrale Authorität sperrbasierte Synchronisation erreicht, z.B. über das Gossip-Protokoll [46]. Bei der dezentralen Sperrung gibt es keine zentrale Authorität, stattdessen muss sich ein Teilnehmer, wenn er eine Ressource sperren möchte davon überzeugen, dass kein anderer Teilnehmer damit in Konflikt steht. Vorteil dieses Ansatz ist eine bessere Skalierbarkeit, da die gesamte Arbeitslast der Sperrung nicht mehr von einem System getragen werden muss.

3.4 Fehlertoleranz & Zuverlässigkeit

In dieser Sektion wollen wir wenige Begriffe aus den Bereichen Fehlertoleranz und Zuverlässigkeit erläutern, die wir später in unseren Lösungsalternativen verwenden.

Unsere erste Frage die uns in diesem großen Themenbereich beschäftigt ist, wie erkennt man bei verteilten Systemen den Ausfall eines Systems. In dieser Sektion wollen wir betrachten wie man in verteilten Systemen Ausfälle einzelner oder mehrerer Systeme erkennen kann. Im Kontext der Arbeit ist es wichtig, dass trotz Ausfall von einem oder mehrerer Systeme, Neustarts etc. trotzdem jeder Task letztendlich ausgeführt wird.

Um den Ausfall eines Systems zu erkennen, kann man einen Heartbeat verwenden [47]. Der Name und der Mechanismus ziehen hier Analogie zu dem Herzschlag eines Tieres. Das schlagen des Herzes bestätigt immer wieder, dass das Tier noch lebt. Bleibt der Herzschlag des Tieres zu lange aus, erklärt man es für tot. Das beobachtete System sendet periodisch einen Herzschlag, eine beliebige Nachricht an den Beobachter, dadurch weiß dieser dass das beobachtete System noch aktiv ist. Bleibt dieser Herzschlag länger als ein gewisser Schwellwert aus, ein Timeout, dann geht der Beobachter davon aus, dass das beobachtete System ausgefallen ist.

Wir wollen kurz im Bereich der Fehlertoleranz den Begriff Single Point of Failure erklären. Ein Single Point of Failure ist ein Teilsystem dessen Ausfall das gesamte restliche System ebenfalls zum stoppen bringen würde. Ist ein Teilsystem ein Single Point of Failure ist das meist ein Hinweis daraufhin, dass man die Aufgaben dieses Teils redundant ausführen sollte. Ein Single Point of Failure birgt ein großes Risiko, da sich ein Fehler auf diesen einzelnen Punkt auf das gesamte System auswirkt.

3.5 Hinleitung zu den Lösungsalternativen

In diesem Teil werden zuerst aus der Problembeschreibung, Ziel und den Rahmenbedingungen einige Vorüberlegungen und Fragen gestellt. Die Lösungsalternativen müssen diese Vorüberlegungen und Fragen beachten und beantworten. Darauf folgt ein kurzer Teil, in welchem wir den Lösungsalternativen vorwegnehmen, dass unser Problem durch Parallelisieren der Tasks gelöst wird und Sperrbasierte Synchronisation mit zentralem Konsens stattfindet. Nach dieser Sektion folgen drei Kapitel für drei betrachtete Lösungsalternativen. Diese Kapitel stellen Entwürfe einer Lösung vor und orientieren sich an den Vorüberlegungen und gestellten Fragen und weißen eine ähnliche Struktur auf.

3.5.1 Vorüberlegungen

Wir nennen eine Chartanalyse folgend Task. Ein Arbeiterthread oder Arbeiter bezeichnet einen Thread einer Applikation welcher, unter anderem, Tasks ausführt.

Es folgen eine Reihe von Aussagen, Folgerungen aus Problem, Ziel und Rahmenbedingungen.

- Ein Task lässt sich nicht weiter parallelisieren.
- Tasks sind unabhängig voneinander.
- Die Laufzeit eines einzelnen Task ist vor dessen Abschluss unbekannt.
- In einem Google Drive Ordner sind alle Eingabedateien, dabei ist eine Datei ein Task.
- Eine Eingabedatei dessen Chartanalyse fertig ausgeführt ist liegt nicht mehr im Eingabeordner.
- Jederzeit können Eingabedateien in den Eingabeordner hinzugefügt oder entfernt werden.
- Die Threads auf einem Computer haben geteilten Hauptspeicher, aber Threads verschiedener Systeme nicht.
- Wir wollen gute horizontale Skalierung erreichen, denn mehr kleine Prozessoren sind günstiger als einzelne viel schnellere Prozessoren zu kaufen.
- Das System wird immer ungefähr gleich starke Prozessorkerne beinhalten, so dass die Laufzeit eines Tasks auf allen Threads gleich schnell ist.

- Threads und Server können ungeplant stoppen.
- Ein abgebrochener Task soll neu bearbeitet werden.
- Zweifache Ausführung eines Task soll vermieden werden.
- Ressourcen sollen sinnvoll ausgenutzt werden.
- Die Installation sollte wenig Wartung beanspruchen und wenig Komponenten umfassen.
- Unser Backend soll ein verteiltes System mit mehreren Applikationsinstanzen und mehreren Prozessorkernen sein.

Wir können aus den einzelnen Vorüberlegungen Fragen, Problematiken, formulieren welche zusammenhängende Überlegungen zusammen behandeln. Die Lösungsalternativen lösen diese Fragen:

1. Was für Komponenten und Infrastruktur werden benötigt?

2. Wie erkennen wir neue Tasks im System?

3. Wann wird welcher Task bearbeitet?

4. Wo wird ein Task bearbeitet?

5. Wie verhindern wir doppelte Ausführungen?

6. Wie kann das System skaliert werden?

7. Wie werden ungeplante Ausfälle behandelt?

3.5.2 Beschleunigung durch parallele Ausführung von Tasks

Die Lösungsalternativen erreichen die Beschleunigung des Systems alle über die Parallelisierung von Tasks. Das bedeutet mehrere Tasks werden in mehreren Threads auf mehreren Prozessorkernen auf mehreren Systemen gleichzeitig ausgeführt. Das nennt man Multiprocessing in verteilten Systemen.

Die Lösungen betrachten nur Ansätze in welchen wir pro Prozessorkern einen Arbeiterthread haben. Ein Ansatz mit mehr Threads als Prozessorkernen ist ebenfalls interessant, in welchem ein Prozessorkern mehrere Tasks via Multitasking nebenläufig abarbeitet. Betrachtet man ein Beispielszenario, in welchem mehrere Tasks auf einem Prozessorkern ausgeführt werden, dann hätten kürzere Tasks eine geringere Latenz gegenüber einer zufälligen Reihenfolge. Je nach Situation könnte so eine Eigenschaft vorteilhaft sein, wir betrachten dies aber nicht weiter. Aufgrund des höheren technischen Overheads muss man so einen Ansatz vorsichtig abwägen [48, Seite 73].

Manche Arten von Chartanalysen, wie bspw. die Parameteroptimierung, ließen sich anhand des Fork-Join Modells(3.1.2) nochmals unterteilen und Parallelisieren. Diese Möglichkeit betrachten wir jedoch im weiteren nicht, sondern gehen im Ausblickskapitel(9.2) nochmal kurz darauf ein.

Unser Ansatz, paralleles Ausführen einzelner Tasks, macht einen großen Teil des Programms parallel. Dadurch erreichen wir nach Ahmdahl's Law eine hohe Beschleunigung gegenüber der sequentiellen

Ausführung.

Unser Parallelisierungsproblem ist auf Grund der fehlenden Abhängigkeiten und Zusammenhänge der einzelnen Tasks ein embarrassingly parallel Problem. Das bedeutet die parallelen Threads profitieren nicht durch gemeinsamen Hauptspeicher. Das erlaubt uns einzelne Arbeiterthreads auch als einzelne Systeme zu betrachten, wir sehen keinen Unterschied zwischen zwei Arbeiterthreads auf einem physischen System oder auf zwei verschiedenen Systemen. Da unsere Tasks voneinander unabhängig sind, kann die Ausführung auch isoliert stattfinden und die Threads müssen während der Task-Ausführung nicht miteinander kommunizieren.

Die Lösungsalternativen erreichen die Beschleunigung der Taskausführungen also durch Ausnutzen der parallelen Rechenressourcen auf den verteilten Systemen. Dazu muss die Gesamtlösung mit Lastenverteilung und Scheduling eine gute Verteilung der Tasks auf die Rechenressourcen und Zeit erreichen.

3.5.3 Ausschluss doppelter Ausführung über zentralisierte Sperrbasierte Synchronisation

Alle Lösungsalternativen benötigen Prozesssynchronisierung, denn es muss verhindert werden, dass ein Task mehrfach ausgeführt wird. Wird ein Task ausgeführt, dann bieten weitere Ausführungen desselben Tasks keinen Mehrwert und verschwenden Rechenzeit. Verschwendete Rechenzeit bedeutet wiederum, dass man die gleiche Task-Menge mit weniger Rechenkapazität erreichen könnte. Das widerspricht der Anforderung an eine infrastrukturminimale Lösung.

Ein Task hat für uns vier Zustände. Offen, bedeutet der Task muss ausgeführt werden. In Bearbeitung, ein Task der gerade von einem Thread bearbeitet wird. Geschlossen, ein Task der erfolgreich fertig bearbeitet wurde. Fehlgeschlagen, wenn während der Taskausführung ein Fehler aufgetreten ist. In folgendem Zustandsdiagramm (Abb. 3.1) sind diese ebenfalls nochmal abgebildet.

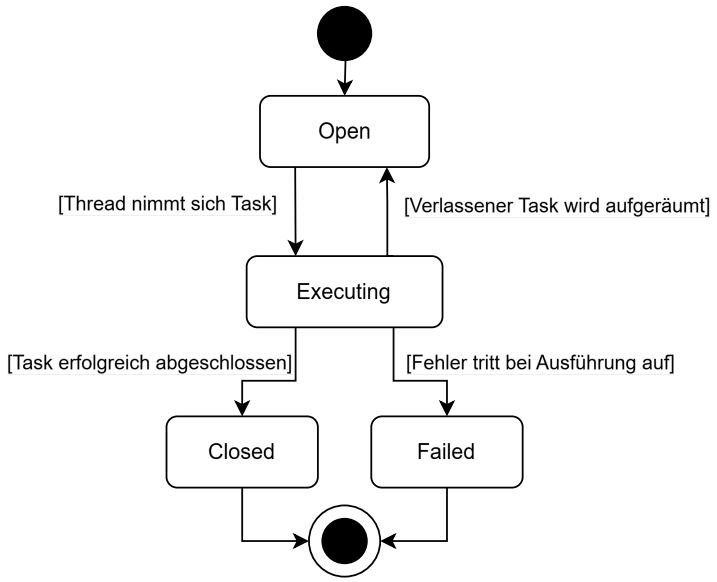

Abbildung 3.1: Zustandsdiagramm für Tasks

Das gesamte System muss sicherstellen, dass ein Arbeiter nur dann beginnt einen Task zu bearbeiten,

wenn kein anderer Arbeiter den Task bereits ausführt oder ausgeführt hat. Das bedeutet der Task muss offen sein. So ist sichergestellt, dass ein Task nur einmal ausgeführt wird.

Doppelte Ausführungen werden ausgeschlossen, wenn das Überführen des Task-Zustand von offen zu in Bearbeitung für das gesamte System synchron passiert. Diesen Zustandsübergang definieren wir als kritischen Bereich. Im kritischen Bereich befindet sich immer nur maximal ein Teilnehmer, so kann nur ein Teilnehmer den Task-Zustand von offen zu in Bearbeitung überführen. Diesem Teilnehmer gehört damit der Task und er darf ihn ausführen. Das Betreten und Verlassen des kritischen Bereichs muss synchronisiert über alle Systeme passieren. Betreten bedeutet das Sperren eines oder mehrerer Tasks, Verlassen bedeutet Freigeben der Sperre. Die Lösungsalternative in Kapitel sechs hat Sperren über einzelne Tasks, die Lösungsalternativen aus Kapitel vier und fünf sperren die Menge der offenen Tasks. Dabei erreicht das Sperren der Menge der offenen Tasks ebenso den Ausschluss von Doppelausführungen.

Die drei vorgestellten Lösungsalternativen verwenden zentralisierte sperrbasierte Synchronisation(3.3). Beim ersten Ansatz aus Kapitel vier, durch eine ActiveMQ Instanz und bei den anderen beiden Lösungsalternativen in Kapitel fünf und sechs durch eine relationale Datenbank. Diese zentrale Komponente bietet allen Teilnehmern die Sperrlogik, zum Sperren von Tasks.

Alternativ zur zentralisierten Sperrung sind auch verteilte Ansätze denkbar. Bei den verteilten Ansätzen gibt es keine zentrale Instanz, die Teilnehmer müssen sich untereinander synchronisieren. Der gewünschte Vorteil einer verteilten Sperrlogik ist das Reduzieren der Infrastrukturkomponenten, durch Weglassen der Datenbank oder ActiveMQ. Ebenso stellen Datenbank und ActiveMQ jeweils Single points of failure dar.

Ein Konsensbasierter Ansatz in welchem eine Applikationsinstanz die Synchronisierung handhabt scheint auf Grund der Ähnlichkeit zum zentralisierten Ansatz mit einer relationalen Datenbank interessant zu sein. Diese Alternative wurde nicht ausgearbeitet, so sind noch mehrere Fragen offen.

- Was ist wenn die Leader-Applikationsinstanz unerwartet ausfällt?
- Brauchen wir eine Form von Persistenz?

Eine dezentrale Sperrung, in welcher es keine zentrale Authorität gibt ist auch denkbar. Der Vorteil hier ist, dass die Rechenlast nochmals besser verteilt wird und nicht ein System den gesamten Overhead übernehmen muss, wie es beim konsensbasierten Ansatz der Fall wäre.

Aus zeitlichen Gründen wurde jedoch kein Algorithmus für verteilte sperrbasierte Synchronisation weit genug untersucht um eine sinnvolle Implementierung umzusetzen.

Ein weiterer Vorteil verteilter Ansätze ist, dass Algorithmen für diese meist direkt Mechanismen beinhalten um Ausfälle korrekt zu behandeln. Ein Sorge ist jedoch, dass der Implementierungsaufwand potentiell höher ist. Es wurde nicht untersucht, ob es einfach zu integrierende, bereits implementierte, Algorithmen gibt.

Ein weiterer Nachteil vieler der Algorithmen ist, dass die Systeme mehr Nachrichten austauschen müssen. Bei Algorithmen welche eine der verteilten Systeme als Koordinator wählen, fallen Nachrichten zum Auswählen dieses Koordinators an. Bei anderen Algorithmen müssen Teilnehmer mit mehr als einem anderen Teilnehmer, im schlechtesten Fall mit allen anderen Teilnehmer Nachrichten zum Betreten und Verlassen des kritischen Bereichs austauschen. Durch das Ausfallen oder Hinzukommen eines Teilnehmers in das System fallen ebenfalls Nachrichten an. Die hohe Anzahl an Nachrichten ist für Systeme mit hochfrequenter Nutzung des kritischen Bereichs und einer hohen Teilnehmeranzahl meist ein großes Performanzproblem. Bei unserem System ist die Frequenz für einen einzelnen Arbeiter den kritischen Bereich zu benutzen auf Grund der langen Taskausführung recht gering. Es ist also durchaus denkbar, dass dieses Performanzproblem in unserem Fall nicht auftritt. Es müsste genau untersucht werden ob das Entfernen der Infrastrukturkomponente den Overhead der erhöhten Nachrichtenzahl es wert ist.

Keine der Lösungsalternativen wählt einen Ansatz der verteilten sperrbasierten Synchronisation, weil die Lösungsalternativen in Kapitel fünf und sechs, eine relationale Datenbank verwenden, welche bereits als Gegegeben gilt.

Kapitel 4

Work-Pool Lastenverteilung mit zustandslosen Arbeitern

In diesem Kapitel wird eine Alternative vorgestellt, welche die Verwaltung von Tasks zentralisiert und einem zentralisierten dynamischen Work-Pool Lastenverteilungsansatz ähnelt. Lediglich die Ausführung der Tasks ist in Arbeiterthreads auslegt. Durch diesen zentralisierten Ansatz unterscheidet sich die Alternative von den anderen beiden vorgestellten Ansätzen. Diese Alternative birgt im Kontext der Arbeit Nachteile wegen ihrer Infrastrukturanforderungen und scheidet daher für unsere Umsetzung aus. Für diese Alternative muss eine ActiveMQ Instanz eingerichtet und verwaltet werden, bei den zwei anderen Lösungsalternativen können wir eine bestehende relationale Datenbank nutzen. Lässt man diese spezielle Rahmenbedingung der bereits vorhandenen Datenbank außen vor dann ist diese Alternative ein solider Entwurf um zu einer guten Lösung zu führen. Im kommenden Bewertungskapitel 7 wird dies auch nochmal etwas ausgeführt.

Es folgt eine illustrative Abbildung 4.1 welche die Lösungsalternative und die Interaktionen zwischen den Komponenten verbildlicht. Diese Komponenten und Interaktionen werden in den kommenden Sektionen detailliert erläutert und die Abbildung dient zur Orientierung so wie ein kompakten Überblick über diese Lösungsalternative. Dabei sind die Ovale die einzelnen Systeme / Komponenten der Lösungsalternative. Die Pfeile sind logische Aufrufe an andere Systeme.

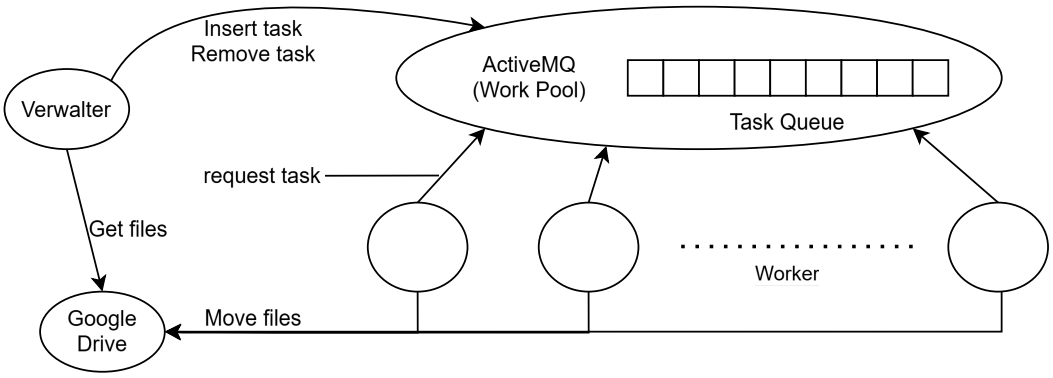

Abbildung 4.1: Work-Pool LV mit ActiveMQ

4.1 Komponenten und Infrastruktur

Die Lösungsalternative hat eine Applikation welche neue Tasks erkennt und diese in eine Warteschlange von ActiveMQ einträgt. Von dieser Applikation haben wir exakt eine Instanz, genannt den Verwalter. Es gibt exakt eine ActiveMQ Instanz, mit einer Warteschlange um die Tasks abzulegen.

Das ActiveMQ ist zustandsbehaftet, denn es persisitiert seinen Zustand und Logs in lokalen Dateien. Optional, durch hinzunehmen einer relationalen Datenbank kann ActiveMQ so konfiguriert werden, dass es seine Daten in der Datenbank persistiert. Dadurch würde die ActiveMQ Instanz ebenfalls zustandslos sein.

Die Lösungsalternative hat eine Applikation um Tasks auszuführen, die Arbeiterapplikation. Ein Arbeiterthread dieser Applikationsinstanzen holt sich aus der ActiveMQ Warteschlange einen Task, führt diesen aus und wiederholt das Ganze. Das System hat mindestens einen Arbeiterthread in mindestens einer Arbeiterapplikationsinstanz.

4.2 Taskerkennung

Der Verwalter verwendet Polling um neue Tasks zu erkennen und folglich der ActiveMQ Warteschlange hinzuzufügen. Der Verwalter ruft zuerst die ActiveMQ Warteschlange auf um die FileIds der derzeitigen Eingabedateien in der Warteschlange zu erhalten. Als zweiten Schritt ruft er den Eingabeordner im Google Drive auf, um die FileIds der derzeitigen Eingabedateien im Google Drive Ordner zu erhalten. Der Verwalter filtert aus den FileIds des Google Drive Ordner die FileIds der Warteschlange heraus und erhält damit die Menge an FileIds neuer Tasks. Diese trägt er in die ActiveMQ Warteschlange ein. Um vom Nutzer entfernte Eingabedateien zu erkennen filtern wir aus den FileIds der Warteschlange die FileIds des Google Drive Ordners heraus und löschen die Einträge in der ActiveMQ Warteschlange. In der Implementierung muss darauf geachtet werden, dass ActiveMQ und Google Drive in konsistentem Zustand bleiben. Nimmt sich ein Arbeiterthread einen Task, dann muss dieser aus dem Google Drive Eingabeordner verschoben sein, bevor der Warteschlangeneintrag von ActiveMQ verschwindet. Ansonsten gibt es dass Szenario in welchem der Verwalter zwischen diesen Events den Task erneut der Warteschlange hinzufügt.

Ein eventbasierter Ansatz wäre mit Google Cloud Webhooks(2.2.3) umsetzbar. Der Webhook würde sich an das Event, dass eine Datei in den Eingabeordner verschoben wurde anhängen, also das Hinzufügen eines neuen Tasks in unserem Kontext. Der Webhook würde dann unseren Verwalter mit dem Event aufrufen. Vorteil eines Webhooks wäre eine potentielle Reduktion der Latenz zwischen Zeitpunkt zu dem ein neuer Task hinzugefügt wurde und Zeitpunkt zu dem unser System den neuen Task erkennt und vermeidet unnötige Pollinganfragen. Die extra Latenz durch den Polling Ansatz ist relativ zur Dauer eines Tasks vernachlässigbar. Die Google Cloud Webhooks müssten zusätzlich eingerichtet und verwaltet werden, was einen Mehraufwand darstellt und deswegen nicht bevorzugt wird. Ein Aspekt der aus zeitlichen Gründen nicht weiter verfolgt wurde ist wie sich die Google Cloud Webhooks verhalten, wenn ein neuer Task hinzugefügt wird, während der Verwalter offline ist. Da bei einem Webhook-Ansatz nur dann Nachrichten zwischen unserer Anwendung und der Google API ausgetauscht werden, wenn ein neuer Task verfügbar ist und nicht in kurzen zeitlichen Abständen, spart uns das potentiell viele API-Anfragen an die Google API ein. Die Zahl der API-Anfragen ist dann relevant, wenn wir die Nutzungsbeschränkungen der Google API nicht mehr einhalten können [49].

Noch besser statt den Webhook den Verwalter aufrufen zu lassen, das Eingabeevent in eine Google Cloud Function auslagern, welche den neuen Task direkt in die ActiveMQ Warteschlange macht. Das würde den Verwalter überflüssig machen.

4.3 Verteilung von Tasks auf Ressourcen und Zeit

Tasks sind Google Docs Dateien mit Inhalt im YAML-Format. Der Inhalt dieser Datei ist erst zur Ausführung des Tasks wichtig. Unser System betrachtet dabei nur Metadaten, wie FileId und Speicherort. Zu Anfang ist eine Eingabedatei im Eingabeordner in Google Drive. Analog zu den vier Task-Zuständen(3.1) schieben wir die Eingabedatei vom Eingabeordner in den Executing-Ordner und danach in den Closed oder Failed-Ordner. Das Verschieben aus dem Eingabeordner ist ein logisch wichtiger Schritt, da er dadurch aus der Menge der offenen Tasks genommen wird. Die restlichen Ordner dienen zur Übersichtlichkeit für den Nutzer und im Falle des Failed-Ordners für den Maintainer. Google Drive ist ohne Mühen von uns persistent und sehr zuverlässig und fehlertolerant.

Der Verwalter fügt die Tasks des Eingabeordners in die ActiveMQ Warteschlange hinzu. Die Warteschlange wird dabei mit dem Standardverhalten, first in first out, betrieben. Da wir derzeit keine Priorisierung für Tasks definiert haben, ist das ausreichend. Eine Priorisierung der Tasks wäre relativ leicht mit der ActiveMQ Warteschlange in Zukunft umsetzbar, da das Warteschlangenverhalten konfigurierbar ist.

Der Verwalter und ActiveMQ setzen einen Work-Pool mit dynamischer Taskmenge um, wir entkoppeln die Eingabeevents und dessen Verarbeitung. Der Work-Pool Ansatz eignet sich für Problemstellungen wie unsere gut, in welchen relativ wenige Arbeiter rechenintensive Arbeiten erledigen [30, Seite 204+205]. Kommen weitere Eingaben dazu während das System voll ausgelastet ist, werden diese einfach in die Warteschlange hinzugefügt und gehen nicht verloren.

Jeder Arbeiterthread fragt ActiveMQ aktiv nach einem neuen Task nachdem er den letzten Task fertig bearbeitet hat. Ein simpler, aber sehr effektiver Ansatz für die Lastenverteilung und Scheduling.

Wir wollen die Vorteile dieses Ansatzes etwas beleuchten indem wir andere Möglichkeiten gegenüber stellen.

Durch die unbekannte Ausführungsdauer der Tasks scheiden statische Lastenverteilungsansätze, welche aktuelle Ressourcenauslastung nicht in Betracht ziehen, aus. Gegenüber unserem Work-Pool Ansatz, wo Arbeiter sich ihre Tasks abholen, gibt es auch den Ansatz in welchem eine zentrale Anwendung, die Tasks an die Arbeiterthreads aktiv verteilt. Das nennt man dann einen Dispatcher, die Rolle würde dann der Verwalter übernehmen. Da die Ausführungsdauer der Tasks im Vorraus unbekannt ist, kann der Dispatcher aber nicht proaktiv vorplanen. Der Dispatcher kann also nur, wie folgt, reaktiv agieren. Entweder wenn er sieht dass ein Arbeiterthread nichts tut und gibt ihm einen Task. Oder der Arbeiterthread benachrichtigt den Dispatcher dass er keinen Task mehr hat und bekommt darauf hin einen Task. Der Work-Pool Ansatz ist einfacher in der Implementierung und schneller, da zwischen dem Verwalter und den Arbeiterthreads weniger Nachrichten ausgetauscht werden müssen. Ein Ansatz mit aktivem Dispatcher könnte dann trotzdem Sinn machen, wenn sich die Arbeiterthreads in ihrer Geschwindigkeit unterscheiden. Dann könnte man beim Verteilen der Tasks die schnelleren Threads bevorzugen um die Performanz des Systems zu steigern.

Die Lösung setzt ein greedy Scheduling und faires Scheduling um. Greedy, weil jeder Thread sich ohne Beachtung der anderen Threads einen neuen Task holt. Da jeder Thread die gleichen Chancen beim Holen eines neuen Tasks hat, ist es ebenfalls fair. Daraus folgt sehr gute Ressourcenauslastung der Arbeiterthreads, was wichtig für eine infrastrukturminimale Lösung ist. Da ein Thread seine eigene Ressourcenauslastung als Entscheidungsgrundlage für das schedulen eines neuen Tasks zu sich, hat unser Schedulingansatz, wenig überraschend, die Lastenverteilungseigenschaft.

Da während der Laufzeit des Systems immer weitere Tasks hinzukommen ist der Scheduling Ansatz dynamisch.

Das Scheduling ist ebenso non preemptive, denn solange Tasks keine Prioritäten haben, muss man

keinen Task abbrechen um einen wichtigeren vorzuschieben.

Wegen fehlender Priorisierung, homogener Prozessoren und unbekannter Tasklänge brauchen wir keine dymanic-reassignments. Wir können die Auswahl des nächsten Tasks für eine bestimmte Rechenressource nicht besser als zufällig wählen.

4.4 Verhindern von doppelten Ausführungen

Die Lösung braucht ebenso eine Form von Synchronisation um konkurrierende Arbeiterthreads korrekt abzuhandeln. Wenn zwei Arbeiterthreads quasi zur gleichen Zeit bei ActiveMQ nach einem neuen Task fragen, dann stehen diese Threads in Konkurrenz zueinander. Das ActiveMQ arbeitet diese Anfragen serialisierbar ab, es können also keine zwei Arbeiterthreads denselben Task bekommen. Die Synchronisation nimmt uns hier ActiveMQ komplett ab. Jedoch, wie im Teil zur Taskerkennung(4.2) angemerkt, muss die Transaktion, Warteschlangeneintrag abholen, so lange andauern, bis die Datei des Tasks nicht mehr im Google Drive Eingabeordner ist. Umsetzbar sollte dies durch Verwendung von Client-Acknowledgements von ActiveMQ sein, bei der ein Klient den korrekten Erhalt der Nachricht manuell bestätigt [50]. ActiveMQ markiert sich bei diesem Acknowledgement Modus die Nachricht als geliefert und unbestätigt und entfernt den Eintrag in der Warteschlange erst nach Erhalt des Acknowledgements. Da die Bestätigung manuell vom Klient ausgeführt wird, kann der Klient in dieser Zeit die Eingabedatei im Google Drive verschieben und anschließend das Acknowledgement an ActiveMQ senden. Solange wir nur einen Verwalter einsetzen ist doppeltes Hinzufügen neuer Tasks trivialerweise nicht möglich.

4.5 Skalieren des Systems

Der Verwalter und ActiveMQ muss wenn notwendig vertikal hochskaliert werden, da die Lösung keine verteilte Lösung derer Aufgaben vorsieht. Arbeiterthreads können quasi beliebig horizontal skaliert werden.

4.6 Fehlertoleranz

Wir nutzen die ActiveMQ Warteschlange mit dem Client-Acknowledgement Feature. Dieses führt dazu, dass ActiveMQ nach Herausgabe eines Tasks auf ein Acknowledgement des Client, also des Arbeiterthreads, wartet. Das Warten ist nicht blockierend für ActiveMQ. In Erweiterung zum Verhindern von Doppelausführungen in der Sektion vor dieser, senden wir das Acknowledgement nicht nachdem die Eingabedatei aus den Google Drive Eingabeordner verschoben wurde, sondern nachdem der Arbeiter den Task fertig bearbeitet hat. Die ActiveMQ Warteschlange hat dann also im laufenden Betrieb einige Nachrichten, welche als geliefert und unbestätigt markiert sind, welche die Tasks die derzeit vom System in Bearbeitung sind wiederspiegeln. Wenn ActiveMQ die Verbindung zu einem Arbeiterthread verliert, wird der Task, auf Grund des fehlenden Client-Acknowledge, erneut in die Task Warteschlange eingefügt. Damit haben wir unerwartete Ausfälle von Arbeiterthreads elegant gelöst. Nicht nur das, wir müssen dadurch Arbeiterthreads nicht gracefully abschalten. Ein Aspekt der hier nicht beachtet wurde, die Laufzeit eines Tasks kann sehr lang sein, d.h. man muss potentiell noch manuell Konfigurationen bearbeiten um nicht in Timeouts von ActiveMQ zu laufen. Ebenso ungeklärt, was passiert mit den gelieferten unbestätigten Nachrichten, wenn das ActiveMQ ausfällt. ActiveMQ persistiert seine Task Warteschlange automatisch. Standardmäßig persistiert ActiveMQ seine Warteschlange mit binären Logs auf der Festplatte des Server. Fällt der Verwalter unerwartet aus, dann können die Arbeiter immernoch die Tasks die in der ActiveMQ Warteschlange eingetragen sind abarbeiten. Während der Verwalter offline ist, kann der Nutzer dem

System keinen neuen Task übergeben und das System reagiert auch nicht darauf wenn der Nutzer eine Eingabe aus dem Google Drive Eingabeordner entfernt.

Kapitel 5

Nicht kooperative verteilte Lastenverteilung mit synchronisiertem Zugriff auf Google Drive Eingabeordner

In diesem Lösungsansatz haben wir viele verteilte, zustandslose Threads. Jeder Thread holt selbständig Tasks aus dem Eingabeordner und führt diese aus. Die relationale Datenbank ist für die verteilten Threads zur Koordination um doppelte Taskausführungen zu vermeiden und um Fehlertoleranz gegen ausgefallene Threads zu erreichen. Ein Thread gilt als ausgefallen, wenn er durch einen Systemabsturz, Schließen der Anwendung oder ähnlichen Events außerhalb unserer Kontrolle eine lange Zeit oder garnicht mehr ausgeführt wird. Diese Lösungsalternative ist implementiert, dessen Implementierung wird in Kapitel 8 beschrieben.

Es folgt eine illustrative Abbildung 5.1 welche die Lösungsalternative und die Interaktionen zwischen den Komponenten verbildlicht. Diese Komponenten und Interaktionen werden in den kommenden Sektionen detailliert erläutert und die Abbildung dient zur Orientierung so wie ein kompakten Überblick über diese Lösungsalternative. Dabei sind die Ovale die einzelnen Systeme / Komponenten der Lösungsalternative. Die Pfeile sind logische Aufrufe an andere Systeme.

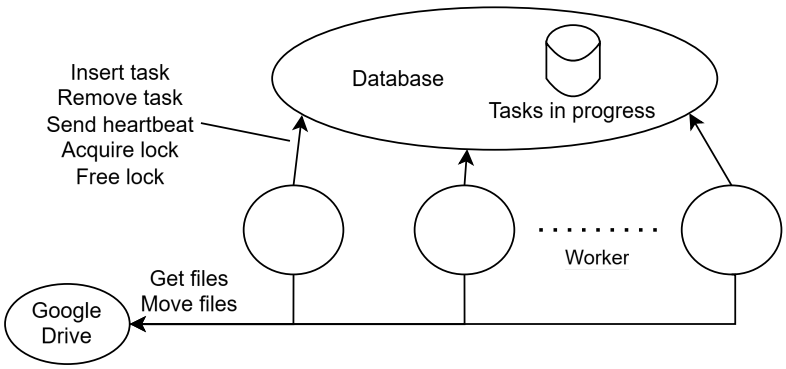

Abbildung 5.1: Verteilte LV mit sync. Zugriff auf Google Drive

5.1 Komponenten und Infrastruktur

Die Lösungsalternative umfasst eine einzelne Applikation. Von der Applikation gibt es mindestens eine Instanz. Die Applikation verwendet eine relationale Datenbank. Die Applikation hat so viele Threads wie Prozessorkerne zur Verfügung stehen. Dadurch können wir den Prozessor voll auslasten, in Sektion 3.5.2 haben wir uns kurz auseinandergesetzt mit einem Ansatz der mehr Threads als Prozessorkerne vorsieht.

5.2 Taskerkennung

Die erste Lösungsalternative in Kapitel 4 nutzt zur Erkennung neuer Tasks einen Polling Ansatz und hat einen eventbasierten Ansatz diskutiert. In diesem Lösungsansatz verwenden wir ebenfalls aus gleichem Grund einen Polling Ansatz. Im Gegensatz zur ersten Lösungsalternative haben wir keine weitere Datenstruktur welche die Menge der offenen Tasks wiederspiegelt. Die Menge der offenen Tasks sind die Dateien welche im Google Drive Eingabeordner liegen. Das System erkennt neue Tasks durch die Existenz derer im Eingabeordner implizit. Das System erkennt also nicht explizit das Ablegen der Datei im Eingabeordner vom Nutzer.

5.3 Verteilung von Tasks auf Ressourcen und Zeit

So wie in der ersten Lösungsalternative haben wir dieselbe Google Drive Ordnerstruktur mit vier Google Drive Ordnern für die vier Task-Zustände.

Möchte ein Thread einen Task bearbeiten, schiebt er den Task in den Executing Ordner. Um Fälle zu vermeiden in welchen zwei Threads die gleiche Datei zeitgleich verschieben, setzen wir sperrbasierte Synchronisation, mit Hilfe der relationale Datenbank, um. Genauer wird dies in folgender Sektion 5.4 behandelt.

Betrachten wir nun wie das System die Tasks auf seine Rechenressourcen verteilt. Das System verteilt die Tasks an die Threads die nichts zu tun haben dadurch dass jeder Thread sich, wenn er nichts zu tun hat einen neuen Task holt. Das ist ein dynamischer verteilter nicht kooperativer Lastenverteilungsansatz (3.2.2). Die Arbeiter kommunizieren nicht untereinander, so fällt keinerlei Kommunikationsaustausch an. Sind genügend Tasks verfügbar dann sind die Prozessorkerne immer ausgelastet. Aus Scheduling-Perspektive ist unser Ansatz greedy, da jeder Thread sich unabhängig von anderen Threads einen Task holt. Der Ansatz ist fair, da alle Threads die gleiche Chance haben einen Task zu holen. Unsere Verteilung der Tasks hat nur Overhead in Form der Synchronisation und Fehlertoleranz die in den zwei folgenden Sektionen erläutert werden. Dadurch ist die Ressourcenauslastung sehr hoch und wir erwarten eine sehr performante Lösung.

Gegenüber der ersten Lösungsalternative in Kaptiel 4 haben wir durch die verteilte Lastenverteilung keine zentrale Instanz mehr welche einen potentiellen Performanz-Bottleneck und Single Point of Failure darstellt.

Aus den gleichen Gründen wie in Kapitel 4.3 brauchen wir kein preemptive-Scheduling und auch keine dynamic-reassignments.

Zuletzt betrachten wir kurz wie ein Arbeiter einen bestimmten Task auswählt. Da wir keine Reihenfolge oder Priorisierung einzelner Tasks haben, muss diese Auswahl nicht bewusst getroffen werden und die Reihenfolge ist beliebig. Ist ein Task in Bearbeitung wird diese von der Anwendung nicht mehr unterbrochen. Dadurch wird der Task dann schnellstmöglich fertig ausgeführt.

5.4 Verhindern von doppelten Ausführungen

Das Verschieben einer Eingabedatei aus dem Google Drive Eingabeordner in einen anderen Google Drive Ordner gilt als Task-Zustand von offen auf in Bearbeitung setzen. Diese Operation muss mit allen Systemen synchronisiert sein um doppelte Taskausführung zu verhindern wie in Sektion 3.5.3 erstmals beschrieben.

Wir können als Synchronisierungsmechanismus jedoch nicht Google Drive alleine verwenden. Das Verschieben einer Google Drive Datei wird über einen API-Aufruf an die Google Drive API angestoßen. Aus der Antwort des API-Aufrufs kann der Aufrufer jedoch nicht erkennen ob sein Aufruf zum Verschieben der Google Drive Datei geführt hat oder ob die Datei bereits im gleichen Google Drive Ordner lag. Also muss der Arbeiter zuerst schauen ob die Datei des Tasks noch im Google Drive Eingabeordner liegt und anschließend verschieben. Da diese zwei Operationen nicht atomar sind, kann es zu Mehrbenutzerkonflikten kommen und dadurch zu doppelten Taskausführungen.

Um doppelte Ausführungen zu verhindern synchronisieren wir Threads beim Holen eines Tasks mit einer relationalen Datenbank. Konkreter, zu einer gegebenen Zeit kann nur ein einzelner Thread eine Eingabedatei aus dem Eingabeordner in den Executing Ordner verschieben. Ist der Task im Executing Ordner wird kein weiterer Thread diesen Task auswählen, da ein Thread sich einen neuen Task immer aus dem Eingabeordner sucht. So bearbeiten keine zwei Threads den gleichen Task.

Wir wollen mit der gegebenen relationalen Datenbank diese sperrbasierte Synchronisation implementieren. Bei zwei konkurrierenden Threads muss das Lock holen für den zweiten Thread direkt erfolglos sein. Wir betrachten zuerst das Nutzen der Transaktionen von relationalen Datenbanken. Bei zwei konkurrierenden Transaktionen tritt der Fall auf, die erste Transaktion macht ein Commit und die zweite Transaktion macht daraufhin einen Rollback. Das Problem steckt im Rollback. Innerhalb der Transaktion wurden bereits Änderungen in anderen Systemen vorgenommen, z.B. Verschieben von Dateien im Google Drive, welche man nicht sauber zurückrollen kann. Möchte man den Sperrmechanismus mit dem Transaktionsmodell von relationalen Datenbanken dennoch nutzen, dann muss sichergestellt werden dass jede Datenbankoperation stets auf den neusten Daten ausgeführt wird, auch wenn diese neusten Daten von noch laufenden Transaktionen stammen. Das bedeutet eine Transaktion muss von einer anderen Transaktion geänderte und nicht committete Änderungen sehen, das nennt man Dirty-Read. Das Nutzen eines Transaktionsmodells welches Dirty-Reads zulässt sollte aufmerksam machen, so eine Umsetzung wäre unsauber. Im Falle von PostgreSQL sind Dirty-Reads nie möglich [51], da PostgreSQL keine Implementierung von Transaktionen anbietet bei der Dirty-Reads möglich sind. In unserem Ansatz ist deshalb das Lock abholen eine Transaktion und das Freigeben des Locks eine Transaktion. Das Lock abholen spiegelt sich durch das Ändern eines Datensatzes in einer Tabelle wieder, analog das Freigeben des Locks. Die ACID-Kriterien von Datenbanken helfen uns hier den logischen Schritt Lock holen und Lock freigeben konsistent auch bei konkurrierenden Anfragen umzusetzen.

Aufgrund der Dynamik langer und unterschiedlicher Task-Ausführungszeiten erwarten wir wenig Konkurrenz beim Lock holen und freigeben. Deshalb entscheiden wir uns für ein optimistischen Ansatz unseres Sperrens. Die erste Anfrage an unsere Datenbank ist damit bereits, “Ich habe das Lock!”, statt bei einem pessimistischen Ansatz wo die erste Frage eher in die Richtung ginge “Ist das Lock frei?”. Ein optimistischer Ansatz bietet den Vorteil im Falle ohne Konkurrenz etwas weniger Overhead zu produzieren.

Die Logik ein Lock zu holen und Lock freizugeben durch Änderungen an Datensätzen wollen wir uns im Folgenden genauer ansehen. In der Datenbank haben wir in einer Tabelle eine Zeile, diese soll als logisches Lock dienen. Zum Akquirieren des Locks setzt ein Arbeiter ein Update Befehl an die Tabelle ab. Der Arbeiter setzt sich als neuen Besitzer des Locks. Entscheidend dabei ist die Where Bedingung des Update Statements. Die Bedingung muss prüfen, dass das Lock vorher keinen Besitzer hatte. Aus der

Rückgabe kann der Arbeiter erkennen, ob sein Update Statement zu einer Änderung geführt hat, wenn ja war die Operation erfolgreich und er ist Besitzer des Locks. Wenn nein, dann war ein konkurrierender Arbeiter schneller und das Lock Holen ist erfolglos.

Besitzt der Arbeiter das Lock verschiebt er die Eingabedatei in den Executing Google Drive Ordner und gibt das Lock frei. Beim Freigeben wird der Datensatz erneut modifiziert und das Attribut für den Lockbesitzer wird zurückgesetzt.

5.5 Skalieren des Systems

Das Skalieren gestaltet sich durch den Ansatz der zustandslosen verteilten Arbeiterthreads recht einfach. Arbeiterthreads können beliebig horizontal, durch Hinzufügen oder Entfernen von Anwendungsinstanzen auf weiteren Hardwareressourcen, skaliert werden. Das einfache Skalieren ist für den Nutzer sehr wichtig, dadurch kann er den Umfang seiner eingesetzten Hardwareressourcen seiner aktuellen Arbeitslast sehr genau und ohne großen Aufwand anpassen.

Der Server der Datenbank muss leistungsfähig genug sein um die Anfragen der Arbeiterthreads in einer sinnvollen Zeit abzuarbeiten. Der Server kann vertikal skaliert werden durch das Austauschen der Hardware gegen eine Leistungsfähigere.

5.6 Fehlertoleranz

Diese Lösungsalternative deckt folgende zwei Fehlerszenarien ab.

Szenario eins, Ausfall des Threads der das Lock zum Verschieben der Eingabedatei in den Executing-Ordner hält. Da der Thread der das Lock geholt hat es eigentlich selbst wieder freigeben muss, ergibt sich das Problem, dass das Lock bei dessen Ausfall nie mehr freigegeben würde. Dadurch würde das System komplett blockiert werden und keine weiteren Tasks könnten mehr gestartet werden. Das System muss erkennen können ob der Arbeiterthread der das Lock besitzt noch aktiv ist. Eine einfache Lösung dieses Problems, wir geben einem Lock eine Gültigkeitsdauer. Akquiriert ein Arbeiterthread das Lock, dann hat er Anrecht auf das Lock bis er es zurückgibt oder die Gültigkeitsdauer abgelaufen ist. Ist die Gültigkeitsdauer des Locks abgelaufen gilt es als ungültig und darf von anderen Arbeiterthreads übernommen werden. Die Wahl der Größe des Gültigkeitsdauer ist entscheident, wir betrachten die Extrema. Wählt man die Dauer zu klein, können zwei konkurrierende Threads sich zeitgleich im kritischen Bereich aufhalten und der Sperrmechanismus ist außer Kraft gesetzt. Wählt man die Dauer zu groß, wird das System im Fehlerszenario sehr lange blockiert. Die Logik des Task Holen hat konstant viele Schritte. Da wir beim Task Holen die Google Api verwenden, also ein externes System nutzen sollte man auf einen durchschschnittlichen Wert einen großzügigen Puffer draufrechnen. Diesen Wert kann man dann als Gültigkeitsdauer verwenden.

Umgesetzt ist dieser Mechanismus durch das zusätzliche Speichern des Zeitpunkts in welchem das Lock geholt wurde. Dann kann die Differenz zur aktuellen Zeit mit der konfigurierten Gültigkeitsdauer verglichen werden und dementsprechend das Lock übernommen werden. Dieser Timeout des Locks beeinflusst den fehlerfreien Betrieb in keiner Weise und sollte eher großzügig gewählt werden. Wenn der Timeout noch nicht abgelaufen ist können wir daraus schließen, dass ein anderer Thread derzeit einen Task abholt und der Thread soll warten.

Szenario zwei, ein Thread fällt während der Bearbeitung eines Tasks aus. Ursachen hierfür können Beenden des Applikationsprozess sein, permanenter Netzwerkverlust, Absturz des Servers oder ähnliches.

Als Lösung brauchen wir einen Mechanismus mit welchem wir sehen können welche Tasks in Bearbeitung sind und ob ein Arbeiterthread noch aktiv an seinem Task arbeitet. Ist ein Arbeiterthread nicht mehr aktiv weil er ausgefallen ist, dann kann ein anderes System diesen Ausfall erkennen, den Task aufräumen und wieder als offen markieren, so dass dieser später erneut ausgeführt werden kann. Mit Aufräumen und als offen markieren meinen wir, dass der Task für das System so aussieht als wäre er gerade vom Nutzer hinzugefügt worden darunter das Verschieben des Tasks zurück in den Google Drive Eingabeordner. Die nötigen Daten hierfür legen wir wieder in der Datenbank ab um die Arbeiterthreads zustandslos zu halten.

Wir implementieren einen Heartbeat-Mechanismus. Grob erläutert, jeder Arbeiter hat einen Herzschlag während er einen Task bearbeitet. Dieser ist für die anderen Systeme einsehbar, weil die Daten in der Datenbank abgelegt werden. Unter den Daten ist auch der Task den der Arbeiter derzeit ausführt. Ein Arbeiter sendet periodisch Updates an die Datenbank, bildlich sein Herzschlag. Setzt der Herzschlag eines Arbeiters zu lange aus, wird dieser als tot erklärt und den Task den er bearbeitet hat wird wieder als offen markiert. Genauer erläutert ist dieser Mechanismus im Implementierungskapitel (8).

Kapitel 6

Nicht kooperative verteilte Lastenverteilung mit synchronisiertem Zugriff auf einzelne Tasks

Diese Lösungsalternative verwendet Aspekte der ersten und zweiten Lösungsalternative aus Kapitel 4 und 5. Wie in Lösungsalternative zwei haben wir hier viele verteilte, zustandslose Threads. Auch hier holt sich jeder Thread Tasks und führt diese aus.

Gegenüber der Lösungsalternative zwei kommt noch ein einzelner weiterer zustandsloser Thread hinzu, der Verwalter, der Tasks aus dem Eingabeordner in eine Datenbanktabelle einträgt. Der Verwalter ist fast gleich wie der Verwalter aus der ersten Lösungsalternative. Eine Zeile der Tabelle entspricht einem Task. Jeder Arbeiter holt sich die Tasks aus der Datenbank statt aus dem Google Drive Eingabeordner. Die Datenbank ist hier so wie in Lösungsalternative zwei zur Koordination gegen doppelte Taskausführungen und für die Fehlertoleranz eingesetzt.

Es folgt eine illustrative Abbildung 6.1 welche die Lösungsalternative und die Interaktionen zwischen den Komponenten verbildlicht. Diese Komponenten und Interaktionen werden in den kommenden Sektionen detailliert erläutert und die Abbildung dient zur Orientierung so wie ein kompakten Überblick über diese Lösungsalternative. Dabei sind die Ovale die einzelnen Systeme / Komponenten der Lösungsalternative. Die Pfeile sind logische Aufrufe an andere Systeme.

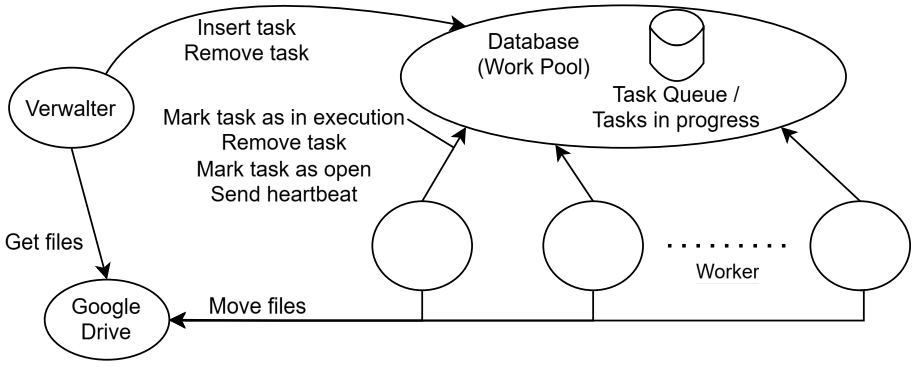

Abbildung 6.1: Verteilte LV mit sync. Zugriff auf einzelne Tasks

6.1 Komponenten und Infrastruktur

Wir haben eine Verwalterapplikation, von dieser gibt es exakt eine Instanz. Mindestens eine Instanz der Arbeiterapplikation und exakt eine Instanz einer relationalen Datenbank.

6.2 Taskerkennung

So wie in der ersten Lösungsalternative in Kapitel 4.2 erkennt der Verwalter durch pollen des Google Drive Eingabeordners neue Eingabedateien. Der Ablauf ist dabei gleich. Der Verwalter filtert aus den FileIds des Google Drive Eingabeordners die FileIds die in der Datenbanktabelle sind und fügt für diese neue Datenbankeinträge ein. Für die FileIds aus der Datenbanktabelle für offene Tasks welche nicht mehr im Google Drive Eingabeordner sind werden die Datenbankeinträge gelöscht. Die letztgenannten FileIds gehören zu Tasks welche vom Nutzer wieder aus dem Eingabeordner entfernt wurden.

6.3 Verteilung von Tasks auf Ressourcen und Zeit

Der Lastenverteilungsansatz und Scheduling-Ansatz ist exakt derselbe wie in der ersten Lösungsalternative in Kaptiel 4. Anstatt der ActiveMQ Warteschlange haben wir eine Tabelle in der relationalen Datenbank. Ein Arbeiter holt sich einen Task durch Modifizieren des Datenbankeintrags des Tasks, führt diesen aus und wiederholt den Prozess. Genau wie in Kapitel 4 erwarten wir dass der Work-Pool Ansatz so wie die greedy und faire Schedulingstrategie uns große Beschleunigung und Ressourcenauslastung bringt.

6.4 Verhindern von doppelten Ausführungen

So wie in der zweiten Lösungsalternative in Kapitel 5.4 synchroniseren wir die Arbeiter über eine relationale Datenbank. Dabei nutzen wir denselben Mechanismus der dort zum Lock Akquirieren verwendet wird für das Akquirieren eines Tasks.

In der Datenbanktabelle stehen alle offenen Tasks und Tasks die in Bearbeitung sind. Zwei konkurrierende Arbeiter wählen aus der Tabelle den gleichen offenen Task aus und versuchen diesen zu akquirieren. Über Update Statements versuchen beide Arbeiter den Task als von sich in Bearbeitung zu markieren. Im Update Statement ist in der Where Bedingung die Kondition, dass der Task davor noch offen war. Die konkurrierenden Arbeiter erkennen ob ihr Update Statement zu einer Änderung geführt hat und analog dazu, ob sie den Task erhalten haben oder nicht. Gehört dem Arbeiter der Task, darf er ihn ausführen und nach Abschluss der Ausführung den Tabelleneintrag des Tasks löschen.

6.5 Skalieren des Systems

Horizontales Skalieren der Arbeiter ist problemlos möglich. Der Verwalter und die Datenbank können nur vertikal skaliert werden.

6.6 Fehlertoleranz

Wir betrachten das Ausfallszenario zwei aus der zweiten Lösungsalternative in Kapitel 5.6, wenn ein Thread während des Bearbeitens eines Tasks unerwartet ausfällt. Das Problem ist sein Task ist als in Bearbeitung markiert und muss wieder als offen markiert werden. Auch hier implementieren wir einen Heartbeat-Mechanismus. Der Heartbeat Mechanismus verwendet die existierende Tabelle und der Arbeiter

sendet einen Herzschlag an den Tabelleneintrag zu dem Task den er bearbeitet. Bleibt dieser zu lange aus, verliert der Arbeiter den Task und der Task wird wieder als offen markiert.

Kapitel 7

Bewertung und Entscheidung der Lösungsalternativen

In diesem Kapitel werden die drei vorgestellten Lösungsalternativen verglichen und eine Entscheidung gefällt.

Wir wollen zuerst die Lösungsalternativen bewerten, dazu vergleichen wir diese zuerst und gehen anschließend auf die Lösungsalternativen einzeln ein. Wir betrachten wie jede Lösungsalternative gegen die anderen Lösungsalternativen abschneidet und begründen damit unsere Entscheidung.

7.1 Vergleich

Im folgenden ist eine Tabelle auf der in den Zeilen Kriterien und in den Spalten die drei Lösungsalternativen zu finden sind. Die Kriterien beinhalten relevante Aspekte um die Lösungen besser bewerten und vergleichen zu können. Die Kriterien werden zuerst kurz vorgestellt und in Kontext gesetzt.

- **Infrastruktur- komponenten bei n Arbeiterapplikationen:** Was für Komponenten beinhaltet die Lösungsalternative. Jede Komponente muss implementiert und dessen Installation verwaltet werden, das Kriterium gibt einen Indiz auf dessen Infrastrukturanforderungen.
- **Ist Arbeiterapplikation zustandslos?:** Eine zustandslose Arbeiterapplikation ist hilfreich für das einfache horizontale Skalieren der Arbeitermenge.
- **Beschleunigung (Durchsatz, Makespan, Response Time, Ressourcenauslastung):** Wie hoch ist die Beschleunigung durch die Lösungsalternative. Dabei ist der Ausschluss doppelter Ausführungen ausschlaggebend. Zudem ist wichtig, dass alle Arbeiter Tasks bekommen sofern welche verfügbar sind.
- **Fehlertoleranz:** Welche Schwachstellen bezüglich der Fehlertoleranz haben wir.
- **Synchronisierte logische Operationen pro Task:** Welche logischen Schritte werden im synchronisierten Bereich, also maximal von einem Arbeiter zeitgleich, ausgeführt. Das Kriterium gibt einen Indiz über die Größe dieses nicht parallel ausgeführten Teil der Logik.
- **Technischer Overhead für synchronen Bereich ohne Konkurrenz:** Gibt den Overhead durch die technische Umsetzung der Synchronisierung an.
- **Task Reservieren mit Konkurrenz:** Gibt an ob beim Reservieren des Tasks konkurrierende Arbeiter blockieren und ob konkurrierende Arbeiter Kollisionen beim Reservieren erwarten müssen.

- **Task Reservieren mit Konkurrenz, Ausführungszeit bei n Arbeitern:** Gibt die durchschnittliche Ausführungszeit für einen von n Arbeitern an die er benötigt bis er einen Task reserviert hat. Alle n Arbeiter versuchen also gleichzeitig einen Task zu holen. Dabei lassen wir lokale Ausführungszeit außen vor und fokussieren uns auf Aufrufe an andere Systeme, da diese durch den Netzwerkoverhead deutlich langsamer sind.
- **Folgen eines Ausfall der koordinierenden Einheit:** Was sind die Auswirkungen auf das System wenn die koordinierende Komponente des Sytems unerwartet ausfällt.
- **Folgen eines Ausfall des Verwalters:** Was sind die Auswirkungen auf das System wenn die Verwalterkomponente, falls sie existiert, ausfällt.
- **Folgen eines Ausfalls eines Arbeiters:** Was sind die Auswirkungen auf das System wenn ein einzelner Arbeiter ausfällt.
- **Google API Anfragefrequenz wenn n Arbeiter idlen, Pollingfrequenz p:** Wie viel Last übt das System auf die Google API im worst case aus. Kriterium kann auf Grund der Nutzungsbeschränkungen der Google API wie in 4.2 angesprochen relevant sein.

Es folgt die Tabelle, in der zweiten Spalte ist Lösungsalternative eins zu finde, in der dritten Spalte Lösungsalternative zwei und in der hintersten Spalte Lösungsalternative drei.

Kriterien	**Work-Pool LV**	**Verteilte LV mit sync. Zugriff auf Google Drive**	**Verteilte LV mit sync. Zugriff auf einzelne Tasks**
Infrastruktur-komponenten bei n Arbeiterapplikationen	ActiveMQ, Verwalter, n Arbeiter	Datenbank, n Arbeiter	Datenbank, Verwalter, n Arbeiter
Ist Arbeiterapplikation zustandslos?	ja	ja	ja
Beschleunigung (Durchsatz, Makespan, Response Time, Ressourcenauslastung)	Durch Ausschluss von doppelten Ausführungen maximal	Durch Ausschluss von doppelten Ausführungen maximal	Durch Ausschluss von doppelten Ausführungen maximal
Fehlertoleranz	ActiveMQ und Verwalter sind Single Points of Failure	Datenbank ist Single Point of Failure	Datenbank und Verwalter sind Single Points of Failure
Synchronisierte logische Operationen pro Task	Task Reservieren	Task Auswählen, Task Reservieren	Task Reservieren
Technischer Overhead für Task Reservieren ohne Konkurrenz	Modifizieren eines Eintrags in einer ActiveMQ Warteschlange und ein Google API Aufruf	2 DB Update Transaktionen und ein Google API Aufruf	1 DB Update Transaktion und ein Google API Aufruf
Task Reservieren mit Konkurrenz	nicht blockierend, kollisionsfrei	blockierend, kollisionsfrei	nicht blockierend, kollisionsbehaftet
Task Reservieren mit Konkurrenz, Ausführungszeit bei n Arbeitern	$ActiveMQCall + GoogleAPICall$	$n \cdot DatabaseCall + n/2 \cdot GoogleAPICall$	$n/2 \cdot (DatabaseCall + GoogleAPICall)$ *
Folgen eines Ausfall der koordinierenden Einheit	Tasks laufen zuende. Danach blockiert System	Tasks laufen zuende. Danach blockiert System	Tasks laufen zuende. Danach blockiert System
Folgen eines Ausfall des Verwalters	Keine neuen Tasks werden erkannt	-	Keine neuen Tasks werden erkannt
Folgen eines Ausfalls eines Arbeiters	ActiveMQ bemerkt Ausfall und nimmt Task erneut in Warteschlange auf	Ein Arbeiter markiert den verlassenen Task wieder als offen	Ein Arbeiter markiert den verlassenen Task wieder als offen
Google API Anfragefrequenz wenn n Arbeiter idlen, Pollingfrequenz p **	p	$p \cdot n$	p

Tabelle 7.1: Vergleich verschiedener verteilter LV-Modelle

* Bei der Ausführungszeit für das Task Reservieren wurde hier der worst case angenommen, bei welchem jeder der n Arbeiter stets versuchen denselben Task zu holen. Wählen die Arbeiter ihren nächsten Task zufällig wäre die Ausführungszeit geringer.

** In Lösungsalternative eins und drei führt der Verwalter das Polling aus unabhängig davon ob die Arbeiter beschäftigt sind oder nicht. Bei Lösungsalternative zwei führen nur Arbeiter die keinen Task bearbeiten das Polling aus. In einem Szenario in welchem wir viele Tasks haben und die Arbeiter dauerhaft beschäftigt sind, ist die Anzahl an Google API Anfragen durch das Polling in Lösungsalternative zwei geringer.

7.2 Work-Pool LV

Während die anderen zwei Lösungsalternativen einzelne logische Bestandteile, wie den Ausschluss von doppelten Ausführungen und den Heartbeat-Mechanismus selbst auf Basis einer relationalen Datenbank implementieren, greifen wir bei dieser Lösungsalternative auf eine Enterprise Lösung zurück. Die Entwickler von ActiveMQ haben sich sicherlich in ihrer Entwicklung auch ähnlichen Challenges gestellt und diese überwunden und implementiert. Ihr theoretischer Work-Pool Ansatz wie wir in Kapitel 4 erläutert haben ist für unser Problem gut geeignet.

Möchte man diese Alternative implementieren muss man sich recht intensiv mit den Details von ActiveMQ auseinandersetzen. Das Client-Acknowledge Feature, welches verwendet wird um doppelte Ausführungen zu vermeiden und ebenfalls als Fehlertoleranzmechanismus ist wichtiger Bestandteil dieser Lösungsalternative. ActiveMQ wurde zwar konzeptionell investigiert, es wurde jedoch im Rahmen der Arbeit kein experimenteller Prototyp implementiert.

Wie in den Vorgaben 1.3.1 bereits angegeben, ist in unserem Kontext bereits eine relationale Datenbank vorhanden, so fällt diese Lösungsalternative wegen ihrer Infrastrukturanforderungen raus. Ohne diese Vorgabe ist diese Lösungsalternative sehr interessant und ein weiterer investigativer Aufwand ob sie umsetzbar ist wäre lohnenswert.

7.3 Verteilte LV mit sync. Zugriff auf Google Drive

Diese Lösungsalternative hat die geringste Anzahl an Komponenten. Ein Deployment umfasst lediglich die relationale Datenbank und Arbeiterapplikationen. Die geringe Komplexität an die Infrastrukturanforderungen hat die Entscheidung für diese Lösungsalternative ausgelöst. Ihr theoretischer Ansatz ist wie in Kapitel 5 erläutert gut für unser Problem geeignet. Während dem Implementierungsprozess sind Shortcomings dieser Lösungsalternative aufgefallen, welche in den Shortcomings 8.8 weiter erläutert werden. Darunter der Fakt, dass Arbeiter einen Task als offen sehen, wenn er im Google Drive Eingabeornder liegt. Diese Abhängigkeit von der Google API ist unsauber und die anderen zwei Lösungsalternativen haben das mit ihren selbst verwalteten Datenstrukturen besser gelöst. In Form der ActiveMQ Warteschlange bei der ersten Lösungsalternative und Datenbanktabelle der dritten Lösungsalternative.

7.4 Verteilte LV mit sync. Zugriff auf einzelne Tasks

Lösungsalternative drei enthält Aspekte beider anderer Lösungsalternativen. Der grundlegende Work-Pool Lastenverteilungsansatz ist wie bei Lösungsalternative eins ein gut geeigneter Ansatz, wie in Kapitel 6 erläutert. Die Mechanismen um doppelte Taskausführungen zu vermeiden und Fehlertoleranz zu erreichen sind von Lösungsalternative zwei übernommen und clever kombiniert. Der Vorteil gegenüber der zweiten Lösungsalternative ist, dass eine einzelne Datenbanktabelle sowohl die Tasks hält welche gerade von

Arbeitern bearbeitet werden, als auch alle offenen Tasks. Das bietet den Vorteil, dass das Ändern des Task-Zustands auf in Bearbeitung lediglich das Modifizieren einer einzelnen Datenbankzeile ist. Eine kleine und atomare Operation und wie in Kapitel 5 bereits erläutert sehr gut geeignet um Mehrbenutzerkonflikte zu verhindern. Bei Lösungsalternative zwei umfasst das Ändern des Task-Zustands auf in Bearbeitung das Verschieben der Eingabedatei aus dem Google Drive Eingabeordner heraus und einen Datenbankzugriff wegen des Heartbeat-Mechanismus.

Die Funktionalität des Verwalters, der in Lösungsalternative zwei nicht existiert, kann durchaus in die Arbeiterapplikation verschoben werden. Die Tätigkeit nach neuen Tasks zu pollen ist ja in der zweiten Lösungsalternative ebenfalls in der Arbeiterapplikation umgesetzt. Dadurch wäre die Infrastrukturanforderung dieser Alternative gleich hoch, wie von Lösungsalternative zwei und möglicherweise sogar die bessere Wahl. In der Entscheidungsfindung wurde es damals für notwendig gefunden keine eigene Datenstruktur für das Halten aller offenen Tasks zu pflegen sondern dafür direkt den Google Drive Eingabeordner zu verwenden. Deswegen hatte man sich für die zweite Lösungsalternative entschieden, jedoch war diese Einschätzung eine premature Optimization wie in den Shortcomings 8.8 noch genauer erläutert wird.

Kapitel 8

Implementierung

In diesem Kapitel wird, grob der Struktur der Kapitel der Lösungsalternativen folgend, die Softwarearchitektur und Implementierung vorgestellt und erläutert.

Der Umfang der Implementierung ist eine kleinste funktionale Implementierung des Task-Processing. Wir haben als unterstützte Tasks den Overfit-Backtest. Der Mechanismus parallel laufende Threads zu haben ist implementiert. Ebenfalls die Synchronisierung über die Datenbank mit dem Sperrmechanismus und Heartbeat-Mechanismus ist implementiert.

8.1 Technische Umgebung, benutzte Sprachen, Frameworks, Technologien

Das entwickelte Projekt, "Task-Processing", ist ein Java Spring Projekt und verwendet Java Version 17. Für uns relevante Abhängigkeiten von Task-Processing sind Spring-Boot, Spring-Boot-Data, PostgreSQL, Overfit und Google-Api-Services-Drive. Overfit enthält unter anderem Abhängigkeiten für Google-Api-Client, Google-OAuth-Client-Jetty, Google-Api-Services-Sheets und Google-Api-Services-Docs.

8.2 Inbetriebnahme

Um das System in Betrieb zu nehmen benötigt man mindestens ein System auf welchem die Anwendung ausgeführt werden kann und eine relationale Datenbank. Die Datenbank muss einmalig initialisiert werden und gestartet sein. Initialisierung der Datenbank umfasst das Erstellen zweier Tabellen und das Einfügen eines Datenpunkts. Die Anwendung muss außerdem konfiguriert werden, Konfigurationswerte müssen der Anwendung in Form von Property Values, z.B. durch Ablegen in der application.yaml Datei bereitgestellt werden. Konfigurationswerte umfassen Credentials für Datenbankverbindung und Google Api. Credentials für die Datenbank umfasst die URL der Datenbank, so wie Nutzername und Passwort. Für die Google Api muss das secret eines Google-Service-Accounts abgelegt werden. Dieser Google-Service-Account muss eingerichtet werden, so dass er Berechtigung zum Verwenden der Google Sheets, Google Drive und Google Docs Api hat. Außerdem muss der Nutzer zwei bis fünf benötigte Google Drive Ordner anlegen und dem Google-Service-Account der Anwendung Bearbeiterrechte geben. Der Nutzen dieser Ordner wird im weiteren Verlauf im Teil 8.3 weiter erläutert. Die Ids der Google Drive Ordner werden der Anwendung ebenfalls als Konfigurationswerte abgelegt. Als letzten Schritt muss die Task-Processing Anwendung gestartet werden und der Usecase ProcessAllTasks aufgerufen werden. Dann

laufen auf dem System Arbeiterthreads, welche kontinuierlich Tasks verarbeiten und das System ist nutzbar. Weitere Task-Processing Anwendungsinstanzen können auf gleichem wege zusätzlich gestartet werden.

8.3 Nutzerinteraktion

Möchte der Nutzer eine Chartanalyse ausführen, dann muss er im ersten Schritt eine Google Docs Datei erstellen, welche die YAML-Konfiguration für einen Task enthält. In einem zweiten Schritt verschiebt der Nutzer die Google Docs Datei in den Eingabeordner. Das Verschieben in den Eingabeordner gilt für das System als getätigte Eingabe. Die Eingabedatei wird vom System in den "Executing" Google Drive Ordner verschoben und signalisiert dem Nutzer damit, dass dieser Task derzeit in Bearbeitung ist. Nach erfolgreicher Bearbeitung des Tasks wird die Eingabedatei in den "OutputDocs" Google Drive Ordner verschoben und die Task-Ausgabe, im Falle von Backtest-Tasks, ist im "OutputSheets" Google Drive Ordner verfügbar. Bei nicht erfolgreicher Bearbeitung des Task, wird die Eingabedatei in den "Failed" Google Drive Ordner verschoben und im gleichen Ordner eine Textdatei erstellt, welche die Fehlerursache beinhaltet. Der Dateiname dieser Textdatei enthält die FileId der Google Docs Datei dessen Taskbearbeitung erfolglos war. In der Textdatei sind unter anderem die Nachrichten der geworfenen Exceptions zu sehen. Da das Overfitprojekt fachlich verwertbare Nachrichten und Exceptiontypen nutzt, kann das Fehlerprotokoll den Nutzer auf Konfigurationsfehler hinweisen.

Wie die fünf konfigurierten Google Drive Ordner vom Nutzer verwendet werden können haben wir gerade erläutert. Der Google Drive Eingabeordner "Input" ist, wie zuerst in Sektion 1.3.1 beschrieben, der Ort wo ein Nutzer seine Eingabe und wie gerade erläutert, tätigt. Der Google Drive Ausgabeordner für die Backtestergebnisse ist "OutputSheets" und enthält die Ausgaben der erfolgreich ausgeführten Backtests. Die anderen drei verwendeten Google Drive Ordner "Executing", "Failed" und "OutputDocs" dienen zur Übersichtlichkeit, so dass anhand des Orts der Eingabedatei der Status seines Tasks für den Nutzer einsehbar ist. Der Eingabeordner muss sich von den Ordnern "Executing", "Failed" und "OutputDocs" unterscheiden, sonst werden die Tasks nicht korrekt abgearbeitet.

8.4 Grundlegendes Design

Eine Anwendung soll mehrere unabhängige Threads beinhalten. Die Anzahl der Threads in einer Anwendung hängt von der Anzahl der verfügbaren Prozessorkerne ab. Jeder Thread führt eigenständig eine Arbeiterroutine aus. Die Arbeiterroutine ist als Schleife implementiert. Die Schleife umfasst alle Schritte zum Task-Processing, im Groben Task holen und anschließend das Task ausführen.

Die Threads koordinieren sich über die relationale Datenbank. Ein Zweck dieser Datenbank ist das Umsetzen eines Sperrmechanismus zwischen den Threads um gegenseitigen Ausschluss der Threads beim Task holen zu erreichen. Der andere Zweck der Datenbank ist es die Aktivität der Threads, die derzeit einen Task ausführen, zu verfolgen und bei Ausbleiben von Aktivität deren Task wieder als offen zu markieren.

Ein Task hat so wie in Sektion 3.5.3 vorgestellt einen von vier Zuständen, offen, in Bearbeitung, geschlossen oder fehlgeschlagen. Ein Task gilt als offen, wenn er im Eingabeordner liegt. Das ist auch der initiale Zustand eines Tasks wenn er durch den Nutzer hinzugefügt wird. Arbeiter suchen sich zur Ausführung immer einen offenen Task, also einen Task aus dem Eingabeordner. Ein Task muss immer erst in Bearbeitung gewesen sein, bevor er fertig oder fehlgeschlagen sein kann.

Threads sind unabhängig, so dass eine Anwendungsinstanz mit vier Kernen, sich gleich verhält wie zwei Anwendungsinstanzen mit je zwei Kernen. Das macht das Deployment einfacher, da man flexibel in der Auswahl an Systemen ist im Bezug darauf wie viele Prozessorkerne ein System hat. Die unabhängigen Threads kommunizieren nur mit der Datenbank und nicht untereinander.

Wir wollen keinen zentralen Thread haben, der sich um die Verwaltung der Threads in der Anwendungsinstanz kümmert, jeder Thread soll andauernd laufen und Fehler selbst behandeln. Im Fehlerfall wird der Fehler geloggt und die Arbeiterroutine startet erneut.

Die Datenbank wird für die Synchronisierung der Prozesse genutzt, wir speichern die notwendigen Daten für die Synchronisierung in der Datenbank. Das Transaktionsmodell von Datenbanken bietet uns die ACID Kriterien für unsere Operationen. Die ACID Kriterien sind für uns wichtig da die Synchronisierungsdaten für alle Systeme stets konsistent bleiben müssen.

Eingabedaten, Ausgabedaten und Fehlerprotokolle werden alle in Google Drive gespeichert. Google Docs als Eingabedateien ermöglichen dem Nutzer leichte Dateiverwaltung, Anpassung von unterwegs, man muss keine Kopien der Dateien erstellen usw. Fehlerprotokolle werden in txt Dateien in Google Drive abgelegt, so kann der Nutzer ebenfalls wieder die Cloudvorteile nutzen. Für die Ausgabedateien der Overfit-Tasks eignen sich Google Sheets sehr gut für die tabellarische Ausgabe der Chartanalysen, so kann der Nutzer möglicherweise direkt in der Ausgabe Google Sheets Funktionalität nutzen um das Ergebnis zu verfeinern ohne dieses zuerst kopieren zu müssen. Aus der Maintainer Sicht ist die Nutzung von solchen Google-Plattformdateien ebenfalls von Vorteil, da so keine lokalen Dateien in den Systemen gespeichert werden müssen und alles in die Google Drive Cloud ausgelagert ist. Nutzen der Google Cloud für Daten hat auch den Vorteil, dass von uns verwaltete Anwendungsserver zustandslos sind und nur die Datenbank persistente Daten hat. Das ermöglicht ein relativ flexibles Deployment.

8.5 Architektur und Anwendungskomponenten

Die folgenden Sektionen zeigen und erläutern die Architektur der Task-Processing Anwendung hierarchisch von oben nach unten. Wir beginnen an der obersten Abstraktionsebene und gehen schrittweise die Ebenen herab. Die Anwendung teilt ihre Logik in Komponenten auf. Eine Komponente hat eine Teillogik, eine Verantwortlichkeit, das soll dem Clean-Code-Prinzip “single-responsibility” nachgehen [52]. Eine Verantwortlichkeit kann das Kümmern um einen bestimmten Aspekt der Anwendung sein, wie z.B. das Ausführen eines Tasks, das ist in der Komponente `ExecuteTask` gekapselt. Dadurch wird die Logik verständlicher, übersichtlicher und Fehler sind leichter erkennbar. Die einzelnen Komponenten sind in Aktivitätsdiagrammen abgebildet, wo die Aktivitäten weitere Komponenten der Anwendung darstellen. Aktivitäten welche keine eigene Komponente darstellen werden durch einen gestrichelten Rand markiert. Eine Anwendungskomponente ist im Java-Code eine Klasse oder Interface mit dem Namen der Komponente und einer einzelnen `execute(..)` Methode. Das Spring-Framework, wie in 2.2.1 erläutert, instanziiert dann jeder Komponente die benötigten anderen Komponenten. Im Methodenkörper der `execute(..)` Methode führt die Komponente dann ihre Teillogik aus. Wenn wir vom Ausführen einer Komponente reden, dann meinen wir das Aufrufen der `execute(..)` Methode.

8.5.1 “ProcessAllTasks”

Der Usecase `ProcessAllTasks` stellt den Einstiegspunkt der Anwendung dar. Aufgabe der Komponente ist es die verfügbaren Systemressourcen, Anzahl der Prozessorkerne, zu ermitteln und den Start gegeben vieler Arbeiter auszulösen. Im folgenden Aktivitätsdiagramm (Abb. 8.1) sieht man die zwei zugehörigen Activities.

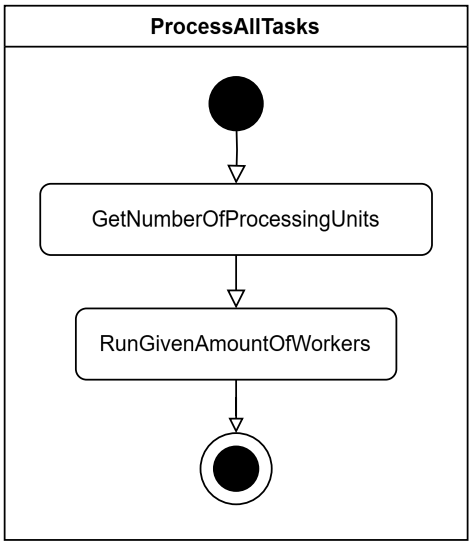

Abbildung 8.1: Aktivitätsdiagramm zur Softwarekomponente `ProcessAllTasks`

8.5.2 “RunGivenAmountOfWorkers”

`RunGivenAmountOfWorkers` ist dafür verantwortlich die korrekte Anzahl an Arbeiter zu starten und zu verwalten. Ein Arbeiter ist ein Thread welcher eine Arbeiterroutine ausführt. Ein Arbeiter hat eine Flag, die `keepWorkingFlag`, diese ist zu Beginn gehisst, also auf wahr gesetzt. Ist diese Flag gesenkt, also false, soll der Arbeiter keine weitere Arbeit anfangen und seinen Thread beenden. Dieser Mechanismus die Arbeiter sauber zu beenden wird derzeit nicht verwendet, stellt aber eine saubere Implementierung dar die Threads nach Erstellung wieder korrekt zu beenden. Möchte man in Zukunft die Anzahl der Arbeiter dynamisch verändern, kann dieser Mechanismus direkt verwendet werden. Im folgenden Aktivitätsdiagramm (Abb. 8.2) sieht man die Ablauflogik.

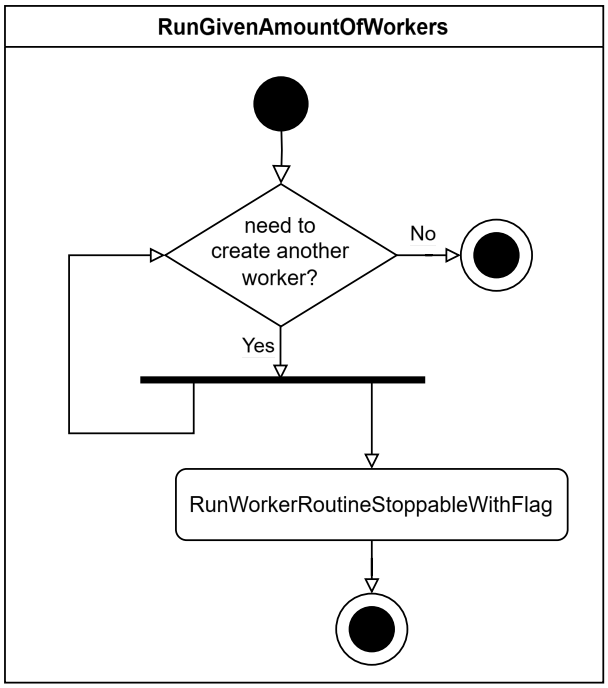

Abbildung 8.2: Aktivitätsdiagramm zur Softwarekomponente `RunGivenAmountOfWorkers`

8.5.3 "`RunWorkerRoutineStoppableWithFlag`"

`RunWorkerRoutineStoppableWithFlag` ist verantwortlich eine Arbeiterroutine auszuführen und Fehler zu fangen. Betrachte das folgende Aktivitätsdiagramm (Abb. 8.3) der Komponente. Die Arbeiterroutine `ContinuouslyDoTasks` ist selbst endlos, jedoch können Fehler auftreten, welche die Komponente selbst nicht behandelt. Diese Fehler können durch unerwartetes Verhaltenen externer Ressourcen ausgelöst werden oder schlichtweg nicht beachtete Fehlerszenarien sein. In allen Fällen soll ein Fehler in einem einzelnen Arbeiter nicht die komplette Anwendung und damit andere Arbeiter zum Absturz bringen. Stattdessen loggen wir den Fehler, so dass ein Maintainer bei Bedarf den Fehler einsehen kann. Die `keepWorkingFlag` wird ebenfalls nach Loggen des Fehlers beachtet und die Arbeiterroutine wird erneut gestartet.

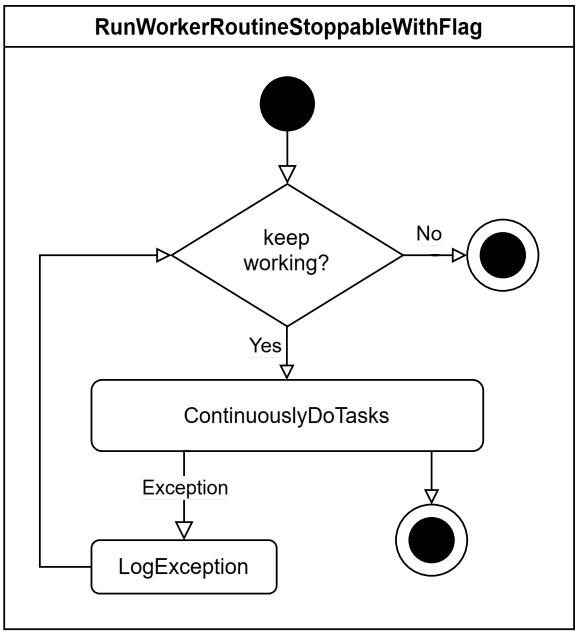

Abbildung 8.3: Aktivitätsdiagramm zur Softwarekomponente `RunWorkerRoutineStoppableWithFlag`

8.5.4 "`ContinuouslyDoTasks`"

`ContinuouslyDoTasks` teilt die Arbeiterroutine in eine Schleife einzelner Iterationen der Arbeiterroutine. Betrachte das folgende Aktivitätsdiagramm (Abb. 8.4) der Komponente. Dies ist die tiefste Ebene in welcher die `keepWorkingFlag` geprüft wird. Eine Iteration der Arbeiterroutine, dargestellt durch Komponente `DoOneWorkIteration` wird also nicht unterbrochen sondern wird zuende ausgeführt. Wenn der Arbeiter in dieser Schleifeniteration keinen Task ausgeführt hat wartet er eine gewisse Zeit. Selbst wenn kein Task ausgeführt wurde, hat der Arbeiter mehrere Anfragen an Datenbank und an Google Drive Api abgeschickt. Das ist eine Maßnahme um eine unnötig hohe Auslastung der Datenbank und Google API zu verhindern.

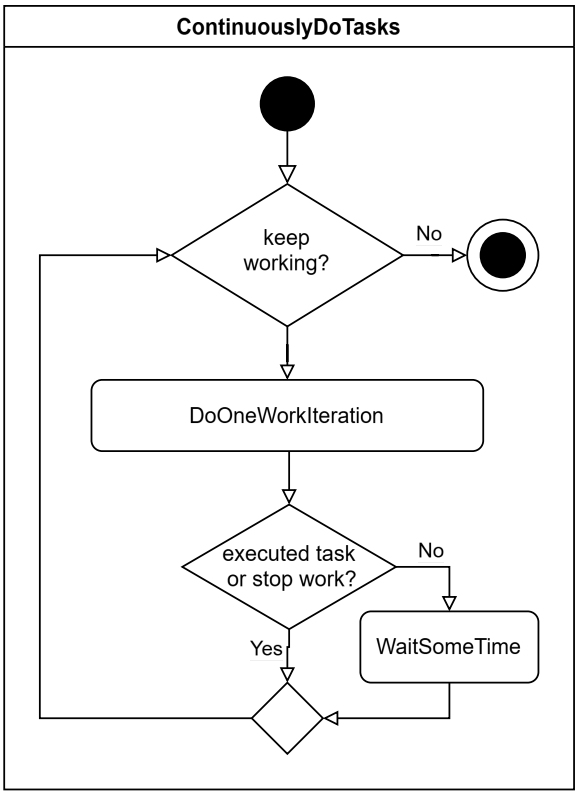

Abbildung 8.4: Aktivitätsdiagramm zur Softwarekomponente `ContinuouslyDoTasks`

8.5.5 "`DoOneWorkIteration`"

`DoOneWorkIteration` ist verantwortlich eine einzelne Iteration der Arbeiterroutine auszuführen. Eine Iteration enthält alle Schritte die notwendig sind um mit einem verteilten System aus verteilten Arbeitern das Task-Processing zu bewältigen. Darunter sind die Schritte die zur produktiven Taskausführung dienen, wie einen Task auswählen (`GetNextTaskIdIfAnyExist` siehe in 8.7) und einen Task ausführen (`ExecuteTask` siehe in 8.8).

Betrachte das folgende Aktivitätsdiagramm (Abb. 8.5). Die produktiven Schritte sind auf dieser Ebene noch abstrahiert in `GetAndExecuteOneTaskIfAnyExist`. Die `DoOneWorkIteration` hat als Verantwortung selbst die Taskwiederherstellung.

Wenn ein Arbeiter abrupt beendet wird und einen Task in Ausführung hatte, dann muss dieser Task wiederhergestellt werden, also für das System wieder als offen gelten. Als abrupt beendet gelten Events außerhalb unserer Kontrolle, wie bspw. das Beenden des Anwendungsprozess durch das Betriebssystem. Um zu erkennen ob ein Arbeiter abrupt beendet wurde, nehmen wir uns einen Heartbeat Mechanismus, wie er bspw. zur Erhaltung von Netzwerkverbindungen verwendet wird, als Vorbild [53]. Ein Heartbeat, zu deutsch Herzschlag, bedeutet im Allgemeinen ein System sendet periodisch Nachrichten, dass es noch "am leben" ist, also funktional, bleibt der Heartbeat eines Systems zu lange aus, gilt das System als "tot" und nicht mehr funktional. In unserer Implementierung richtet jeder Arbeiter einen Heartbeat für den Task den er ausführt ein. Dadurch ergibt sich im System eine Menge an Herzschlägen zu den Tasks welche derzeit vom System ausgeführt werden. Bleibt der Heartbeat eines Tasks für eine zu lange Zeit aus, gilt der Task als verlassen und muss wiederhergestellt werden. `RecoverAbandonedTasks` dient zu dieser Wiederherstellung und wird in Teil 8.5.9 genauer erläutert. Der Heartbeat wird auf der Datenbank letztendlich durch

periodische Update Statements auf eine Tabelle `TasksInProgress` umgesetzt. Dazu erstellt der Arbeiter in `CreateHeartbeatHolder` einen separaten Thread, welcher die Update Statements ausführt. Zu diesem Zeitpunkt in der Iteration führt der Arbeiter jedoch noch keinen Task aus, d.h. der Heartbeat startet noch nicht direkt. Der Arbeiter hat also einen Halter für den Heartbeat, diesem teilt er später sein Ziel, also den Task, mit. Das Ziel des Heartbeat setzen wird in der Aktivität `SetHeartbeatTarget` gemacht, welche auf dieser Ebene in `GetAndExecuteOneTaskIfAnyExist` steckt. Zum Ziel des Heartbeat setzen zählt das Erstellen des Heartbeats, durch Einfügen eines neuen Datenpunkts in die `TasksInProgress` Tabelle und das starten des Threads welcher die Update Statements ausführt. Der Arbeiter muss in allen Fällen dafür sorgen, dass der erstellte Heartbeatthread auch wieder beendet wird und der Heartbeat gelöscht. Dafür befindet sich die Komponente `GetAndExecuteOneTaskIfAnyExist` in einem try-Block und die Komponente `StopAndDeleteHeartbeat` im finally-Block. `StopAndDeleteHeartbeat` stoppt den Thread und löscht den Heartbeat. Wir verwenden zum Stoppen des Threads den Mechanismus der `keepWorkingFlag`s, diesen Mechanismus haben wir in Teil 8.5.2 bereits vorgestellt. `StopAndDeleteHeartbeat` weiß ob `SetHeartbeatTarget` ausgeführt wurde und kann die notwendigen Schritte ausführen um den Heartbeat und das Heartbeat-Ziel zu löschen.

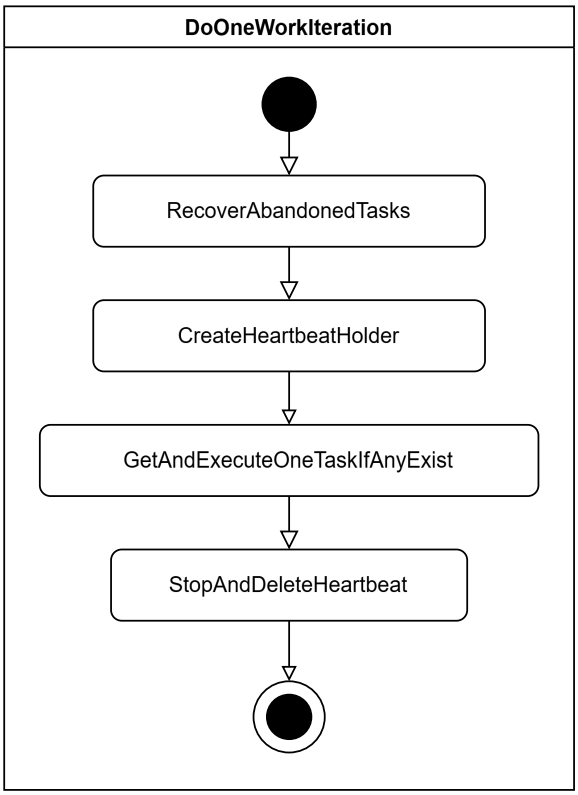

Abbildung 8.5: Aktivitätsdiagramm zur Softwarekomponente `DoOneWorkIteration`

8.5.6 "`GetAndExecuteOneTaskIfAnyExist`"

`GetAndExecuteOneTaskIfAnyExist` ist verantwortlich für das Task Holen, in gegenseitigem Ausschluss mit anderen Arbeitern und einen geholten Task auszuführen. Ist kein Task vorhanden weil der Eingabeordner leer ist, dann kann entsprechend kein Task ausgeführt werden. Betrachte das folgende Aktivitätsdiagramm (Abb. 8.6).

Die Komponenten `WaitAndAcquireGoogleDriveLock` und `FreeGoogleDriveLock` repräsentieren für uns das Betreten und Verlassen des kritischen Bereichs über das Task holen. Einen Task aus der offenen Taskmenge entfernen wollen wir nur in gegenseitigem Ausschluss anderen Arbeitern gegenüber machen, um konkurrierende Arbeiter daran zu hindern den gleichen Task auszuwählen und zu bearbeiten. Die Tätigkeit des Holens eines Tasks ist erst nach Ausführen der `FreeGoogleDriveLock` Komponente abgeschlossen, an diesem Punkt gehört dem Arbeiter der Task, der Task gilt als in Bearbeitung und der Arbeiter darf die Bearbeitung des Tasks beginnen.

Im Anschluss wird der Task in der Komponente `ExecuteTaskAndMarkAsCompletedOrFailed` ausgeführt und ebenfalls in einen der zwei terminalen Zustände gebracht, geschlossen oder fehlgeschlagen.

Die Komponente `GetAndExecuteOneTaskIfAnyExist` ist beim Task Holen dafür verantwortlich den gegenseitigen Ausschluss über Sperren des Google Drive Eingabeordners zu versichern. Wenn ein Fehler während des Task Holens auftritt sorgt hier das Exceptionhandling von der Komponente `FreeGoogleDriveLock` dafür dass bei einem Fehler der vom Sperrmechanismus ausgelöst wurde das Task holen keinen inkonsistenten Zustand hinterlässt.

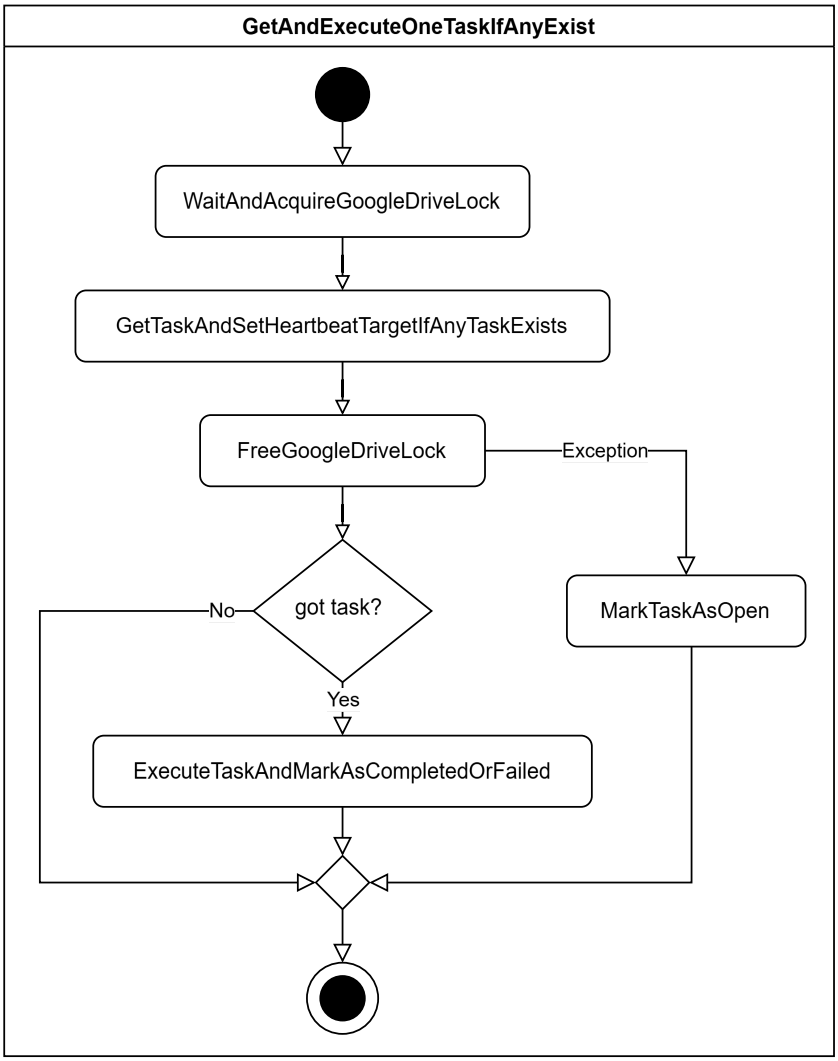

Abbildung 8.6: Aktivitätsdiagramm zur Softwarekomponente `GetAndExecuteOneTaskIfAnyExist`

8.5.7 "GetTaskAndSetHeartbeatTargetIfAnyExists"

`GetTaskAndSetHeartbeatTargetIfAnyExists` ist Teil des Task Holens. Die Komponente ist verantwortlich dafür einen Task zu finden, diesen Task als Besitz des Arbeiters und in Zustand in Bearbeitung zu bringen. Falls der Eingabeordner leer ist und dementsprechend kein Task gefunden werden kann, macht die Komponente nichts mehr. Betrachte das folgende Aktivitätsdiagramm (Abb. 8.7).

Die Komponente `GetNextTaskIdIfAnyExist` gibt uns die FileId des Tasks zurück der als nächstes zu bearbeiten ist. Da unsere Tasks derzeit noch keine Prioritäten haben ist die Auswahl aus der Menge der offenen Tasks beliebig.

Um das Task holen abzuschließen müssen wir den Task aus der Menge der offenen Tasks entfernen, also aus dem Google Drive Eingabeordner verschieben, das übernimmt die Komponente `MarkTaskAsBeingInExecution`. Ebenfalls müssen wir den Heartbeats auf den geholten Task aktivieren mit `SetHeartbeatTarget`, welche den Heartbeat erstellt und beginnt.

Wir betrachten kurz was bei Fehlern in den einzelnen Schritten passiert. Tritt ein Fehler in `GetNextTaskIdIfAnyExist` auf, haben wir noch nichts modifizert, somit keinen inkonsistenten Zustand erzeugt. Tritt ein Fehler in `SetHeartbeatTarget` auf, dann muss die Komponente `GetTaskAndSetHeartbeatTargetIfAnyExists` nichts unternehmen, da `DoOneWorkIteration` mit `StopAndDeleteHeartbeat` verantwortlich für ein korrektes Stoppen und Löschen des Heartbeats ist. Tritt ein Fehler in `MarkTaskAsBeingInExecution` auf, dann liegt die Datei des Tasks immernoch im Google Drive Eingabeordner und kann somit von anderen Arbeitern ausgewählt werden und der Heartbeat wird wie gerade erläutert ebenfalls wieder korrekt aufgeräumt.

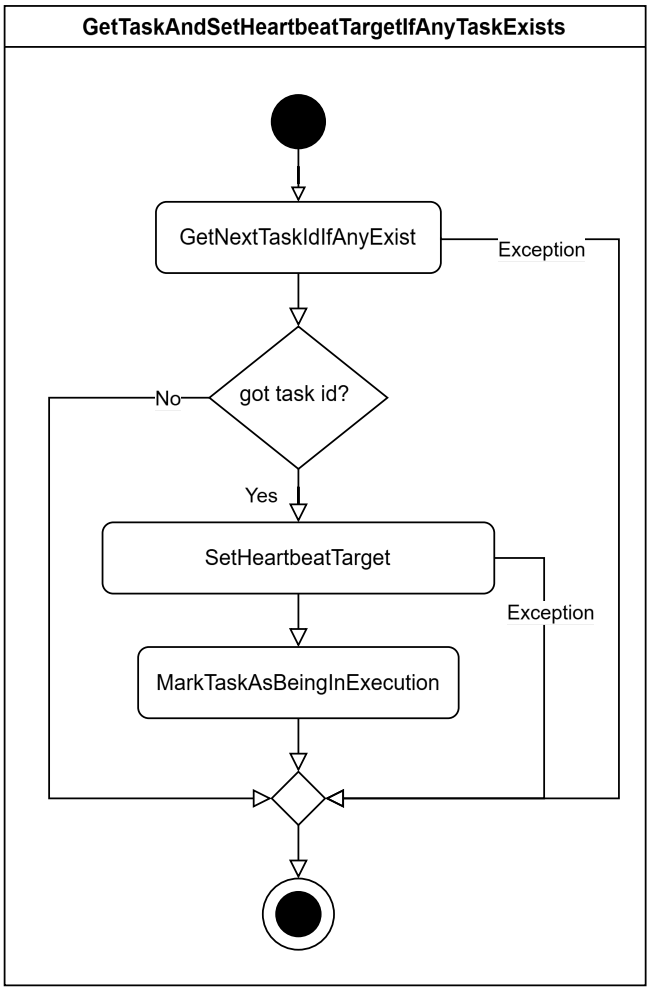

Abbildung 8.7: Aktivitätsdiagramm zur Softwarekomponente `GetTaskAndSetHeartbeatTargetIfAnyExists`

8.5.8 "`ExecuteTaskAndMarkAsCompletedOrFailed`"

`ExecuteTaskAndMarkAsCompletedOrFailed` ist verantwortlich dafür einen Task auszuführen und ihn in einen terminalen Zustand zu bringen, geschlossen oder fehlgeschlagen. Betrachte das folgende Aktivitätsdiagramm (Abb. 8.8).

Die Komponente `ExecuteTask` führt den Task aus, also die Chartanalyse und dieser Schritt kann dadurch sehr lange brauchen. Nach Abschluss der Taskausführung wird der Task mit `MarkTaskAsCompleted` als abgeschlossen markiert und der Task befindet sich dann im Zustand geschlossen.

Tritt ein Fehler während der Taskausführung oder während dem Überführen des Task-Zustand in geschlossen auf wird ein Log über den Fehler in `WriteErrorLog` persistent abgelegt. Anschließend wird der Task in `MarkTaskAsFailed` als fehlgeschlagen markiert. Der Log über den Fehler beinhaltet den Stacktrace inklusive den Nachrichten der geworfenen Exceptions und wird im konfigurierten "Failed" Google Drive Ordner neben die Eingabedatei des Tasks abgelegt. Dieser Log ist für den Nutzer und Maintainer einsehbar und dient dazu den aufgetretenen Fehler nachzuvollziehen. So wird der Nutzer auf potentielle Konfigurationsfehler aufmerksam gemacht, dieser kann dann die Konfiguration seines Tasks anpassen und dem System durch Verschieben in den Google Drive Eingabeordner erneut den Task zur Ausführung übergeben.

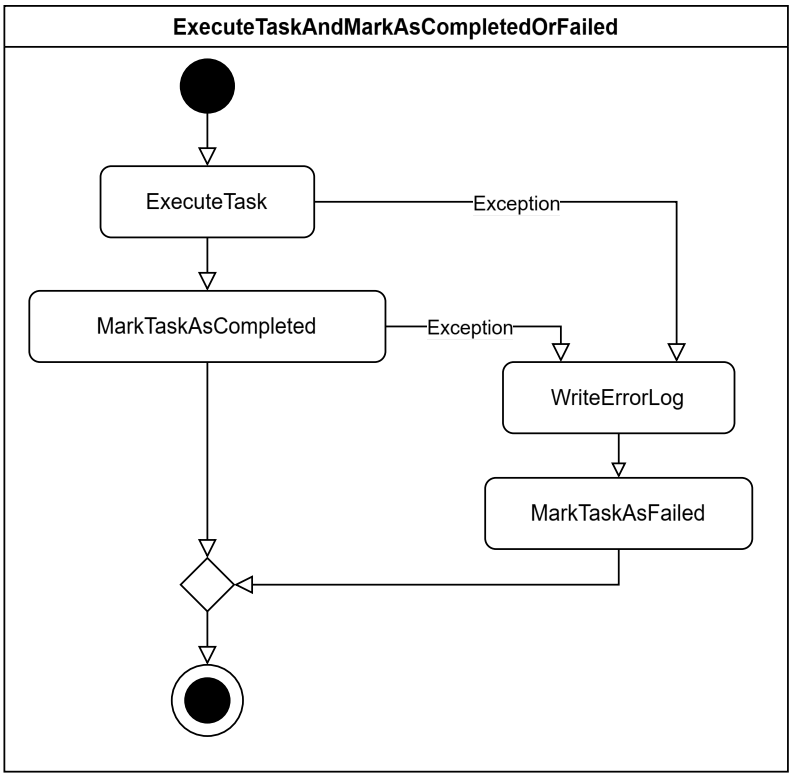

Abbildung 8.8: Aktivitätsdiagramm zur Softwarekomponente `ExecuteTaskAndMarkAsCompletedOrFailed`

8.5.9 "`RecoverAbandonedTasks`"

`RecoverAbandonedTasks` ist für das Wiederherstellen verlassener Tasks verantwortlich. Ein Task gilt als verlassen, wenn der Hearbeat zu diesem Task existiert aber der Heartbeat zu lange ausbleibt. Verlassene Tasks sollen wieder in die Menge der offenen Tasks aufgenommen werden. Verlassene Tasks entstehen, wenn ein Arbeiter während der Bearbeitung des Tasks, nach Beginn und vor erfolgreichem oder erfolglosem Abschließen des Tasks von außerhalb der Anwendung beendet wird. Zum Beispiel durch Schließen der Anwendung durch das umgebene Betriebssystem. Das folgende Aktivitätsdiagramm zeigt die notwendige Logik verlassene Tasks zu offenen Tasks zu transformieren. In einem ersten Schritt werden alle Tasks bestimmt welche als Verlassen gelten, passiert in Komponente `GetFileIdsOfAbandonedTasks`. Anschließend werden die Eingabedateien der verlassenen Tasks in den Eingabeordner verschoben und danach der Heartbeat des Tasks gelöscht. Beachte wenn ein Arbeiter wie erklärt geschlossen wird, wird auch der Thread der die Heartbeats sendet beendet, dadurch bleibt bildlich gesehen ein totes Herz zurück, als Löschen des Heartbeat wird das Aufräumen dieses Herzes gemeint. Danach gilt der Task wieder als offen und hat denselben Zustand im System wie als er initial vom Nutzer in den Eingabeordner verschoben wurde.

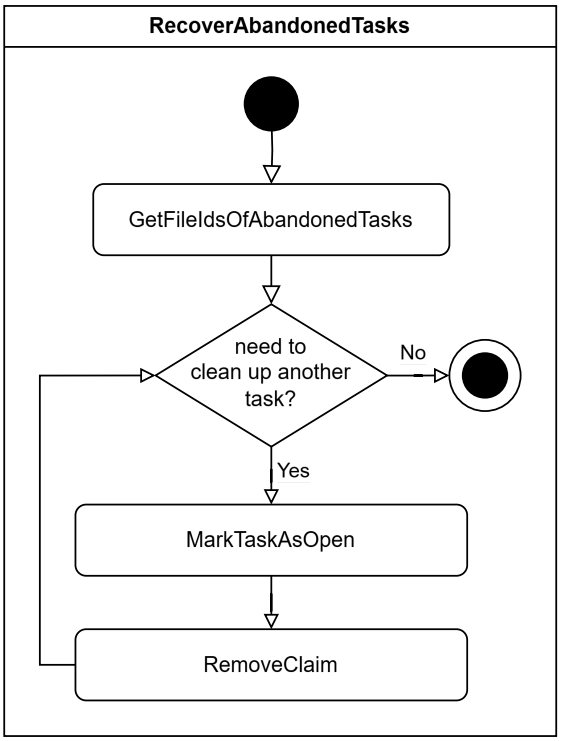

Abbildung 8.9: Aktivitätsdiagramm zur Softwarekomponente `RecoverAbandonedTasks`

`GetFileIdsOfAbandonedTasks` gibt eine Liste an FileIds für die Eingabedateien zurück, wessen Heartbeat zu lange ausblieb. Dafür filtert die Komponente die Menge aller `TasksInProgress` aus der gleichnamigen Tabelle dessen `TimeElapsedSinceLastHeartbeat` höher als ein gegebener Grenzwert ist. In folgendem Aktivitätsdiagramm ist die Logik aufgezeigt.

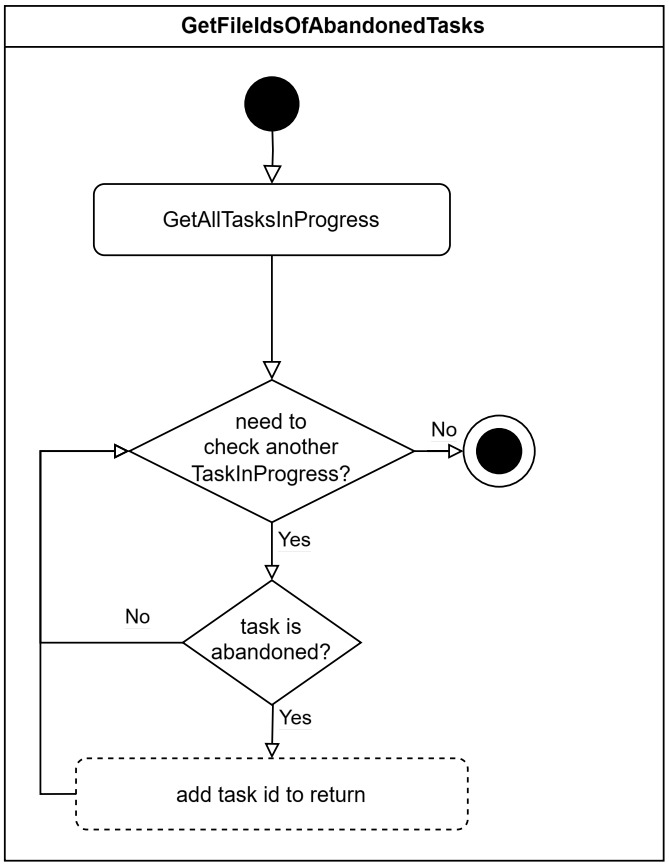

Abbildung 8.10: Aktivitätsdiagramm zur Softwarekomponente `GetFileIdsOfAbandonedTasks`

8.6 Sperren des Google Drive Eingabeordners mit relationaler Datenbank

Um zu verhindern dass zwei konkurrierende Threads den gleichen Task holen und beide den Task ausführen wollen wir das Herausnehmen eines Tasks aus der Menge der offenen Tasks und markieren des Tasks als "in Bearbeitung" nur in gegenseitigem Ausschluss erlauben. Unter Herausnehmen eines Tasks verstehen wir das Verschieben der Eingabedatei in den Executing Google Drive Ordner, so dass ein weiter Thread, der im Eingabeordner nach offenen Tasks schaut, diesen nicht mehr findet. Durch den Heartbeat-Mechanismus kann das System als Ganzes nachvollziehen, dass der Task in Bearbeitung ist und nicht mehr offen unabhängig vom Ort der Eingabedatei im Google Drive.

Die Implementierung ist mit Spring-Data-JPA, Spring-Data-JDBC (2.2.2) und PostgreSQL umgesetzt. Wir verwenden jedoch keine der Lockmodi von PostgreSQL selbst [54]. Die Locklogik ist in unserem Code umgesetzt und wird nun kurz erläutert. Ein Lock spannt sich über eine Ressource, einen Google Drive Ordner. Es kann entweder frei sein oder belegt. Ist das Lock belegt, dann hat ein Thread derzeit Besitz vom Lock ergriffen. Der Versuch ein Lock zu bekommen ist optimistisch, der erste Aufruf zur Datenbank ist ein Update Statement, welches das Lock als eigenen Besitz markiert. Das Update Statement beinhaltet in seiner Where Klausel eine Bedingung die wahr ist wenn das Lock zuvor frei war. Aus der Rückgabe erkennen wir ob das Update statement zu einer Änderung geführt hat und damit ob wir Besitz vom Lock ergreifen konnten oder nicht. Gehört das Lock nun uns, sind wir fertig. Wenn uns das Lock nicht gehört

lesen wir das Lock aus, darin enthalten ist die Id des Locks und die vergangene Zeit zu dem Zeitpunkt an dem das Lock belegt wurde. Die Id wird beim Lock nehmen erstellt und beim freigeben gelöscht und wiederholt sich nie. Diese zwei Schritte wiederholen wir bis uns das Lock gehört oder die vergangene Zeit einer einzelnen Lock-Id einen Grenzwert überschreitet. Ist der Grenzwert überschritten versuchen wir das Lock der Id, die abgelaufen ist, zu übernehmen. Wir setzen dafür ein Update Statement ab, welches in der Where Klausel eine Bedingung hat die wahr ist, wenn das Lock zuvor die abgelaufene Id enthielt. Dieses Update Statement kann fehlschlagen, wenn ein anderer Thread das Lock bereits übernommen hat. In diesem Fall fangen wir die Logik von vorne an. Wenn wir das Lock übernehmen konnten sind wir fertig. Der Mechanismus Besitz des Locks zu ergreifen bevor der Besitzer es freigibt ist für Fälle in denen ein Thread, während er das Lock besitzt, terminiert oder für lange Zeit sein Programm nicht fortführen kann. Dadurch vermeiden wir das Blockieren des Systems durch so einen Thread. Ein Lock hat also ein Gültigkeitszeitraum, ist dieser Abgelaufen hat der Thread kein Anrecht mehr auf das Lock und andere Threads können es übernehmen. Dieser Mechanismus ähnelt traditionelleren Lockimplementierungen welche oftmals auf ihren Transaktionen Timeouts haben um sich vor Ressourcenverschwendung und langem Blockieren von Ressourcen anderer Teilnehmer auf Grund von festgefahrenen Threads schützen.

Das Freigeben des Locks markiert das Lock als frei. Dabei wird die eigene Lock-Id zurückgesetzt.

Die Implementierung mit der relationalen Datenbank sieht wie folgt aus. Wir haben eine Locktabelle genannt `locks`. Ein Eintrag in der Locktabelle ist ein Lock für einen Eingabeordner. Die Implementierung unterstützt derzeit nur einen einzelnen Eingabeordner, deshalb wird die Tabelle mit einem Eintrag initialisiert.

Die Tabelle hat Spalten, `resourceId : String`, `lockId : String` und `locked_at : DateAndTime`.

`resourceId` ist der Identifikator für die Ressource die gesperrt werden kann, hier also der Identifikator unseres Eingabeordners.

`lockId` ist der Identifikator für die Besitzinstanz des Locks. Auf logischer Ebene wird die Besitzinstanz beim Nehmen des Locks erstellt und beim Freigeben des Locks gelöscht. Besitzt niemand das Lock, ist die Id ein leerer String und signalisiert damit dass das Lock frei ist. Ein Thread generiert eine `lockId`, diese muss unabhängig des Threads und Zeit ein einmaliger Wert sein. Umgesetzt durch das Verwenden von UUIDs. Diese merkt sich der Thread bis er das Lock wieder freigibt. Anhand der `lockId` kann jeder Thread im System sehen, ob das Lock frei ist, ob es einem anderen Thread gehört oder ob es dem Thread selbst gehört.

`locked_at` ist der Zeitstempel zu welchem sich ein Thread das Lock geholt hat. Fragt man den Status eines Locks ab, also einer Tabellenzeile, dann kann man anhand der vergangenen Zeit zum `locked_at` Zeitstempel sehen wie lange diese Instanz des Locks bereits gehalten wird.

Ist diese vergangene Zeit zu hoch, dann kann man davon ausgehen, dass der Lockhalter auf Grund eines Fehlers das Lock nicht mehr freigeben kann. In dem Fall, dass die vergangene Zeit zu hoch ist, darf ein anderer Thread das Lock, obwohl es nicht frei ist, übernehmen. Dabei muss der Thread aufpassen, dass in der Zwischenzeit nicht ein dritter Thread das Lock bereits übernommen hat. Bei der gezwungenen Übernahme des Locks, wechselt die Besitzinstanz des vorherigen Halters direkt zur eigenen Besitzinstanz. Die gezwungene Übernahme ist eine Mechanismus, so dass im gerade benannten Fehlerfall das System nicht unendlich lange blockiert ist. Die vergangene Zeit ist zu hoch wenn sie größer als ein konfigurierter Millisekundenwert ist. Diesen Wert sollte man so wählen, dass ein Lockhalter genug Zeit hat einen Task zu suchen, diesen im Google Drive zu verschieben und als “in Bearbeitung” zu markieren.

8.6.1 Heartbeat-Mechanismus mit relationaler Datenbank

In der Tabelle `TasksInProgress` haben wir zu jedem Task der derzeit in Bearbeitung ist einen Eintrag. Ähnlich zum Attribut der vergangenen Zeit seit Lockakquisition der `locks` Tabelle hat diese Tabelle eine Spalte mit vergangener Zeit seit der letzten Modifikation. Hier wollen wir durch die Verbindung eines Timeouts mit einem Hearbeat Mechanismus die Ausfälle einzelner Threads erkennen. Ein Thread während er im kritischen Bereich vom Holen eines Tasks ist, erstellt einen Eintrag mit einem Identifikator des Tasks in der `TasksInProgress` Tabelle. Während Bearbeitung des Tasks modifiziert der Thread den Datenbankeintrag in regulären Abständen, wie ein Hearbeat. Fällt der Thread aus wird der Datenbankeintrag nicht mehr aktualisiert. Ein anderer Thread schaut in dieser Tabelle nach ob der Heartbeat für einen Tasks zu lange ausgeblieben ist. Ist das der Fall verschiebt der Thread die Eingabedatei des Tasks zurück in den Eingabeordner und löscht danach den entsprechenden Datenbankeintrag aus der `TasksInProgress` Tabelle. Danach setzt er sein Tun wie gewohnt fort.

8.7 Highlights

8.7.1 Implementierung in Softwarekomponenten abstrahiert

Durch das Aufteilen der einzelnen Verantwortlichkeiten in Softwarekomponenten weiß eine einzelne Komponente garnicht wie eine andere Komponente ihre Logik ausführt. Durch dieses Verstecken der Implementierung von Teillogik kann man die Implementierung einer Komponente leicht austauschen.

Nehmen wir bspw. die Komponente `RunGivenAmountOfWorkers`, in Teil 8.5.2 erläutert, diese ist derzeit mit einer While-Schleife und vereinfacht ausgedrückt mit einem `Thread.start()` implementiert. Die Implementierung dieser Komponente könnte man nun austauschen gegen einen Ansatz welcher einen `ThreadPoolExecutor` (2.2.4) nutzt. Der Rest der Anwendung ist von solch einer Änderung nicht betroffen.

Am wichtigsten für die Verwendung des Projekts ist es, dass die Taskausführung mit `ExecuteTask` sauber gekapselt wurde. Das Task-Processing Projekt soll nicht auf die Ausführung bestimmter Algorithmen, wie die unserer Chartanalysen, limitiert sein. Das Task-Processing Konzept der Arbeit benötigt lediglich einen Identifikator für einen Task. Hier nutzen wir die FileId von Google, da unsere Tasks in Google Docs Dateien konfiguriert sind. Die `ExecuteTask` Komponente bekommt diesen Identifikator als Parameter und wie diese Komponente implementiert ist, spielt für das Task-Processing keine Rolle. So kann die `ExecuteTask` Komponente problemlos ausgetauscht werden zu Code welcher den Identifikator für einen Task als Eingabe bekommt und weiß wie er den Task ausführen muss.

8.8 Shortcomings

8.8.1 Vereinen des Lock-Mechanismus und Heartbeat-Mechanismus

Ein Shortcoming der Implementierung ist, dass der Lock-Mechanismus und Heartbeat-Mechanismus sich sehr ähneln, jedoch unterschiedlich implementiert sind. Die Grundidee der Mechanismen ist die Gleiche, ein Lock auf eine Ressource, welche nur von einem Arbeiter gehalten werden darf. Die Ressourcen der Locks sind die Google Drive Eingabeordner, Ressourcen des Heartbeats die Tasks. Identifier für die Ressource ist bei beiden der Primärschlüssel der Tabelle. Beide Mechanismen nutzen einen Identifier wodurch der Besitzer prüfen kann ob er die Ressource hält. Bei Locks in Form der Lock-Id und bei Heartbeat in Form der creation-Id. Der Timeout-Aspekt der den Besitz nach zu langer inaktiver Zeit ungültig macht ist bei beiden ebenfalls derselbe.

8.8.2 Task ist offen genau dann wenn er im Google Drive Eingabeordner liegt

"Premature optimization is the root of all evil." Dieses Zitat von Donald E. Knuth [55] beschreibt das Phänomen, dass Softwareentwickler oftmals zu eilig sind vermeintliche Schwachstellen im Code zu verbessern bevor sie diese Schwachstellen wirklich identifizieren konnten. Der Arbeit ist dieser Fehler unterlaufen. In der Entscheidungsfindung wurde es damals für notwendig gefunden keine extra Datenstruktur für das halten aller offenen Tasks zu pflegen sondern dafür direkt den Google Drive Eingabeordner zu verwenden. Gemeint mit Google Drive Eingabeordner verwenden, ist wie in der Überschrift geschrieben, das System einen Task als offen wertet, genau dann wenn er im Google Drive Eingabeordner liegt. In Sektion 7.4 haben wir schon kurz angemerkt, dass diese Abhängigkeit nicht gut ist und wollen das kurz erklären.

Unsere Implementierung sollte nicht von spezifischen Technologien wie hier Google Drive abhängen. Solche Abhängigkeiten sind in Zukunft schwieriger aufzulösen, falls andere Technologien verwendet werden sollen. Wie in Sektion 8.7.1 bereits angemerkt haben wir durch unsere Softwarekomponenten, die Implementierungsdetails für die Google Drive implementierung bereits vor der Businesslogik abstrahiert.

Ein anderer Aspekt dieser Abhängigkeit, ist dass der Arbeitsschritt für einen Arbeiter um sich einen Task zu holen recht umfangreich ist. Ein Arbeiter muss sich eine Sperre auf den Google Drive Eingabeordner holen, dann eine Eingabedatei auswählen und diese im Google Drive verschieben. Danach gibt der Arbeiter diese Sperre wieder frei. In Lösungsalternative drei, wie in Sektion 7.4 erläutert reduziert diese Lösungsalternative den Umfang auf das Modifizieren eines einzelnen Datensatzes in unsrer relationalen Datenbank.

8.8.3 Doppelte Ausführungen sind doch möglich

Unsere Implementierung kann leider nicht mit Garantie doppelte Ausführungen verhindern. Das Fehlerszenario tritt dann auf, wenn ein Arbeiterthread eine lange Zeit pausiert und dann fortsetzt, ein Phänomen welches in verteilten Systemen auftreten kann [56, Seite 310]. Das inhärente Problem ist, dass wir einen Arbeiter als tot erklären, weil ein gewisser Timeout überschritten wurde. Die Probleme solcher Teilausfälle in verteilten Systemen zu lösen ist komplex [56, Seite 311]. Die Auswirkungen sind, pausiert ein Arbeiter länger als die implementierten Timeouts, dann kann ein Task doppelt ausgeführt werden. Diese Auswirkungen werden von der Arbeit vorerst als annehmbar befunden.

Wir erläutern kurz wann dieses Szenario auftritt. Arbeiter 1 holt die Sperre auf den Google Drive Eingabeordner, wird lange Zeit nicht vom Betriebssystem geschedult, Arbeiter 2 übernimmt die Sperre von Arbeiter 1 da ihm die Zeit ausgelaufen ist, dann sind beide im kritischen Bereich und können dementsprechend denselben Task auswählen. Ohne den Heartbeat-Mechanismus würde das nicht auffallen. Versuchen beide Arbeiter ihren Heartbeat auf denselben Task zu schalten, schlägt es für den zweiten fehl. So ist in diesem Szenario die doppelte Ausführung des Tasks verhindert. Der Heartbeat-Mechanismus verhindert doppelte Ausführungen aber auch nicht immer. Setzt der Heartbeat für eine zu lange Zeit aus, weil der Thread lange pausiert wird, dann gilt der Task als verlassen und ein anderer kann die Bearbeitung wieder starten. Der pausierte Thread könnte nun fortsetzen und beide bearbeiten dann den gleichen Task.

Kapitel 9

Ergebnis

9.1 Bewertung

In diesem Teil wollen wir unseren Ansatz und Lösung bewerten. Wir wollen betrachten ob wir den Misstand beheben konnten und in wie fern wir unser gestelltes Ziel erreicht haben. Außerdem wird betrachtet ob und wie einfach das entwickelte Projekt auch für andere Bereiche verwendet werden kann.

Den Misstand der in Teil 1.2 vorgestellt wurde beschreibt die Probleme des TA eine wirtschaftliche Handelsstrategie zu entwickeln. Das Overfitprojekt welches gute Algorithmen für Chartanalysen enthält ist nicht geeignet nutzbar, da der TA viele langlaufende Chartanalysen am Tag ausführen möchte.

Die Nutzbarkeit haben wir deutlich verbessert, der TA kann seine Chartanalyse in einem Google Doc konfigurieren und sie anschließend dem System übergeben, durch das einfache Verschieben der Google Doc Datei in einen konfigurierten Google Drive Eingabeordner. Der TA kann außerdem beliebig viele solche Konfigurationsdateien für Chartanalysen dem System ablegen und dieses führt diese mit den vorhandenen Ressourcen aus.

Die benötigte Zeit all die Chartanalysen die der TA benötigt auszuführen ist durch das parallele Ausführen stark gesunken. Durch die Fehlertoleranz Maßnahmen der Heartbeats auf Tasks die in Bearbeitung sind und den Aufräumjob stellt das System sicher dass kein Task verloren geht.

Haben wir mit der Arbeit unser in Sektion 1.3 definiertes Ziel erreicht und eine zufriedenstellende Lösung erstellen können?

Unser Ziel haben wir als “Infrastruktur-minimales paralleles Backend für Offline-Chartanalysen” definiert.

Das Deployment der Lösung umfasst eine uns gegebene relationale Datenbank und replizierte Arbeiterapplikationen. Die Arbeiterapplikationen arbeiten unabhängig voneinander und haben als einzige Abhängigkeit unseres Systems die Datenbank. In unserem Entscheidungsprozess haben wir die infrastrukturminimale Anforderung an die Lösung stets beachtet und diesen Punkt voll erfüllt. Ebenfalls durch Vermeiden von doppelten Ausführungen verschwendet das System kaum Ressourcen in seiner Arbeit.

Unsere Lösung, ein kontinuierlich laufendes Backend führt parallel Chartanalysen aus, dabei ist keinerlei Interaktion mit dem Backend selbst notwendig. Ein Nutzer legt die Konfiguration für die Chartanalyse im Google Drive Eingabeordner ab und das Backend führt diese anschließend aus und macht das Ergebnis dem Nutzer verfügbar. Wir haben also ein paralleles Backend für Offline-Chartanalysen erstellt und sind damit dem Ziel auch in diesem Punkt gerecht.

Das Ergebnis der Arbeit ist eine erfolgreiche Recherche wie das Problem der Arbeit konzeptionell gelöst werden kann und eine Implementierung, welche zwar in in Sektion 8.8 erwähnte Schwachstellen

aufweißt, aber eine gute Basis zur Weiterentwicklung darstellt. Die Implementierung erreicht eine Lösung unserer Problematik.

9.2 Ausblick

Hier werden weitere Features, Verbesserungen des Projekts diskutiert.

Generell ein Ausblick auf die Zukunft des Projekts geben uns die Shortcomings aus Kapitel 8.8. Im Bewertungskapitel zur dritten Lösungsalternative in Sektion 7.4, so wie im Shortcoming Kapitel wiederholt, haben wir angedeutet dass wir möglicherweise auch in die Richtung der dritten Lösungsalternative gehen wollen.

9.2.1 Batch-Scheduling und Work-Stealing

Unsere Implementierung könnte mit zusätzlichem Batch Scheduling und Work Stealing erweitert werden. Eine Notwendigkeit hierfür wäre, wenn die Threads sehr lange warten müssten bis sie das Lock und damit einen neuen Task bekommen. Das kann auftreten, wenn die Tasks, gegen die Erwartung, eine geringe Dauer haben. Dann muss ein Arbeiter sich öfter einen Task holen und dementsprechend kann es sich stauen. Durch Batch-Scheduling würde sich ein Thread nicht einen sondern mehrere Tasks auf einmal nehmen. So hat ein Thread mehrere Tasks gleichzeitig in Bearbeitung. Der Thread hat dann eine lokale Warteschlange mit Tasks. Der Thread arbeitet dann zuerst seine Warteschlange komplett ab, bevor er nach neuen Tasks sucht. Das Heartbeat Feature müsste etwas überdacht werden, so dass ein Arbeiter einen Herzschlag an mehrere Tasks zeitgleich senden kann. Ob das Batch-Scheduling lohnenswert ist muss abgewägt werden.

Ein Nachteil der Entstehen kann ist, dass ein Arbeiter noch Tasks in der Warteschlange hat während ein anderer Arbeiter nichts zu tun hat. Dieses Problem könnte man durch Implementieren von Work Stealing beheben. Spätestens hier stellt sich dann aber die Frage ob der implementierte Ansatz noch der Richtige ist.

9.2.2 Mehrstufige Parallelisierung

Manche Tasks haben vielleicht die Möglichkeit nochmals in weitere parallele Tasks aufgeteilt zu werden. Chartanalysen wie bspw. die Parameteroptimierung aus dem Overfitprojekt könnten durchaus mit dem Fork und Join Modell (3.1.2) wie in 3.5.2 angemerkt nochmals weiter parallelisiert werden. So könnten diese Art von Tasks noch weiter beschleunigt werden. So einen Mechanismus in das Task-Processing einzubauen ist vermutlich sehr komplex, da ein einzelner Task nun mehr als nur einen Prozessorkern benötigt um eine Beschleunigung zu erreichen. Es wurden sich über die Umsetzung des Features keine weiteren Gedanken gemacht.

9.2.3 Auslesbarer Task-Fortschritt

Da Chartanalysen teilweise sehr lange dauern, möchte der Nutzer den Fortschritt einsehen um einschätzen zu können wann er das Ergebnis auslesen kann.

Den Fortschritt einer Chartanalyse zu einer gegebenen Zeit kann nur ihr eigener Algorithmus bestimmen. Der Backtestalgorithmus des Overfitprojekts durchläuft die Datenmenge anhand zeitlicher Schritte von vorne nach hinten durch. Für diesen Backtest kann man den Fortschritt dann anhand dem Verhältnis zwischen abgearbeiteten Zeitschritten und gesamten Zeitschritten ablesen. Der Backtest weiß im Vorhinein zwar nicht wie viele Operationen er pro Zeitschritt ausführen muss, aber durch die hohe Anzahl an Zeitschritten bei lang laufenden Backtests sollte sich das normalisieren.

Eine grundlegende Idee, wäre es der `ExecuteTask` Komponente eine neue Klasse `TaskProgress` als Parameter mitzugeben. Die `TaskProgress` Klasse hat dann eine `update(..)` Methode mit welcher der Code in `ExecuteTask` dann selbstständig von seinem Fortschritt berichten kann. Es wurden sich jedoch keine weiterführenden tieferen Gedanken zur Umsetzung des Features gemacht.

9.2.4 Tasks können abgebrochen werden

Der Nutzer möchte Chartanalysen die in Bearbeitung sind abbrechen können. Damit der Nutzer Ressourcen im System freimachen kann, so dass das System neue, für den Nutzer wichtigere, Chartanalysen früher ausführen kann.

Da die Chartanalysen teilweise sehr lange dauern kann der zeitliche Gewinn durch Abbrechen einer Chartanalyse sehr hoch sein. Einige Aspekte fallen zur Umsetzung des Features direkt ein. Wie sieht dieses Feature für den Nutzer aus, wie kann er es bedienen. Dazu wurden sich keine weiteren Gedanken gemacht.

Von der Logik bietet es sich an die Tabelle `TasksInProgress` für ein Abbrechen-Signal zu verwenden, da diese ja genau die Tasks in Bearbeitung enthält. Durch hinzufügen einer neuen Spalte für eine Flag könnte man das Abbrechen des Task signalisieren. Ähnlich zum Feature der `keepWorkingFlag`s muss die Implementierung eines Tasks, die Flag in der Datenbank an geeigneten Zeitpunkten überprüfen und dementsprechend ihre Arbeit abbrechen.

Glossar

Dependency Injection Dependency Injection ist eine Methode der... 9, 10

IoC Container Inversion of Control Container. Komponente von Spring Framework in welchem instanzierte Beans nach Prinzip der Dependency Inversion gehalten werden.. 9, 10

Master-Worker Ein Master Thread der Arbeit auf seine mehreren Arbeiterthreads verteilt.. 21

Multitasking Definition in Anlehnung an Wikipedia, Artikel 'Multitasking'. Multitasking bezeichnet die Fähigkeit eines Betriebssystems, mehrere Aufgaben (Tasks) (quasi-)nebenläufig auszuführen.. 14

Property Values Name von Spring, beschreibt Werte zur Konfiguration einer Anwendung. Darunter bspw. Kommandozeilenparameter.. 10, 49

`Runnable` Runnable ist ein Datentyp aus Java. Er stellt eine anonyme parameterlose rückgabelose Methode dar. Mit `Runnable.run()` wird diese Methode ausgeführt.. 12

Servicemodelle Cloud Servicemodelle wie IaaS (Infrastructure as a Service), PaaS(Platform as a Service), SaaS(Software as a Service) etc.. 18

Single Point of Failure Ein Single Point of Failure ist ein Bestandteil eines Systems, welcher bei dessen Ausfall den Ausfall des gesamten Systems mit sich bringt.. 21

Thread Definition in Anlehnung an Wikipedia, Artikel 'Thread (Informatik)'. Ein Thread ist ein Ausführungsstrang oder eine Ausführungsreihenfolge in der Abarbeitung eines Programms. Ein Thread ist Teil eines Prozesses.. 16

YAML Definition in Anlehnung an RedHat, Artikel 'Was ist YAML?'. YAML Ain't Markup Language ist eine von Menschen lesbare Sprache zur Serialisierung von Daten, die häufig zum Schreiben von Konfigurationsdateien verwendet wird.. 10, 50

zustandslos Ist eine Eigenschaft einer Komponente im Deployment, welche keinen Zustand hat, keine Daten die persisitert werden müssen.. 33

Literatur

[1] M. J. J., *Technische Analyse der Finanzmärkte*, de. Munich, Germany: FinanzBuch Verlag, Mai 2006.

[2] Google Finance, *NASDAQ: AMZN*, https://g.co/finance/AMZN:NASDAQ?window=1M, Publisher: Google, Accessed on: 2025-01-22, 2025.

[3] Spring by VMware Tanzu, *Spring Boot*, https://spring.io/projects/spring-boot, Publisher: Spring by VMware Tanzu, Accessed on: 2025-04-04, 2025.

[4] Spring by VMware Tanzu, *Externalized Configuration :: Spring Boot*, https://docs.spring.io/spring-boot/reference/features/external-config.html, Publisher: Spring by VMware Tanzu, Accessed on: 2025-03-19, 2025.

[5] Spring by VMware Tanzu, *Spring Data*, https://spring.io/projects/spring-data, Publisher: Spring by VMware Tanzu, Accessed on: 2025-03-19, 2025.

[6] Spring by VMware Tanzu, *Spring Data JPA*, https://spring.io/projects/spring-data-jpa, Publisher: Spring by VMware Tanzu, Accessed on: 2025-03-19, 2025.

[7] Spring by VMware Tanzu, *Spring Data JDBC*, https://spring.io/projects/spring-data-jdbc, Publisher: Spring by VMware Tanzu, Accessed on: 2025-03-19, 2025.

[8] Google, *Einfacher Zugriff auf Google APIs über Java*, https://developers.google.com/api-client-library/java?hl=de, Publisher: Google, Accessed on: 2025-02-26, 2025.

[9] Google, *Files and folders overview :: File characteristics*, https://developers.google.com/workspace/drive/api/guides/about-files#characteristics, Publisher: Google, Accessed on: 2025-04-04, 2025.

[10] Google, *Google Drive-Lösungen entwickeln*, https://developers.google.com/drive?hl=de, Publisher: Google, Accessed on: 2025-02-27, 2025.

[11] Google, *Nach Dateien und Ordnern suchen*, https://developers.google.com/drive/api/guides/search-files?hl=de, Publisher: Google, Accessed on: 2025-02-27, 2025.

[12] Google, *Dateien zwischen Ordnern verschieben*, https://developers.google.com/drive/api/guides/folder?hl=de#move-files, Publisher: Google, Accessed on: 2025-02-27, 2025.

[13] Google, *Text mit der Docs API aus einem Dokument extrahieren*, https://developers.google.com/docs/api/samples/extract-text?hl=de, Publisher: Google, Accessed on: 2025-02-27, 2025.

[14] Google, *Struktur eines Google Docs-Dokuments*, https://developers.google.com/workspace/docs/api/concepts/structure?hl=de, Publisher: Google, Accessed on: 2025-04-3, 2025.

[15] Google, *Text mit der Docs API aus einem Dokument extrahieren*, https://developers.google.com/workspace/docs/api/samples/extract-text?hl=de, Publisher: Google, Accessed on: 2025-04-3, 2025.

[16] Google, *Tabelle Erstellen*, https://developers.google.com/sheets/api/guides/create?hl=de, Publisher: Google, Accessed on: 2025-02-27, 2025.

[17] Google, *Zellenwerte lesen und schreiben*, https://developers.google.com/sheets/api/guides/values?hl=de, Publisher: Google, Accessed on: 2025-02-27, 2025.

[18] RedHat, *Was ist ein Webhook?* https://www.redhat.com/de/topics/automation-and-management/was-ist-ein-webhook, Publisher: RedHat, Accessed on: 2025-04-04, 2024.

[19] Google, *Notifications for resource changes*, https://developers.google.com/drive/api/guides/push#making-watch-requests, Publisher: Google, Accessed on: 2025-02-19, 2025.

[20] Oracle Autoren, *The Java Tutorials :: Interrupts*, https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html, Publisher: Oracle, Accessed on: 2025-04-03, 2025.

[21] Oracle Autoren, *Package java.util.concurrent.atomic*, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html, Publisher: Oracle, Accessed on: 2025-04-03, 2025.

[22] Oracle Autoren, *C JDBC Coding Tips :: C.1 JDBC and Multithreading*, https://docs.oracle.com/en/database/oracle/oracle-database/21/jjdbc/JDBC-coding-tips.html#GUID-EE479007-D105-4F82-8D51-000CBBD4BC77, Publisher: Oracle, Accessed on: 2025-04-03, 2025.

[23] Oracle Autoren, *Interface Executor*, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html, Publisher: Oracle, Accessed on: 2025-04-02.

[24] Apache, *Flexible & Powerful Open Source Multi-Protocol Messaging*, https://activemq.apache.org/, Publisher: Apache, Accessed on: 2025-04-04, 2025.

[25] J. Reock, *What Is Apache ActiveMQ?* https://www.openlogic.com/blog/what-apache-activemq#what-is-activemq-01, Publisher: OpenLogic by Perforce, Accessed on: 2025-04-04, 2020.

[26] The PostgreSQL Global Development Group, *What is PostgreSQL?* https://www.postgresql.org/about/, Publisher: The PostgreSQL Global Development Group, Accessed on: 2025-04-09, 2025.

[27] Wikipedia-Autoren, *Nebenlaeufigkeit*, https://de.wikipedia.org/wiki/Nebenlufigkeit, Publisher: Wikipedia, Accessed on: 2025-01-31, 2025.

[28] Daniels220 at English Wikipedia, *File:AmdahlsLaw.svg*, https://en.wikipedia.org/wiki/Heartbeat_(computing), Publisher: Wikipedia, Accessed on: 2025-04-11, 2008.

[29] D. F. H. M. W. Y, „State of the Art in Parallel and Distributed Systems: Emerging Trends and Challenges,“ Feb. 2025. DOI: https://doi.org/10.3390/electronics14040677.

[30] M. A. Barry Wilkinson, „Parallel Programming Techniques And Applications Using Networked Workstations And Parallel Computers 2nd Edition,“ S. 98–205, Mai 2004.

[31] Microsoft Autoren, *Hochskalieren und Aufskalieren im Vergleich*, https://cloud.google.com/discover/types-of-cloud-computing?hl=de, Publisher: Microsoft, Accessed on: 2025-04-04, 2025.

[32] Google, *Welche Arten von Cloud-Computing gibt es?* https://cloud.google.com/discover/types-of-cloud-computing?hl=de, Publisher: Google, Accessed on: 2025-03-19, 2025.

[33] P. Biswas, *Understanding GPUs: Task Parallelism vs. Data Parallelism*, https://www.linkedin.com/pulse/understanding-gpus-task-parallelism-vs-data-prasanna-biswas-kyxgc/, Publisher: LinkedIn Corporation, Accessed on: 2025-04-04, 2025.

[34] S. Ghosh, *Distributed systems* (Chapman & Hall/CRC Computer & Information Science Series), en, 2. Aufl. Philadelphia, PA: Chapman & Hall/CRC, Sep. 2020.

[35] E. Jafarnejad Ghomi, A. Masoud Rahmani und N. Nasih Qader, „Load-balancing algorithms in cloud computing: A survey,“ en, *J. Netw. Comput. Appl.*, Jg. 88, S. 50–71, Juni 2017.

[36] N. Devi, S. Dalal, K. Solanki u. a., „A systematic literature review for load balancing and task scheduling techniques in cloud computing,“ en, *Artif. Intell. Rev.*, Jg. 57, Nr. 10, S. 11–21, Sep. 2024. DOI: https://doi.org/10.1007/s10462-024-10925-w.

[37] A. Alakeel, „A Guide to Dynamic Load Balancing in Distributed Computer Systems,“ *International Journal of Computer Science and Network Security (IJCSNS)*, Jg. 10, S. 155, Nov. 2009.

[38] J. B. Fernandes, Í. A. S. de Assis, I. S. Martins, T. Barros und S. Xavier-de-Souza, *Adaptive Asynchronous Work-Stealing for distributedload-balancing in heterogeneous systems*, Publisher: arXiv, Accessed on: 2025-04-09, 2024. arXiv: 2401.04494 [cs.DC]. Adresse: https://arxiv.org/abs/2401.04494.

[39] S. K. Prasad, A. Gupta, A. Rosenberg, A. Sussman und C. Weems, Hrsg., *Topics in parallel and distributed computing*, en, 1. Aufl. Cham, Switzerland: Springer International Publishing, Okt. 2018.

[40] M. Ajtai, J. Aspnes, M. Naor, Y. Rabani, L. J. Schulman und O. Waarts, „Fairness in scheduling,“ *Journal of Algorithms*, Jg. 29, Nr. 2, S. 306–357, Nov. 1998.

[41] R. D. Blumofe und C. E. Leiserson, „Scheduling multithreaded computations by work stealing,“ en, *J. ACM*, Jg. 46, Nr. 5, S. 720–748, Sep. 1999.

[42] U. A. Acar, A. Charguéraud und M. Rainey, *Greedy Sharing: Load Balancing on Weakly Consistent Memory*, https://www.charguéraud.org/research/2012/syncfree/greedy.pdf, Publisher: Arthur Charguéraud, Accessed on: 2025-04-01, 2012.

[43] Datenbanken-verstehen.de Autoren, *Sperrverfahren in Datenbanken*, https://www.datenbanken-verstehen.de/lexikon/sperrverfahren-datenbanken/, Publisher: Datenbanken-verstehen.de, Accessed on: 2025-04-04.

[44] J. Hannan, „Konkurrierende Zugriffe,“ in *Ein praktischer Führer für das Datenbank-Management*, J. Hannan, Hrsg. Wiesbaden: Vieweg+Teubner Verlag, 1988, S. 77–92, ISBN: 978-3-322-88831-0. DOI: 10.1007/978-3-322-88831-0_7. Adresse: https://doi.org/10.1007/978-3-322-88831-0_7.

[45] A. Kemper und A. Eickler, *Datenbanksysteme* (de Gruyter Studium), de, 10. Aufl. de Gruyter, Feb. 2018.

[46] LinkedIn community, *What are some common distributed locking patterns and anti-patterns that you have encountered or used?* https://www.linkedin.com/advice/0/what-some-common-distributed-locking-patterns?lang=de&originalSubdomain=de, Publisher: LinkedIn, Accessed on: 2025-04-04.

[47] Wikipedia-Autoren, *Nebenlaeufigkeit*, https://en.wikipedia.org/wiki/Heartbeat_(computing), Publisher: Wikipedia, Accessed on: 2025-04-10.

[48] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes und D. Lea, *Java Concurrency in Practice*. Addison-Wesley, Mai 2006, ISBN-10: 0321349601 ISBN-13: 978-0321349606.

[49] Google, *Nutzungsbeschränkungen*, https://developers.google.com/workspace/docs/api/limits?hl=de, Publisher: Google, Accessed on: 2025-04-09, 2025.

[50] Apache, *AcknowledgementMode Enumeration*, https://activemq.apache.org/components/nms/msdoc/1.6.0/vs2005/Output/html/T_Apache_NMS_AcknowledgementMode.htm, Publisher: Apache, Accessed on: 2025-04-08, 2025.

[51] The PostgreSQL Global Development Group, *13.2. Transaction Isolation*, https://www.postgresql.org/docs/current/transaction-iso.html, Publisher: The PostgreSQL Global Development Group, Accessed on: 2025-04-03, 2025.

[52] R. C. Martin, *The Single Responsibility Principle*, https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html, Publisher: Robert C. Martin, Accessed on: 2025-04-04, 2014.

[53] M. Kawazoe Aguilera, W. Chen und S. Toueg, „Heartbeat: A timeout-free failure detector for quiescent reliable communication," in *Lecture Notes in Computer Science*, Ser. Lecture notes in computer science, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, S. 126–140.

[54] The PostgreSQL Global Development Group, *13.3. Explicit Locking*, https://www.postgresql.org/docs/current/explicit-locking.html, Publisher: The PostgreSQL Global Development Group, Accessed on: 2025-04-03, 2025.

[55] D. E. Knuth, „Structured Programming with go to Statements," *ACM Comput. Surv.*, Jg. 6, Nr. 4, S. 268, 1974, ISSN: 0360-0300. DOI: 10.1145/356635.356640.

[56] M. Kleppmann, *Designing data-intensive applications*. Sebastopol, CA: O'Reilly Media, März 2017.