Fraunhofer IWM

HOME HOCHSCHULE MERSEBURG FH University of Applied Sciences

FACHBEREICH INFORMATIK UND KOMMUNIKATIONS-SYSTEME

Bachelorarbeit

zur Erlangung des Grades
Bachelor of Science (B.Sc.)

zum Thema

„Entwicklung einer Anwendung zur Steuerung der Stapelverarbeitungsfunktionen in Abaqus“

Verfasser:	Lars Richter
Geboren am:	24.11.1990
Geburtsort:	Greifswald
Hochschule:	Hochschule Merseburg (FH)
Studiengang:	BAIN10
Studienrichtung:	Angewandte Informatik
Eingereicht am:	28.08.2013
Mentor:	Prof. Dr. U. Heuert Hochschule Merseburg (FH) Fachbereich Ingenieur– und Naturwissenschaften (INW) Geusaer Straße 88, 06217 Merseburg
Betreuer:	Dipl. Inf. (FH) Jörg Santilian Fraunhofer-Institut für Werkstoffmechanik Halle Walter-Hülse-Straße 1, 06120 Halle

Inhaltsverzeichnis

Glossar

CPU	Central Processing Unit, keine klare Abgrenzung des Begriffs in der Literatur, Nutzung in dieser Arbeit als Synonym für eine datenverarbeitende Logik-Einheit, bzw. einen Prozessorkern
Job	ein abzuarbeitender Simulationsauftrag oder Prozess
.NET	von Microsoft entwickelte Softwareplattform, welche Schnittstellen und Klassenbibliotheken für verschiedene Programmiersprachen bereitstellt
SSH	Secure Shell, Netzwerkprotokoll zur Sicherstellung verschlüsselter und authentifizierter Netzwerkverbindungen
SQL	Structured Query Language, Programmiersprache für die Erstellung und Bearbeitung von relationalen Datenstrukturen und die Abfrage von Datensätzen aus relationalen Datenbanken
TCP	Transmission Control Protocol, Transportprotokoll zur zuverlässigen und verbindungsorientierten Paketvermittlung
Token	(engl. Wertmarke, Zeichen), wird verwendet um Ressourcen eindeutig zuzuweisen, synonyme Verwendung zu einer Lizenz
XML	Extensible Markup Language, standardisierte Sprache für die Beschreibung von Datenmodellen und Daten für den programm- und plattformunabhängigen Datenaustausch

Einleitung

Die Finite Element Methode (FEM) ist ein modernes Lösungsverfahren zum Berechnen von Festkörper- und Strömungssimulationen. Diese Methode liefert eine Näherung an eine exakte Lösung. Durch das Anpassen verschiedener Lösungsparameter kann man die Genauigkeit und Rechenzeit der Simulationen direkt beeinflussen. Eine Software für Finite Element Berechnungen stellt „Abaqus“ dar.

Die Software Abaqus/Standard ist ein Gleichungslöser für lineare und nichtlineare Differenzialgleichungen. So können mittels Abaqus Schwingungs-, Strömungsdynamiken, Mehrkörpersysteme, Crash-Analysen, Statikuntersuchungen, Wärme-, Akustikkopplungen und das Verhalten von Strukturen untersucht werden. Abaqus wird so bei physikalischen Problemstellungen unter anderen in Bereichen des Fahrzeugbaus, des Maschinenbaus und in der Luft- und Raumfahrttechnik verwendet.

Am Fraunhofer Institut für Werkstoffmechanik Halle (IWMH) wird ebenfalls Abaqus genutzt. Es werden Lösungen zur Erhöhung der Sicherheit, Verfügbarkeit und Lebensdauer von verschiedensten Bauteilen erarbeitet. Mit Abaqus werden Konzepte zum optimalen Werkstoffeinsatz und zur Formgebung erstellt.

Dabei werden für die aufwendigen Berechnungen bestimmte Mengen an Lizenzen benötigt. Da nicht unbegrenzt Lizenzen zur Verfügung stehen, kommt es oft zu einem Engpass bei der Nutzung von Abaqus. Um diesen Engpass zu umgehen, werden Simulationen mit einer minimalen Anzahl an Lizenzen gestartet. In Folge vervielfacht sich die benötigte Berechnungszeit und die Arbeit mit dem Simulationsprogramm wird unproduktiv.

In dieser Arbeit wird der Umgang mit der Simulationssoftware, die damit verbunden Arbeitsschritte und die Architektur des Softwaresystems, in welcher das Programm benutzt wird, analysiert. Des Weiteren wird ein Konzept zur optimalen Auslastung der Simulationsserver und der verfügbaren Lizenzen erarbeitet. Aufbauend auf diesem Konzept wird durch softwaretechnische Vorgehensweise ein System entwickelt, welches die Simulationsaufträge des Nutzers entgegen nimmt und automatisiert abarbeitet. Dieses System wird umfassend dokumentiert, um die Erweiterbarkeit sicher zu stellen.

1 Scheduling Grundlagen

Scheduling wird nötig, wenn zwei Prozesse gleichzeitig rechenbereit sind, aber nur begrenzte Ressourcen zur Verfügung stehen. Als Ressourcen können physische Geräte (z.B. Laufwerke), Informationseinheiten (z.B. gesperrte Datensätze), Prozessorkerne oder Lizenzen gelten. Nun muss der Scheduler entscheiden, welcher Prozess der Ressource zugewiesen wird. Diese Entscheidung wird durch Schedulingalgorithmen getroffen. In der Betrachtung der Algorithmen stellt eine *Central Processing Unit (CPU)* die Ressource für einen Prozess dar.

1.1 Einführung

Das Scheduling findet keine Anwendung, wenn nur ein Prozess sich im rechenbereiten Zustand befindet. Erst wenn mehr Prozesse existieren, als verfügbare Ressourcen zur Verfügung stehen, wird Scheduling benötigt. Sonst kommt es dazu, dass lange Warteschlangen entstehen, viele Prozesse durch einige rechenintensive Prozesse blockiert werden und die Abarbeitung insgesamt verzögert wird.

Das Scheduling wird unter verschiedenen Zielsetzungen durchgeführt. Jeder Prozess muss Rechenzeit auf der CPU erhalten. Die CPU darf nicht durch einzelne Prozesse blockiert werden. Dieses Ziel wird als Fairness bezeichnet. Des Weiteren müssen die gesetzten Schedulingregeln eingehalten und sichtbar durchgeführt werden. Keine der festgelegten Regeln darf durch einzelne Prozesse außer Kraft gesetzt werden. Das dritte Hauptziel ist die Balance, welches beschreibt, dass alle Systemkomponenten gleichmäßig ausgelastet sein müssen. Dies wird erreicht, indem ein- und ausgabelastige Prozesse mit rechenintensiven Prozessen vermischt werden, um einen Leerlauf der CPU zu verhindern.

Zur Beurteilung eines Schedulingalgorithmus, welcher die aufgezählten Ziele verfolgt und die Regeln einhält, existieren drei Metriken: Durchsatz, Durchlaufzeit und CPU-Auslastung [1]. Der Durchsatz beschreibt wie viele Prozesse pro Zeiteinheit abgearbeitet werden. Je mehr Prozesse gerechnet werden, desto besser. Die Durchlaufzeit beschreibt die Zeit von dem Moment an, als der Prozess geladen wurde, bis zum Zeitpunkt, bei dem der Prozess beendet. Die dritte Metrik ist die CPU Auslastung, welche nur bedingt als Merkmal genutzt werden kann. Die Auslastung beschreibt nur, wie stark das System zurzeit belastet wird und dementsprechend sollten zur Effizienzsteigerung Leerlaufzeiten vermieden werden.

Generell ist das Ziel, den Durchsatz zu maximieren und die Durchlaufzeit zu minimieren. Jedoch müssen beide Ziele nicht unbedingt miteinander einhergehen. So kann man den Durchsatz durch die Abarbeitung vieler kurzer Prozesse maximieren, indem man sie langen Prozessen voran stellt. Dadurch erhöht sich jedoch die mittlere Durchlaufzeit, da die langen

Prozesse nie von der CPU gerechnet werden. Bei der Entwicklung eines Scheduling-algorithmus muss man so sehr genau darauf achten, welche Konsequenzen auf lange Sicht entstehen könnten.

Von den allgemeinen Zielen unterscheiden sich spezielle Ziele verschiedener Softwaresysteme. Die Systeme lassen sich in 3 Umgebungen gliedern [1].
Stapelverarbeitungssysteme arbeiten eine Menge an Prozessen mit unbestimmter Länge ab. Da keine Benutzerinteraktion zu erwarten ist, optimieren die Scheduler in diesen Systemen auf Durchsatz, der Durchlaufzeit und der CPU Auslastung. Interaktive Systeme rechnen mit der Unterbrechung durch den Benutzer. Man muss folglich laufende Prozesse unterbrechen, um geringe Antwortzeiten zu erhalten. In Echtzeitsystemen rechnen bekannte Prozesse, welche stets nur eine kurze Zeit aktiv sind. Dadurch wird darauf geachtet, dass Deadlines eingehalten werden und die Laufzeit der einzelnen Prozesse voraus gesagt werden können.

Bei der Auswahl eines geeigneten Schedulingalgorithmus muss auf die Abarbeitung der FEM Simulationen am Fraunhofer Institut für Werkstoffmechanik in Halle eingegangen werden. Dabei handelt es sich um ein Stapelverarbeitungssystem. Die limitierende Größe ist nicht die Anzahl der CPUs der Server, sondern die verfügbaren Softwarelizenzen. Ungeachtet dessen bleiben die Ziele und die Metriken, nach denen ein Algorithmus zu optimieren ist dieselben. Die Analyse des Lizenzsystems hat gezeigt, dass es am effizientesten ist eine Simulation mit allen verfügbaren Lizenzen rechnen zu lassen. Aus diesem Grund bezieht sich die weitere Betrachtung der Schedulingalgorithmen auf die nicht-parallele Ausführung der Simulationen.
Des Weiteren lassen sich keine Aussagen darüber treffen, wie viel Rechenzeit eine Simulation zur Abarbeitung genau benötigen wird. Man kann diese Größe nur ungenau schätzen. Aus diesem Grund ist ein Scheduling, welches die verbleibende Rechenzeit eines Prozesses berücksichtigt, nicht möglich.

1.2 First-Come First-Served

Die Abarbeitung der Simulationen durch die Stapelverarbeitungsfunktion von Abaqus erfolgt strikt nach den *First-Come First-Served* Scheduling. Dabei erhält die Simulation das Recht zu rechnen, die zuerst die Ressourcen beantragt. Später eintreffende Simulationen werden in einer Warteschlange nacheinander abgearbeitet.

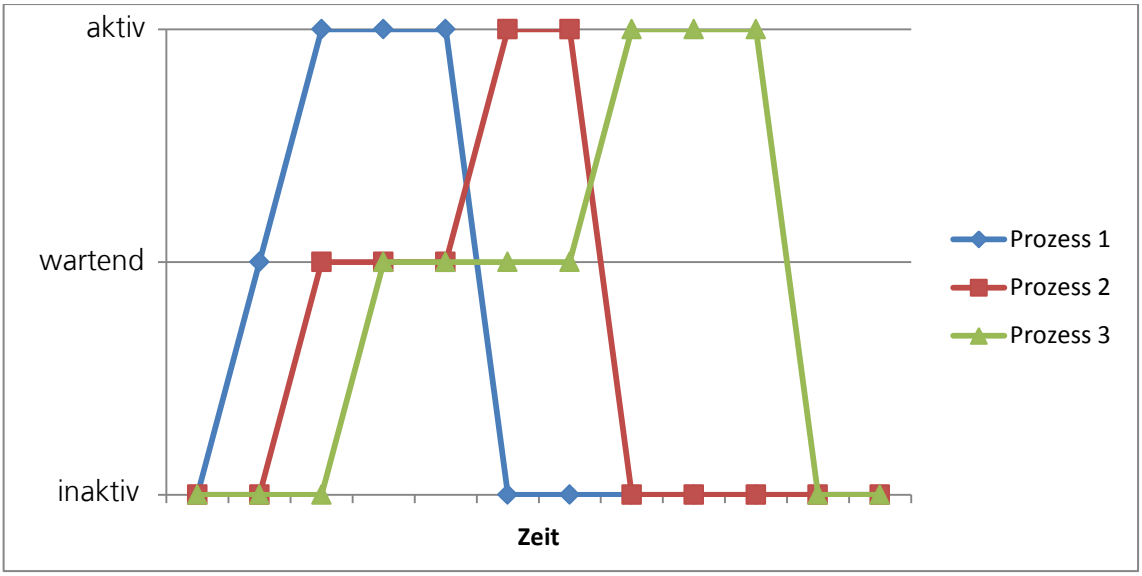


Abbildung 1: Prozessablauf nach dem First-Come First-Served Scheduling

Dieser Schedulingalgorithmus gilt als sehr einfach und schnell zu implementieren. Der Durchsatz und die Durchlaufzeit hängen sehr stark von den einzelnen Prozessen ab, welche gestartet werden. Optimierungen können nicht vorgenommen werden, da alle Prozesse in einer einzigen Warteschlange abgearbeitet werden. Andrew S. Tanenbaum beschreibt die Fairness sehr treffend anhand eines Beispiels: „[Der Algorithmus] ist im gleichen Sinne gerecht, wie es gerecht ist, dass knappe Eintrittskarten für Sport- oder Konzertveranstaltungen an Leute vergeben werden, die bereit sind, sich um 2 Uhr in der Früh anzustellen.“[1]
Darin begründet sich der Nachteil des *First-Come First-Served* Schedulings. Prozesse die am Ende der Warteschlange stehen warten eine unbekannt lange Zeit, bis sie gerechnet werden. Dabei kann es dazu kommen, dass Sie innerhalb eines Zeitabschnittes nie Ressourcen erhalten. Dieses Verhalten ist besonders gefährlich, da in der Warteschlange keine Unterscheidung zwischen hohen und niedrigpriorisierten Prozessen getroffen wird.

1.3 Prioritätenbasierendes Scheduling

Um die Möglichkeit zu schaffen, wichtige Simulationen, welche dringend für die Einhaltung eines Termins benötigt werden, bevorzugt rechnen zu lassen, existiert das prioritätenbasierende Scheduling. Dabei wird durch den Nutzer eine Einschätzung über die Priorität getroffen. Die Priorität hängt dabei von der Dringlichkeit des Ergebnisses, sowie von der erwarteten Simulationsdauer ab. So ist es tragbar, wenn eine große Simulation an statt zehn Stunden, durch den Einschub einer kurzen Simulationen, zehn Stunden und fünf Minuten zur Berechnung benötigt. Wenn sich die Durchlaufzeit der kurzen Simulation von

[1] Tanenbaum, 2003, S. 156

fünf Minuten allerdings auf über zehn Stunden verlängert, ist dies nicht mehr verhältnismäßig.

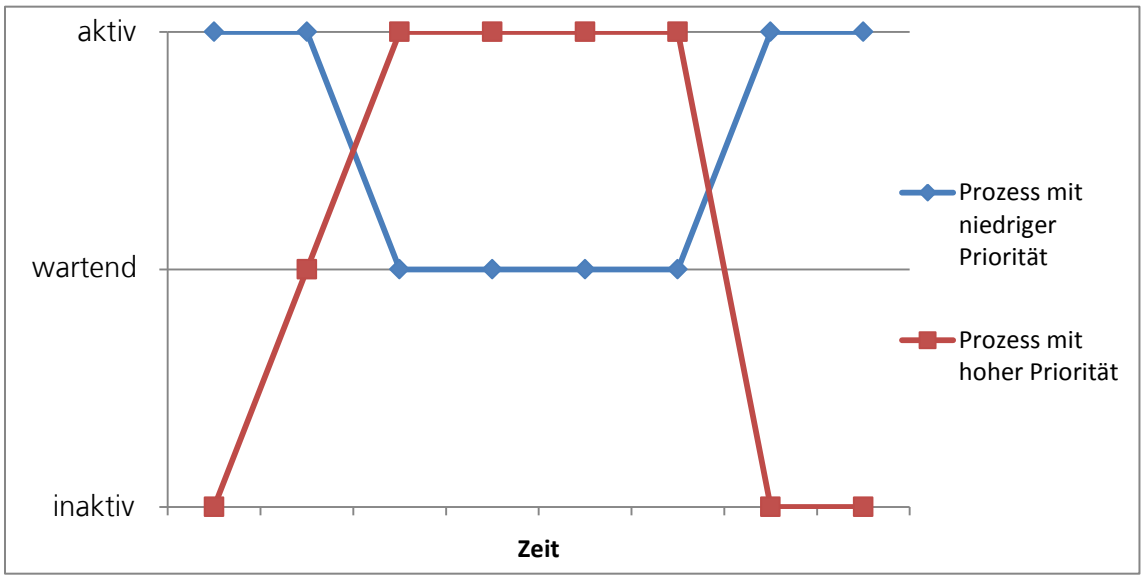


Abbildung 2: Prozesswechsel nach dem Prioritätsbasierenden Scheduling

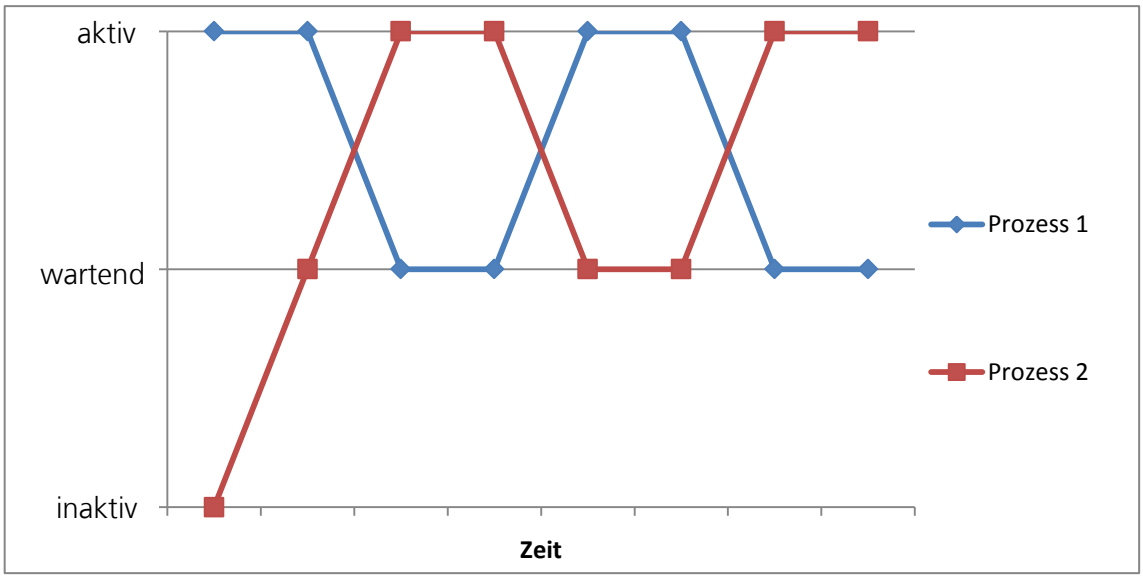
Die Prozesse werden dabei, dem *First-Come First-Served* Scheduling ähnlich, in einer Warteschlange verwaltet. Anstatt den ersten Prozess der Warteschlange auszuwählen, wird die Warteschlange nach dem nächsten höchstpriorisierten Prozess durchsucht, welcher nachfolgend gestartet wird.

Prioritätsbasierendes Scheduling kann mittels statisch oder dynamisch festgelegten Prioritäten erfolgen. Statisch festgelegte Prioritäten werden zu Prozessstart definiert und anschließend nicht verändert. Dynamische festgelegte Prioritäten gehen von diesem statisch festgelegten Wert aus, verändern diesen jedoch anschließend. So kann die Festlegung getroffen werden, dass jeder Nutzer nur eine bestimmte Anzahl an hochpriorisierten Prozessen besitzen darf. Kommt es dazu, dass die Anzahl überschritten wird, werden die Prozesse des Nutzers niedriger priorisiert, bis die Anzahl wieder unterschritten ist.
Eine weitere Möglichkeit ist die Priorität nach bestimmten Zeitabschnitten zu verringern. Wenn die Priorität unter die eines anderen Prozesses fällt, wird dieser gestartet, der derzeit aktive Prozess angehalten und an das Ende der Warteschlange gesetzt. Dies verhindert, dass lange Prozesse ewig rechnen und niedriger priorisierte Prozesse „verhungern", weil sie nie gerechnet werden.

Der ungünstigste Fall der bei dem Scheduling mit statischen Prioritäten eintreten kann ist der, dass die Nutzer das System der Verteilung durchschaut haben und nun alle Prozesse mit der höchsten Priorität starten. Im Ergebnis entsteht eine Warteliste mit Prozessen gleicher Priorität. So haben die Nutzer das prioritätenbasierende Scheduling außer Kraft gesetzt und das *First-Come First-Served* Scheduling eingeführt.

1.4 Prioritätsbasierendes Round Robin Scheduling

Um das Blockieren durch lange hochpriorisierter Prozesse abzuschwächen, kann das Round Robin Scheduling angewendet werden. Dabei werden die Prozesse ihrer Priorität nach einzelnen Warteschlangen zugeteilt. Es entstehen sogenannte Prioritätskategorien. Innerhalb der Warteschlangen wird das Round Robin Scheduling angewendet.
Beim Round Robin Scheduling bekommt jeder Prozess ein bestimmtes Zeitquantum, einen Zeitabschnitt oder Zeitscheibe, zugewiesen. Während dieser Zeit wird er der CPU zugeteilt. Wenn er zum Ende des Quantums noch rechnet, wird er unterbrochen und an das Ende der Warteschlange gestellt. Dieses Schedulingverfahren ist fairer innerhalb der Prioritätsklassen, als das reine prioritätsbasierende Scheduling. Nun kann jedem Prozess ein bestimmter Zeitabschnitt zugesichert werden.



Während der Prozesswechsel wird eine bestimmte Zeit für die Verwaltung benötigt. Man spricht dabei auch von einem Kontextwechsel bei dem Dateien gespeichert und geladen werden. Durch zu kleine Quanten nimmt der Prozesswechsel im Bezug zur Gesamtrechenzeit, zu viel Rechenzeit in Anspruch. Die Quanten sollten folglich so groß gewählt werden, dass die Effizienz nicht darunter leidet.

Wie beim prioritätsbasierenden Scheduling, kann es passieren dass die Nutzer sämtliche Prozesse mit der höchsten Priorität starten, weil diese im Glauben sind, dass ihre eigene Simulationen dann schneller beendet werden. Das einzige was erreicht wird ist, dass die statistische Durchlaufzeit aller Prozesse unnötig in die Höhe getrieben wird, wenn sie alle Prozesse mit der höchsten Priorität starten. Um solch ein Verhalten zu verhindern, müsste der Nutzer bevormundet werden, in dem der Prozess stets nach einem verbrauchten Quantum in der Priorität verringert wird. Das würde aber dazu führen, dass durchaus wichtige Prozesse zu spät beendet werden.

Verlässt man sich jedoch auf die Einschätzung der Priorität durch den Nutzer, ist es möglich ein sehr effizientes Scheduling zu realisieren. Durch die Annahme, dass kurze Prozesse

bevorzugt in den hohen Prioritätsklassen und lange Prozesse in den niedrigeren Prioritätsklassen existieren kann man unterschiedliche Quantengrößen nutzen. Die Abarbeitung der kurzen Prozesse wird durch kurze Zeitquanten nach der Prämisse des Durchsatzes optimiert. Es wird versucht, möglichst viele Prozesse pro Zeit zu bearbeiten. Dabei darf man das Zeitquantum nicht zu klein zu wählen, da sonst das Verhältnis von Prozesswechsel zu Prozessrechenzeit unausgewogen ist.

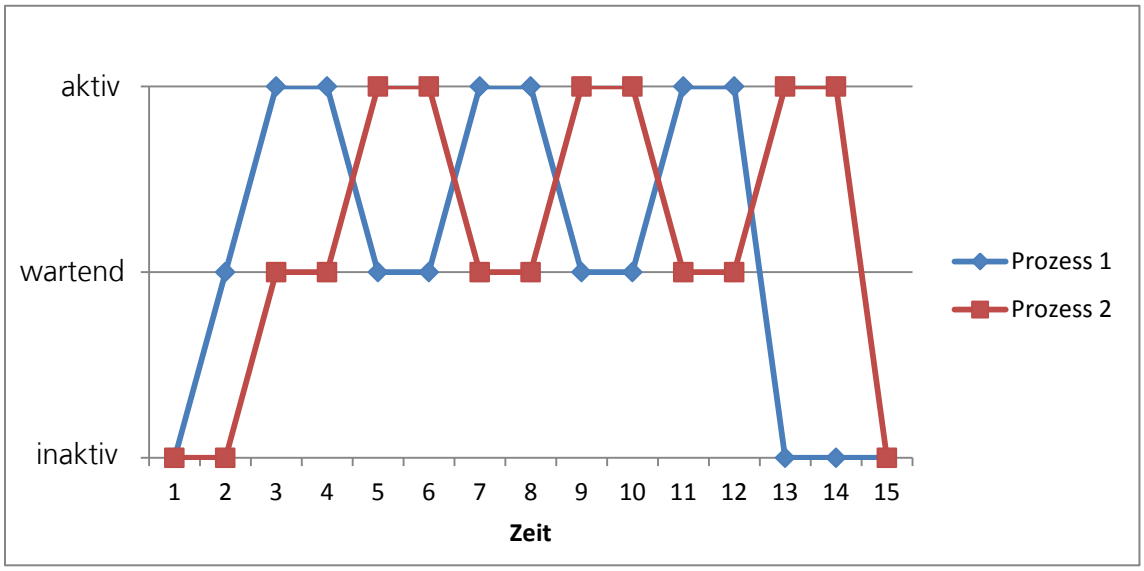


Abbildung 3: Zu häufige Prozesswechsel durch zu kurze Zeitquanten

Lange Prozesse können seltener gewechselt werden, da der Nutzer die Ergebnisse nicht sofort benötigt. Dabei wird die Durchlaufzeit optimiert, in dem möglichst wenig Rechenzeit für Prozesswechsel verbraucht wird.

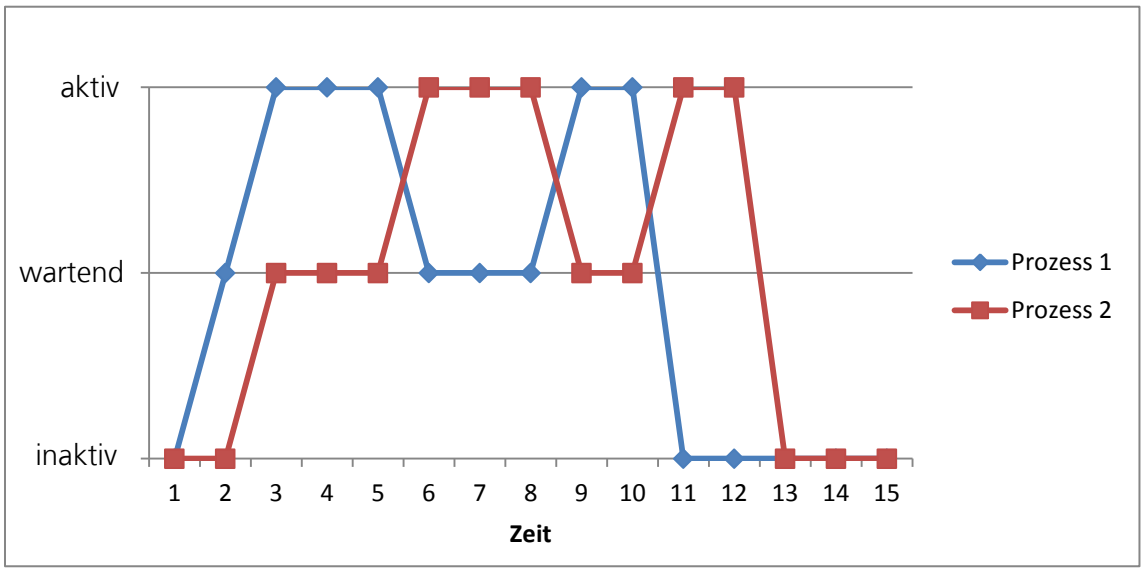


Abbildung 4: Optimalere Quantengröße führt zur Verringerung der Durchlaufzeit

Um den Vorteil für den Nutzer darzustellen, seine Prozesse richtig zu priorisieren, werden exemplarisch 2 Beispiele dargestellt.
In einem System mit zwei Prioritätsklassen werden die Prozesse in der höheren Klasse jede Minute gewechselt und in der niedrigeren Klasse alle 10 Minuten. Nun werden gleichzeitig 2 Prozesse mit hoher Priorität gestartet. Die Prozesse besitzen eine Rechendauer von jeweils 100 Minuten. Wenn ein Prozesswechsel zehn Sekunden benötigt, sind beide Simulationen nach 232 Minuten fertig gerechnet. Das würde bedeuten, dass insgesamt 32 Minuten für die Prozesswechsel benötigt wurden.
Wenn die Prozesse in der niedrigen Prioritätsklasse gestartet worden wären, wären nur 20 an statt der 200 Prozesswechsel nötig gewesen. Die Prozesse wären nach 203,2 Minuten erfolgreich beendet worden.

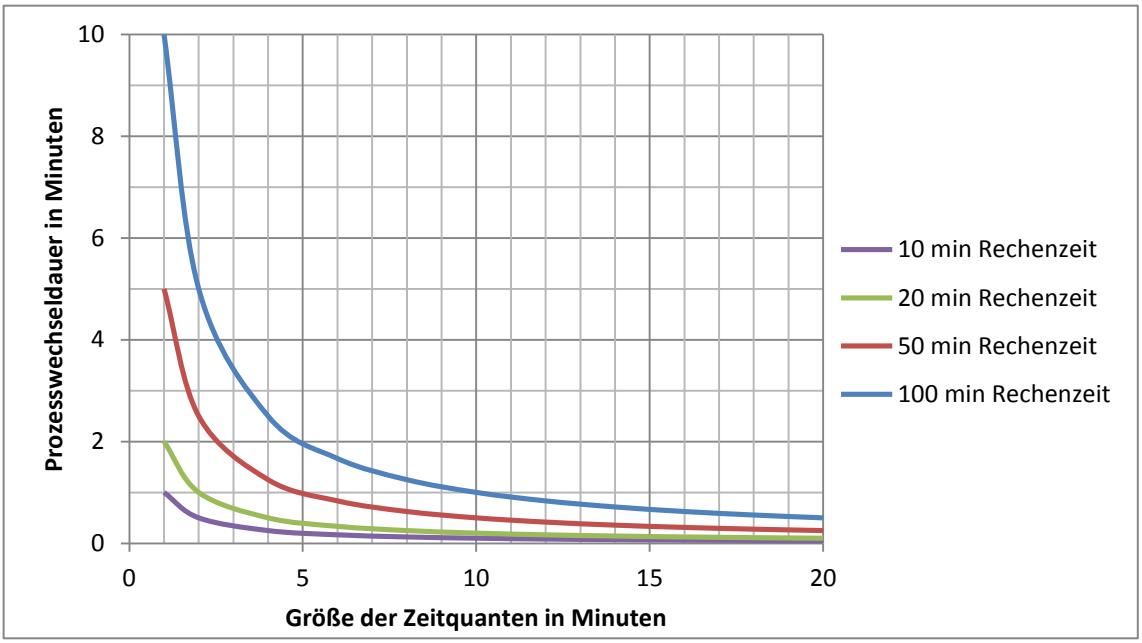


Abbildung 5: Verhältnis der Quantengröße zur gesamten Prozesswechseldauer pro Simulation bei einer Zeit von zehn Sekunden pro Prozesswechsel

Das *Round-Robin*-Scheduling wird mit zunehmender Anzahl an Prozessen in derselben Prioritätsklasse, durch den linearen Anstieg der Prozesswechsel, weniger effizient. Dem kann man nur entgegenwirken, indem längere Quanten gesetzt werden. Abhängig von der Priorität werden verschiedene Quantumgrößen gewählt. Hochpriorisierte, kurze Simulationen werden in Zeitquanten von 2 Minuten gewechselt. Dadurch wird garantiert, dass eine wichtige Simulation innerhalb kürzester Zeit berechnet wird.
Lange Simulationen werden in Zeitquanten von 10 Minuten gewechselt. Dadurch wird erreicht, dass die Prozesswechseldauer nur 1% der Gesamtrechenzeit beträgt.

Des Weiteren sollten die Parameter im laufenden Betrieb modifizierbar bleiben, um das Scheduling neuen Situationen anpassen zu können.

2 Analyse

Die detaillierte Analyse hat für das zu entwickelnde Softwaresystem einen besonderen Stellenwert. Bei Aufnahme der Arbeit waren die Spezifikationen des vorhandenen Softwaresystems noch nicht klar vorhanden. Es mussten die am Scheduling beteiligten Komponenten erfasst werden und es musste geprüft werden, inwiefern sie sich für ein Scheduling eignen.

2.1 Softwaretechnische Vorgehensweise

Eine softwaretechnische Vorgehensweise sichert die Qualität und den Erfolg einer Softwareentwicklung. Um Softwareanforderungen und Realisierungsdetails spezifizieren zu können, wurde eine protypische Entwicklung durchgeführt.

Das Entwicklungsmodell des „Explorativen Pototypings“ ist sehr sinnvoll, um schnell Systemanforderungen festzulegen, welche noch nicht eindeutig bekannt sind. Es werden vorrangig funktionale Komponenten entwickelt, um einen lauffähigen Prototypen zu generieren. Man erhält so die Möglichkeit, in einem frühen Entwicklungsstadium, die Anforderungen mit den zukünftigen Anwendern detailliert zu diskutieren.

Schnittstellen und Komponenten, deren Spezifikationen noch ungewiss sind, lassen sich am ehesten durch das „Experimentelle Prototyping“ erfassen. Dabei wird die Tauglichkeit und Einsetzbarkeit einzelner Softwarekomponenten geprüft. So wird verhindert, dass die Entwicklung der Software bereits in der Entwurfsphase scheitert, weil man sich auf eine zu komplizierte Komponente gestützt hat.

Im Zusammenschluss beider Prototypingmodelle mit dem „Evolutionären Prototyping“ erhält man eine robuste Grundlage für weitere Entwicklungsphasen. Das System geht dabei von den einfachsten Anforderungen und hohlen Schnittstellen und Komponenten aus. Im Laufe weiterer Entwicklungsschritte wird es stetig erweitert und getestet. Die Entwurfsdokumente werden während dessen stets erweitert und korrigiert, falls die Entwicklung eine andere Richtung nimmt, als ursprünglich entworfen. [2]

Selbst wenn die Entwicklung eines Komponentenprototypen zeigen sollte, dass die Implementierung zu komplex oder der Prototyp nicht nutzbar ist, so kann man immer noch in einer frühen Entwicklungsphase einen neuen Prototyp implementieren. Im klassischen Wasserfallmodell, in dem die Implementierung erst nach umfangreicher Analyse und Entwurfsphase stattfindet, würde eine solche Maßnahme nicht ohne erhebliche Mehrkosten möglich sein. Ebenso stellt das Prototyping eine Verbesserung der Softwarequalität bei Auslieferung an den Kunden dar, da dieser schon während der Entwicklung, die Software nach seinen Wünschen gestalten kann. [3]

Jedoch birgt genau dieses Vorgehen auch Risiken. So kann es passieren, dass der Kunde mit dem Test der ersten Prototypen einen sehr weit fortgeschrittenen Entwicklungsprozess assoziiert, obwohl die Prototypen lediglich zum verifizieren der Anforderungen dienen. So kann es zu Missverständnissen über den weiteren Aufwand der Entwicklung kommen. [3]

Unterstützt werden die Ausarbeitungen durch die *Unified Modeling Language (UML)*. Durch die zunehmende Komplexität und der Prämisse der Wiederverwendbarkeit von Software, stellt die Modellierungssprache einen einheitlichen Dokumentationsstandard sicher. Es existiert eine Vielzahl von Modellen, welche Teilprobleme leicht erfassbar werden lassen und unter verschiedenen Schwerpunkten betrachten. [4]

Die weiteren Entwicklungsschritte werden durch die UML unterstützen dargestellt. Dabei wurde darauf geachtet keine Überspezifizierung der Systeme vorzunehmen.

2.2 Ausgangssituation

Die Finite Element Methode ist ein Verfahren zur näherungsweisen Bestimmung einer numerischen Lösung. Dabei wird ein Modell durch verschiedene Eigenschaften und Kräfte beschrieben. Durch verschiedene Parameter wird der Simulationsaufwand direkt erhöht. Ein Modell wird unter anderen durch Knoten beschrieben. Mit einer Erhöhung der Knotenanzahl kommt es zu einer Verfeinerung des Modells. Jeder Knoten besitzt eine bestimmte Anzahl an Freiheitsgraden, welche eine Verschiebung und Rotation erlauben können. Die Erhöhung der Anzahl der Freiheitsgrade geht mit einer Vervielfachung des Rechenaufwands einher. [5]

Der Gleichungslöser der Software Abaqus ist darauf ausgelegt auf mehreren CPUs gleichzeitig (parallel) zu rechnen, um eine Verkürzung der Simulationsdauer zu erreichen. Jedoch werden abhängig von der verwendeten CPU Anzahl die entsprechenden Lizenzen, auch Tokens genannt, benötigt. Da diese Lizenzen käuflich erworben werden müssen, stehen nicht unbegrenzt viele zur Verfügung. Wird eine Simulation gestartet, wenn nicht genügend Lizenzen zur Ausführung zur Verfügung stehen, wird die Simulation in einer Warteliste pausiert. Sind genügend Lizenzen vorhanden, wird die Simulation gestartet.

Der Engpass bei der Abarbeitung der Simulationen in Abaqus ist begründet in der geringen Anzahl der Lizenzen. Um das Problem zu umgehen, dass eine Simulation eine andere blockiert, wird mit der minimalen CPU Anzahl gerechnet. Durch die nicht lineare Verteilung der Lizenzen in Bezug zu der CPU Anzahl wird dieser Engpass noch verstärkt, da die einzelnen Simulationen unnötig lange rechnen.

Tabelle 1: Verhältnis der Lizenzen im Bezug zur CPU Anzahl

CPUs	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Lizenzen	5	6	7	8	9	10	11	12	12	13	13	14	14	15	15

Um die Problematik zu verdeutlichen wird voraus gesetzt, dass insgesamt 15 Lizenzen zur Verfügung stehen. Die einzelnen Nutzer starten ihre Simulation nun mit einer CPU damit sie nur 5 Lizenzen blockieren. So können maximal 3 Nutzer gleichzeitig ihre Simulationen ausführen.
Tests haben gezeigt, dass eine Simulation mit nur einer CPU 105 Minuten zur Abarbeitung benötigt. Dieselbe Simulation würde mit allen 15 CPUs jedoch nur 13 Minuten benötigen.
Würden sich die Nutzer absprechen, wären alle spätestens nach 40 Minuten mit ihren Simulationen fertig. So müssen die Benutzer jedoch 65 Minuten länger auf das Ergebnis der Simulation warten, weil 3 Simulationen gleichzeitig mit nur jeweils einer CPU rechnen.

Das andere Extrem stellt eine Simulation dar, welche mit der Maximalanzahl an Lizenzen rechnet. In diesem Fall würde niemand gleichzeitig rechnen können. Im Falle, dass die Simulation mehrere Tage zum berechnen benötigt, wäre ein Projektrückstand im Zusammenhang mit den wartenden Simulationen vorprogrammiert.

Die unkomplizierteste Möglichkeit, den Engpass vorerst zu beseitigen, liegt in der Erhöhung der verfügbaren Lizenzen, in dem man weitere einkauft. Jedoch ist solch eine Investition nicht zu rechtfertigen, wenn es lange Leerlaufzeiten gibt, in denen das Programm nicht benötigt wird.

Es wird eine Lösung gesucht, welche Simulationen mit der maximal nutzbaren CPU Anzahl ausführt, und diese entsprechend verteilt. Dabei können Simulationen pausiert und fortgesetzt werden. Damit ist die Grundvoraussetzung für ein Scheduling vorhanden.

2.3 Architekturübersicht

Um einen Überblick über die vorhandenen Softwaresysteme zu schaffen und die Analyseschwerpunkte einschätzen zu können, wird die Architekturübersicht der Detailanalyse voran gestellt.
Die vorhandene Softwarearchitektur teilt sich in drei Komponenten auf. Der Nutzer greift direkt auf den Dateiserver und die Simulationsserver zu. Dabei werden die Simulations-Eingabedateien auf dem Dateiserver gespeichert. Der Nutzer startet eine Simulation durch einen Kommandozeilenbefehl an einen Simulationsserver. Dieser greift auf die Dateien auf dem Dateiserver zu und beantragt die Lizenzen vom Lizenzserver. Die Zwischenergebnisse und Statusdateien der Simulation werden von Abaqus auf dem Dateiserver abgelegt. Während der Simulation kontaktiert Abaqus stets den Lizenzserver, ob die Lizenzen noch in Benutzung sind. Endet die Simulation gibt Abaqus die Lizenzen frei.

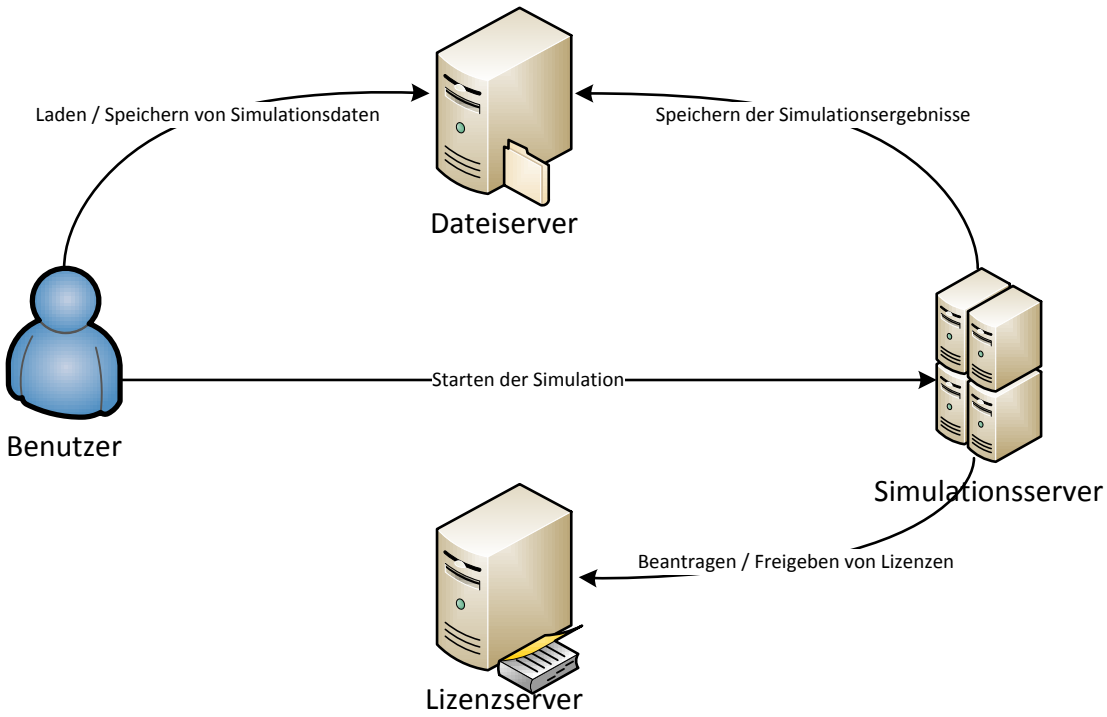


Abbildung 6: Übersicht über die vorhandenen Softwaresysteme

Der Nutzer interagiert folglich nur mit den Datei- und Simulationsservern. Nur die Simulationsserver auf denen Abaqus rechnet, interagieren zum Dateiserver mit dem Lizenzserver.

2.4 Simulationssoftware Abaqus

Abaqus stellt ein Software Paket dar, welches mittels der Finiten Element Methode lineare und komplexe nicht lineare Simulationen lösen kann. Der Nutzer wird in der Erstellung und Auswertung von Simulationen durch Programme mit grafischen Oberflächen unterstützt. Das Softwarepaket teilt sich so in 3 Hauptkomponenten auf. Software für die Vorbereitung der Simulationsdaten (*Preprocessor*), die Berechnung der Simulationen (*Solver*) und die Auswertung der gewonnenen Ergebnisse (*Postprocessor*) [6].

2.4.1 Simulationsablauf

Der typische Ablauf einer Simulation wird durch die Softwarepakete von Abaqus unterstützt. Dabei ist es nicht zwingend notwendig, die Vorbereitung und Nachbehandlung in diesen vorzunehmen.

Abbildung 7: Arbeitsschritte im Zusammenhang mit einer Simulation

Die Vorbereitung der Simulationsdateien findet in der Software *Abaqus/CAE* statt. Es werden die Eigenschaften und die wirkenden Kräfte eines physischen Modells beschrieben. Ebenso können CAD Modelle geladen und bearbeitet werden. Die fertige Eingabedatei, das *Input File*, stellt dabei eine Textdatei dar, welche mit einem Texteditor bearbeitet werden kann.

Diese Eingabedatei wird in den Abaqus Gleichungslöser geladen. Abhängig von der Größe des Modells und den Parametern kann die Berechnung der Simulation einige Sekunden bis Tage beanspruchen. Durch den `cpus` Parameter lässt sich die Rechnung auf mehrere Prozessorkerne verteilen. Dadurch wird eine nichtlineare Verkürzung der Simulationsdauer in Beziehung zur CPU-Anzahl erreicht.

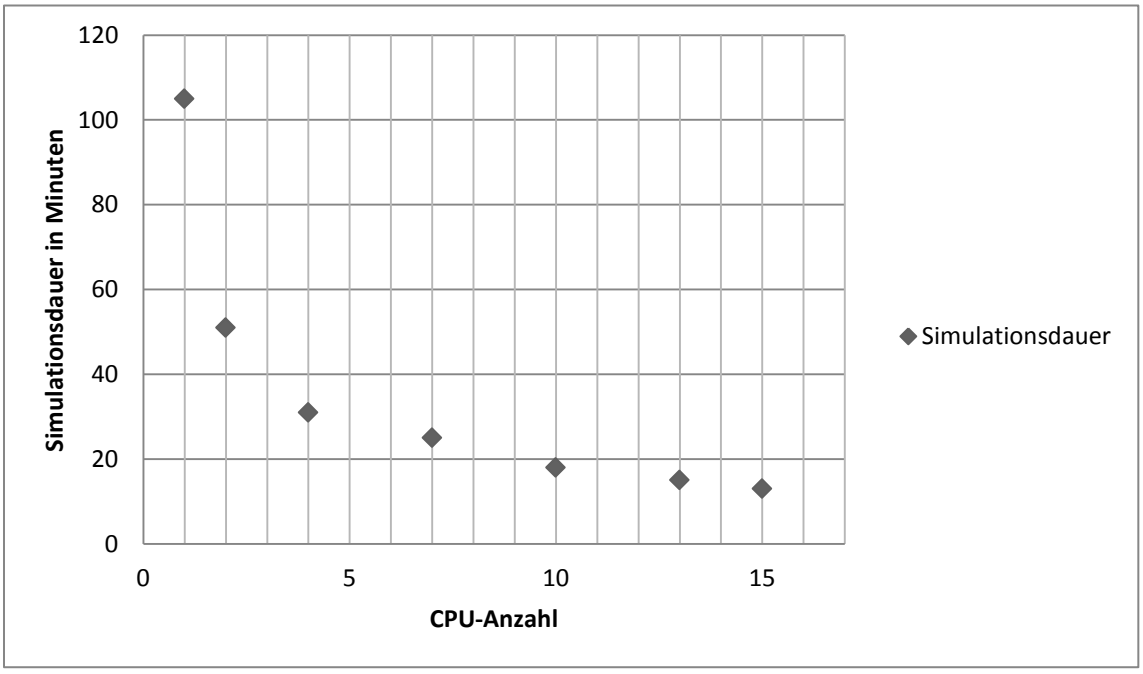


Abbildung 8: Verhältnis der Simulationsdauer einer Simulation bezüglich der genutzten CPU-Anzahl

Die Rechnungen werden von besonders leistungsstarken Servern bewältigt. Auf diesen Servern meldet sich der Nutzer per *Secure Shell (SSH)* an und sendet an diesen den Simulationsauftrag. Dabei wird auf eine Eingabe-Datei verwiesen, welche sich auf dem Netzlaufwerk befinden muss, auf welches der Server Zugriff hat. Folgend der Befehl zum Start einer Simulation:

```
abaqus job=eingabedatei cpus=5
```

Die Eingabedatei beschreibt das Simulationsmodell. Die Rechnung soll auf 5 CPU-Kernen rechnen. Weitere Kommandozeilenbefehle um Simulationsaufträge zu beeinflussen sind `suspend` für das Pausieren, `resume` für das Fortsetzen und `terminate` für das Abbrechen einer Rechnung.

Wenn die Rechnung abgeschlossen ist sind im selben Verzeichnis der Eingabedatei die Ergebnisse der Simulation zu finden. Die Resultate können in der Softwarekomponente *Abaqus/Viewer* betrachtet und ausgewertet werden.

2.4.2 Auswertung nutzbarer Dateien

Mit dem Beginn einer Rechnung durch Abaqus werden verschiedene Dateien erzeugt. Am interessantesten sind dabei Informationen aus den Dateien, welche Angaben zum derzeitigen Fortschritt der Simulation machen.

Die Eingabedatei wird dabei zuerst von einem Präprozessor vorbereitet. Anschließend wird die Analyse gestartet. Die Simulation kann unerwartet beenden. Dies wird in einer Logdatei, sowie in einer Statusdatei angezeigt. Die Abfrage beider Dateien lässt eine verlässliche Aussage über den derzeitigen Status einer Simulation zu.
In beiden Dateien wird in der letzten Zeile der aktuelle Zustand beschrieben. So lassen sich diese Zeilen nach Stichwörtern durchsuchen, mit denen man auf den Status der Simulation schließen kann.

Tabelle 2: Zuordnung von Simulationszuständen zu Stichwörtern in Log- und Statusdateien

Zustand der Simulation	Logdatei	Statusdatei
in Vorbereitung	Abaqus Job / Run pre.exe	(Datei noch nicht angelegt)
erfolgreich beendet	COMPLETED	HAS COMPLETED
nicht erfolgreich beendet	exited with errors	NOT BEEN COMPLETED

Wenn die Log- und Statusdateien nicht vorhanden sind, wurde die Simulation noch nicht durch Abaqus gerechnet. In der Vorbereitung existiert die Statusdatei noch nicht, diese wird erst mit der aktiven Simulation angelegt, da in dieser Informationen zu den Zwischenschritten der Rechnung gespeichert werden.
Leider ist es nicht möglich, verlässliche Aussagen über die noch benötigte Simulationsdauer zu treffen. Dadurch können keine Schedulingalgorithmen verwendet werden, welche sich auf diese Daten stützen.

2.4.3 Schnittstelle

Die Simulationen werden durch einen Kommandozeilenbefehl gestartet. Dazu wird ein verschlüsselter und authentifizierter Zugang zum Simulationsserver durch das Secure Shell (SSH) Protokoll aufgebaut. Hierbei wird durch den Diffie-Hellman Schlüsselaustausch ein gemeinsamer Schlüssel für die Übertragung vereinbart. In dieser verschlüsselten Verbindung authentifiziert sich der Nutzer bei dem Server, indem er eine Signatur mit seinem privaten Schlüssel erstellt, welche vom Server mit dem öffentlichen Schlüssel des Nutzers geprüft wird. Diese Signatur kann nur vom Nutzer erzeugt worden sein. [7]

Dieser Authentifizierungsprozess kann gut implementiert werden, indem auf eine SSH Library und die .NET Umgebung zugegriffen wird. Dazu wird der entsprechende Befehl per Shell Execute an den Server gesendet.

2.5 Lizenzsoftware

2.5.1 Lizenzmanagement

Abaqus nutzt den *FLEXnet network license manager* zur Lizenzverwaltung. Dabei werden die Lizenzen, auch Tokens genannt, bei Simulationsstart von dem Simulationsserver beantragt. Die Tokens werden beim Lizenzmanager ausgecheckt und der Simulation zugewiesen. Somit stehen diese Tokens nun einer anderen Simulation nicht mehr zur Verfügung. Während der Simulation wird periodisch der Lizenzserver kontaktiert und darüber informiert, dass die Tokens noch benötigt werden und nicht freigegeben werden können. Dies werden sie erst, wenn Abaqus die Simulation beendet hat.
Kommt es dazu, dass Abaqus eine Simulation startet, wenn nicht mehr genügend Tokens zur Verfügung stehen, wird die Simulation auf dem Lizenzserver in eine Warteliste geschrieben. Abaqus prüft nun solange den Platz auf der Warteliste, bis die Simulation die benötigten Tokens zugewiesen bekommen hat und somit gerechnet werden darf.

Das Lizenzsystem von Abaqus lässt sich als mehrkernoptimierend bezeichnen, um die das nicht lineare Verhalten von Simulationsdauer zur Anzahl genutzter CPUs mit steigender Simulationsdauer auszugleichen (vgl. S.18 Abbildung 8: Verhältnis der Simulationsdauer einer Simulation bezüglich der genutzten CPU-Anzahl).

Der Zeitgewinn bei der Verwendung von zwei anstatt einer CPU beträgt ca. 50%. Bei einer weiteren Verdopplung der genutzten Kerne beträgt der zusätzliche Zeitgewinn nur noch 40%. Die Lizenzverteilung ist dem angepasst, indem anfangs 4 Token pro Simulation + ein Token pro genutzte CPU beansprucht wird. Ab 8 CPUs verringert sich der Tokenbedarf aller 2 CPUs um ein Token. So bleibt es effizient mit vielen Kernen zu rechnen, obwohl die Simulationsgeschwindigkeit langsamer steigt.

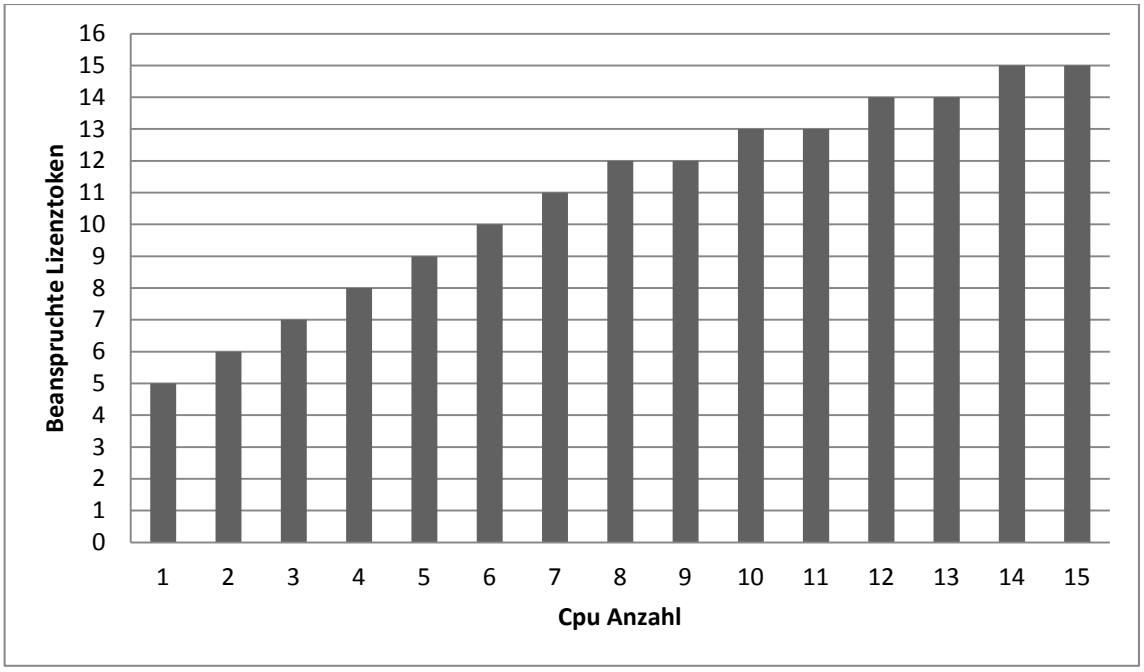


Abbildung 9: Verhältnis der Tokenanzahl zur genutzten CPU Anzahl

2.5.2 Schnittstelle

Zum Abfragen der aktuellen Lizenzauslastung wird das Tool *lmutils* genutzt, da leider keine Schnittstelle direkt auf dem Lizenzserver implementiert ist. Mit diesem Tool kann man die Funktion *lmstat* aufrufen, welche Informationen zur Lizenzauslastung liefert. Man erhält Daten zu den genutzten und insgesamt verfügbaren Lizenzen bzw. Tokens. Die Simulationen werden durch den Nutzer der sie gestartet hat, dem Server auf dem sie laufen, dem Zeitpunkt an dem die Token reserviert wurden und der Anzahl der genutzten Tokens beschrieben.

Es ist nicht geplant, in die Lizenzverwaltung Eingriff zu nehmen. Da nicht nur Abaqus Lizenzen durch den Lizenzmanager verwaltet werden, sondern auch die Lizenzen anderer Programme, bedeutet eine Änderung der Lizenzverwaltung einen Mehraufwand in der späteren Administration. So wird die Lizenzverwaltung dem geschlossenen System von Abaqus und *FLEXnet* überlassen, der zu entwickelnde Dienst nutzt diese nur zum Informationsgewinn.

2.6 Dateiserver

2.6.1 Architektur und Zugriff

Die Simulationsdateien werden auf einem zentralen Dateiserver abgelegt. Die Nutzer mit der entsprechenden Zugangsberechtigung besitzen ein *Home-V*erzeichnis auf diesem Server. In diesem Verzeichnis können Ordner erstellt und Dateien abgelegt werden. Der Zugriff erfolgt bequem über den Windows-Explorer, da das eigene Verzeichnis als Netzlaufwerk eingebunden ist. Die Simulationsserver besitzen ebenso den Zugriff zu dem Dateiserver.

2.6.2 Schnittstelle

Um den Status einer Simulation festzustellen ist es unabdingbar auf die einzelnen Nutzerverzeichnisse, des Dateiservers zuzugreifen. Der Zugriff der Nutzer über die Konsole wird durch die *Secure Shell (SSH)* ermöglicht. Über diese Schnittstelle können die Dateiinhalte abgefragt werden. Da der zu entwickelnde Dienst für das Senden der Simulationsaufträge die *SSH* Schlüssel des Nutzers benötigt, können diese hier ebenso für den Zugriff auf die *Home* Verzeichnisse genutzt werden. Der Dienst benötigt so keine übergreifende Administrationsrechte. So kann er auch nur auf die Verzeichnisse der Nutzer zugreifen, welche den Dienst aktiv nutzen.

2.7 Nutzerprogramm

Das Nutzerprogramm ist die Schnittstelle zwischen dem Benutzer und dem Dienst. Das Programm ersetzt das manuelle Starten der Simulationen über die Kommandozeile. Die Programmnutzung ist obligatorisch, um Schedulingverfahren zu ermöglichen.

2.7.1 Funktionale Anforderungen

Das Programm soll mit einer grafischen Oberfläche den Nutzer im Umgang mit den Abaqus Simulationen unterstützen. Der Nutzer möchte sich die Simulationen anzeigen lassen können. In der Anzeige sollen verschiedene Informationen dargestellt werden, wie zum Beispiel der Besitzer der Simulation, der Name der Eingabe Datei, die Priorität, die Laufzeit und den Status der Simulation.
Des Weiteren soll der Nutzer die Möglichkeit besitzen, Simulationen zu starten, in dem er eine Eingabe Datei auswählt, welche sich auf dem Dateiserver befindet. Weiterhin soll der Benutzer der Simulation eine Priorität und den Typ zuweisen können. Diese Simulation wird anschließend übertragen und durch den zu entwickelnden Dienst verwaltet. Derzeit aktive Simulationen sollen bearbeitet werden können. Man muss die Priorität verändern können, um die Schedulingreihenfolge zu beeinflussen und man soll die Simulation löschen können, falls man das Ergebnis nicht mehr benötigt.

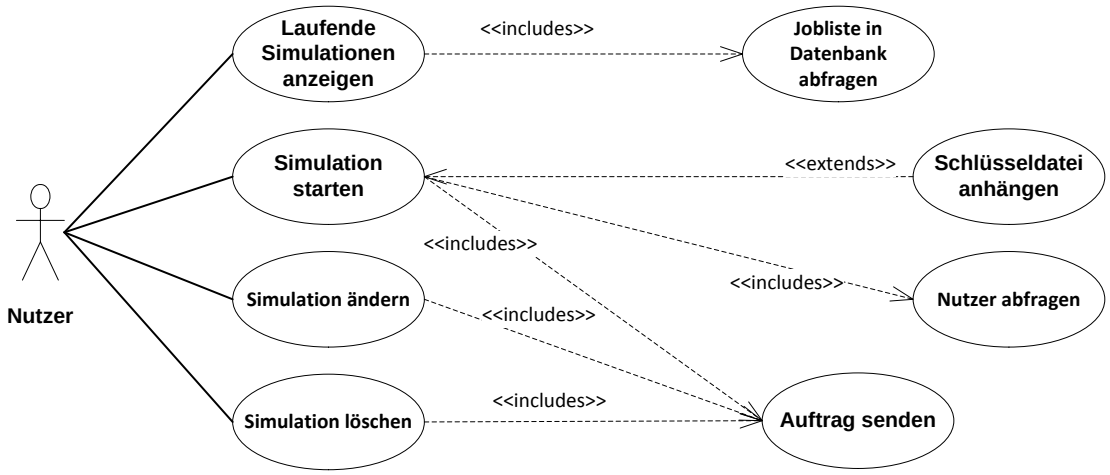


Abbildung 10: Anwendungsfalldiagramm des Nutzers

Ziel ist es, in einer Konfigurationsdatei verschiedene Parameter zu speichern, welche die Verbindungseinstellungen klassifizieren. So ist es unkompliziert möglich, auf sich ändernde IP und Port Nummern reagieren zu können, ohne das Programm neu erstellen zu müssen.

2.7.2 Nichtfunktionale Anforderungen

Das Programm soll primär auf Windows Systemen zum Einsatz kommen. Es wird die Programmiersprache C# und das Framework .NET verwendet. Eine weitere Bibliothek zur Abfrage der MySQL-Datenbank wird ebenso verwendet. Durch die Verwendung des Mono Frameworks ist eine reduzierte Oberfläche für Linux und Mac Systeme zu entwickeln. Die Funktionen bleiben identisch.

Der Nutzer soll schnell den Umgang mit dem Programm erlernen. Zur Unterstützung wird eine Benutzerdokumentation erstellt, welche die Programmnutzung und allgemein die Arbeitsweise erläutert.

Das Programm verfügt über eine Anzeige über den Verbindungszustand zum Dienst, sowie über eine Fehlerausgabe, falls die Verbindung nicht hergestellt werden konnte. Die Antwortzeiten liegen im Millisekundenbereich, so dass eine effiziente Arbeitsweise möglich ist.

2.7.3 Schnittstellen

Das Nutzerprogramm kommuniziert nicht direkt mit den Simulationsservern. Es fragt Daten aus einer MySQL-Datenbank ab. Durch die Datenbank bleibt die Abfrage selbst bei einer hohen Abfrageanzahl sehr performant. Die Simulations- und Änderungsaufträge werden als strukturierte XML-Objekte über eine TCP Datenverbindung an den Dienst gesendet. Diese Daten werden zuerst auf Clientseite validiert und vom Dienst noch einmal geprüft. Erst dann werden Änderungen in der Datenbank vorgenommen. Der Nutzer besitzt zudem keine

Möglichkeit, direkt Änderungen in der Datenbank vorzunehmen, da er nur lesenden Zugriff besitzt.

2.8 Dienstprogramm

Der Dienst reagiert einerseits auf Verbindungsanfragen der Nutzer und führt das Scheduling aus. Es ist der Kern des zu entwickelnden Software Systems.

2.8.1 Funktionale Anforderungen

Der Dienst teilt sich in verschiedene Komponenten auf, welche unterschiedliche Eigenschaften und Aufgaben haben. Der Dienst soll auf Verbindungsanfragen der Nutzer reagieren und führt das Scheduling aus. Um diese Funktionen zu realisieren stellt er Datenbankabfragen und modifiziert diese, fragt die Lizenznutzung ab, er stellt SSH Verbindungen her und liest den Inhalt von Dateien aus. Variable Informationen werden in einer Konfigurationsdatei abgespeichert, welche bei Programmstart eingelesen wird.

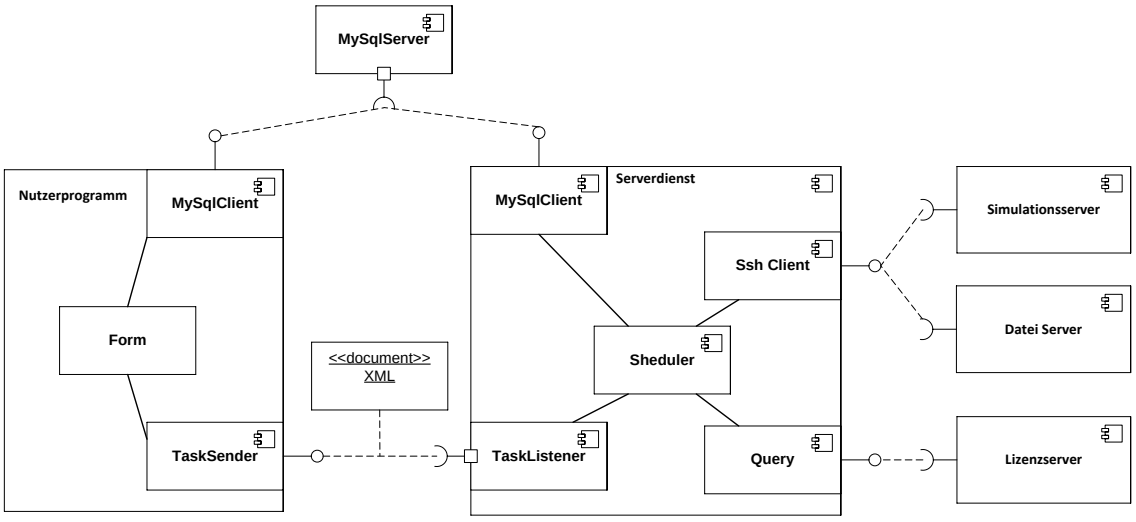


Abbildung 11: Komponentendiagramm des zu entwickelnden Softwaresystems

Ein *TaskListener* lauscht auf einem festgelegten Port bis sich ein Nutzer verbindet und nimmt das vom Nutzer gesendete XML Objekt entgegen. Er sendet dem Nutzer eine Rückantwort wenn das Objekt erfolgreich angekommen ist. Nachfolgend wird geprüft, ob es sich um ein Simulationsauftrag oder ein Änderungswunsch handelt. Die Datenbank wird mit den neuen Daten aktualisiert.

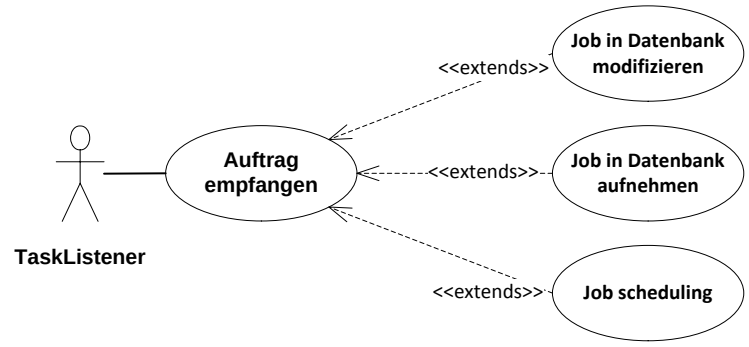


Abbildung 12: Anwendungsfalldiagramm des TaskListeners

Ein *MySQLClient* realisiert die Datenbankzugriffe und Änderungswünsche. Dabei verifiziert er die geplanten Änderungen und führt, falls nötig, weitere Änderungen am Datenbestand aus. So fügt er beim Starten einer Simulation eines unbekannten Nutzers nicht nur die Simulation der Datenbanktabelle an, sondern legt auch einen neuen Nutzerdatensatz in der Datenbank ab.

Ein *SshClient* verwaltet sämtliche Zugriffe, welche das SSH Protokoll nutzen. Fall es zu einer Umstellung oder Änderung des Protokolls kommt, können alle Anpassungen in dieser Komponente durchgeführt werden. Diese Komponente realisiert sämtliche Funktionen um eine Simulation zu starten, zu stoppen, zu pausieren und fortzusetzten. Zusätzlich werden eine Auslastungsabfrage, eine Dateiauslese, sowie ein Remote-Procedure-Call, ein entfernter Funktionsaufruf, implementiert.

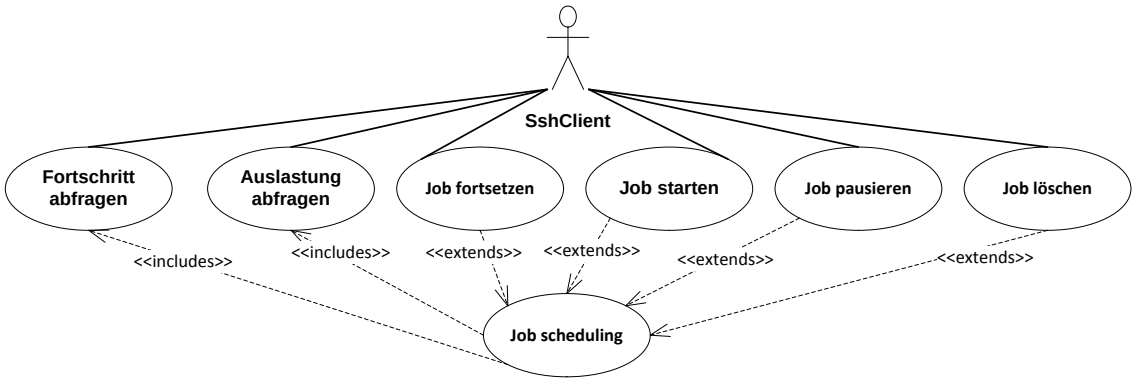


Abbildung 13: Anwendungsfalldiagramm des SshClienten

Mit Hilfe eines Tools zur Abfrage der Lizenzsoftware wird der aktuelle Status der Lizenzerteilung ermittelt. Daraus kann man Rückschlüsse ziehen, ob eine Simulation zurzeit aktiv ist oder aus einem nicht näher bekannten Grund beendet wurde. Der Grund lässt sich durch die Abfrage der Status und Logdateien genauer feststellen.

Das Scheduling soll durch zwei verschiedene Algorithmen erfolgen. Einerseits durch das prioritätenbasierende Scheduling und durch das prioritätenbasierende *Round-Robin*-Scheduling. In der Konfigurationsdatei werden die Parameter dazu festgelegt.

2.8.2 Nichtfunktionale Anforderungen

Durch die Vielzahl von zu implementierenden Funktionen ist es nötig, dass auf einen modularen Aufbau des Systems geachtet wird. Die einzelnen Komponenten müssen gut dokumentiert werden, damit Modifizierungen auf Grund von Änderungen des Softwaresystems schnell realisiert werden können.

Auch der Dienst wird in C# mit dem .Net Framework für Windows entwickelt. Zusätzlich werden Bibliotheken zur Datenbankabfrage und die SSH-Sitzungen verwendet.

Das Scheduling soll in einem in der Konfigurationsdatei festgelegten Zeitraum erfolgen. Während des Schedulings wird die Bearbeitung der Nutzeranfragen verhindert, um inkonsistente Datenbestände zu vermeiden.

3 Entwurf

Zu Beginn des Entwurfs wird eine geeignete Systemarchitektur gewählt, auf welcher die Analysekomponenten abgebildet werden. Durch den prototypischen Entwicklungsansatz geht die Entwurfsphase mit ständiger Analyse und Testen einher. Die Entwürfe der Einzelkomponenten werden auf ihre Realisierbarkeit geprüft und detaillierter ausformuliert, bis die gewünschte Funktionalität realisiert ist.

3.1 Entwurfsüberblick

Im Ergebnis der Analyse wird eine Client-Server Architektur notwendig. Diese stellt die beste Möglichkeit dar, Aufträge verschiedener Nutzer zu bearbeiten und gleichzeitig das Scheduling, ohne umfassende Administrationsrechte zu benötigen, auszuführen. Es werden schon vorhandene Kommunikationswege zwischen den Komponenten genutzt. So müssen folglich keine fehleranfälligen Änderungen am vorhandenen Softwaresystem durchgeführt werden. So ist das Risiko sehr gering, Einschränkungen im laufenden Betrieb durch die zu entwickelnde Software zu erhalten.

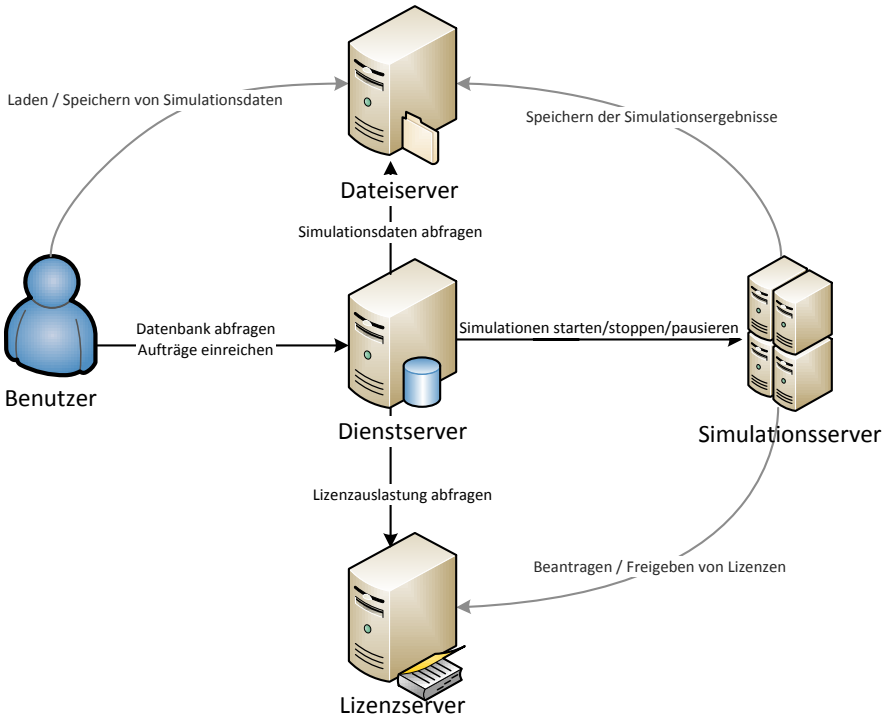


Abbildung 14: Übersicht des geplanten Softwaresystems

Das Nutzerprogramm stellt dabei den Clienten dar. Mit dem Programm fragt der Benutzer die Informationen der Simulationen aus der Datenbank ab. Ebenso werden an den

Dienstserver Simulationsaufträge gesendet. Dabei wird werden die Anforderungen asynchron geschickt, wodurch die Oberfläche des Nutzerprogramms für die Dauer der Kommunikation nicht blockiert [8]. Falls der Benutzer noch nicht in der Datenbank existiert, muss seine Schlüsseldatei mit gesendet werden, um Simulationsaufträge in seinen Namen starten zu können. Anschließend wird das Objekt an den Server gesendet und die Serverantwort auf der Oberfläche ausgegeben.

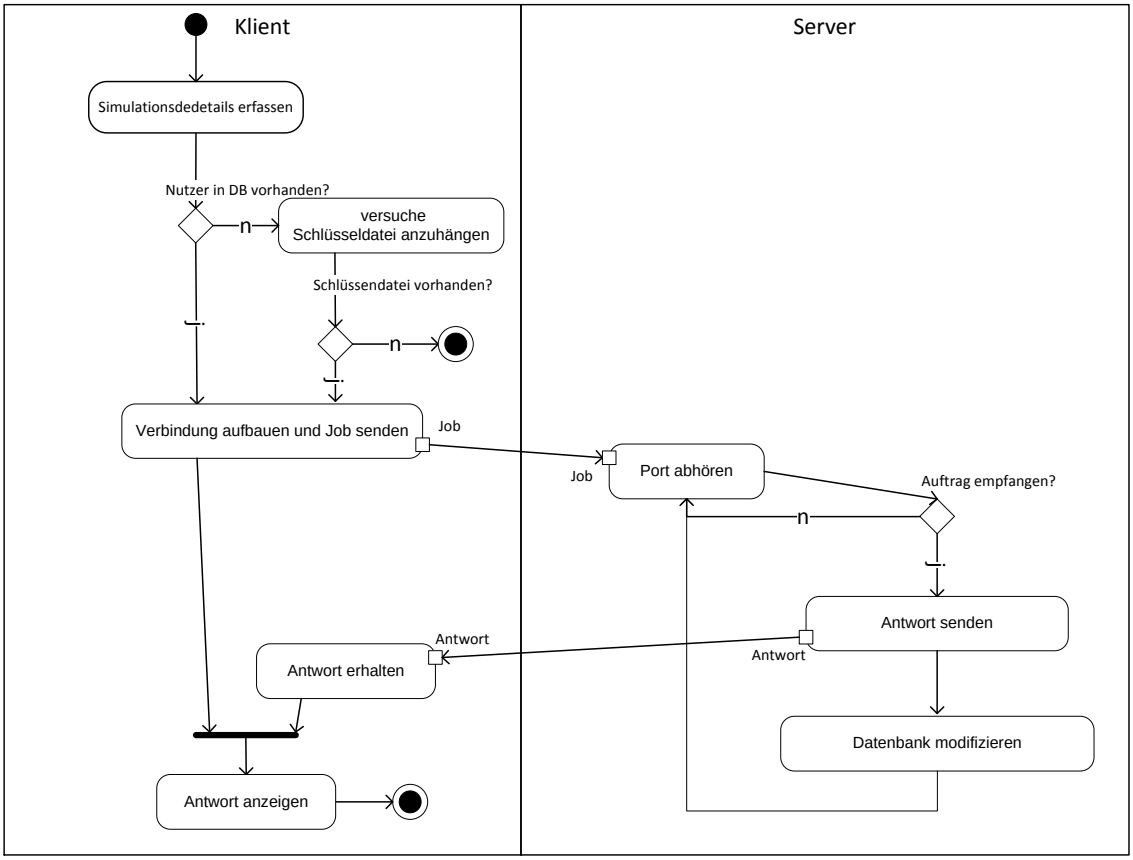


Abbildung 15: Aktivitätsdiagramm Senden einer Simulation

Der Dienst nimmt dabei die Aufträge des Nutzers entgegen und führt zeitgesteuert das Scheduling aus. Dabei werden zum Start des Dienstes die Parameter aus einer Konfigurationsdatei geladen. Nach diesen Parametern wird *TCP-Listene*r gestartet. Verbindet sich ein Nutzer, wird der Port an diesen gebunden und das zeitliche Scheduling wird ausgesetzt. Der Auftrag des Nutzers wird entgegen genommen und eine Empfangsbestätigung wird an diesen zurück gesendet. Nachfolgend wird der Auftrag bearbeitet und das zeitgesteuerte Scheduling wieder aktiviert. Der *TCP-Listener* wird wieder gestartet, bis der nächste Nutzer sich verbindet.

3.2 Kommunikationskomponenten

Die Grundidee der Kapselung von Funktionalitäten in Komponenten ist, wiederverwendbare und leicht zu wartende Softwarebestandteile zu erstellen. Dabei können Details der Komponenten geändert werden, ohne die Funktionalität des Gesamtsystems zu verlieren.

3.2.1 Client– Dienst Kommunikation

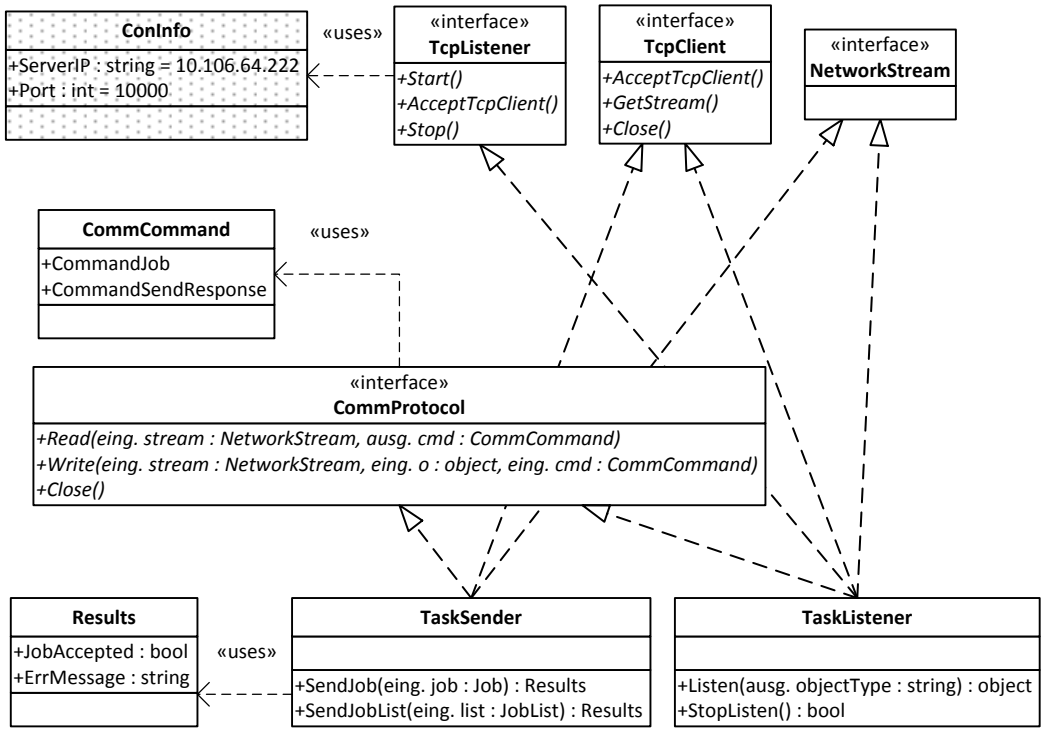


Abbildung 16: Klassendiagramm der TCP Übertragung

Die Daten, welche zwischen dem Nutzerprogramm und dem Dienst ausgetauscht werden, sind kleine Objekte. Es werden lediglich Informationen zu den Simulationen und dem Nutzer ausgetauscht. Der Datenverkehr wird auf wenige Kilobyte beschränkt.
Um eine fehlerfreie und vollständige Übertragung der Daten sicherzustellen wird auf das Transport Protokoll *Transmission Control Protocol (TCP)* gesetzt. Das TCP Protokoll setzt auf dem leitungsvermittelnden *Internet Protokoll (IP)* auf. Um die Datenobjekte strukturiert versenden zu können werden diese mittels der *Extensible Markup Language (XML)* strukturiert serialisiert an den TCP Datenstrom übergeben. Dabei wird ein *Header* der XML Nachricht voran gestellt. Dieser *Header* enthält die Länge der XML Nachricht und den serialisierten Objekttyp. Zwei Klassen, *TaskSender* und *TaskListener*, verwalten das Senden und Empfangen der Daten indem die benötigten TCP Verbindungsschnittstellen des .NET Frameworks an die Klasse *CommProtokoll* übergeben werden. Die Klasse *CommProtocol* serialisiert die Objekte als XML strukturierten Datenstrom.

In Absprache mit dem IT-Sicherheitsbeauftragten des Fraunhofer Instituts für Werkstoffmechanik wurde abgewogen, in wie fern die übertragenen Informationen an den Dienst schützenswert sind. Da die Dienstfunktionalitäten nur im Intranet zur Verfügung stehen, ist davon auszugehen, dass die Gefahr, dass die Kommunikation an der Netzwerkinfrastruktur abgefangen wird, vergleichsweise gering ist. Selbst wenn die Daten abgefangen und geändert werden würden, könnte kein übermäßiger Schaden entstehen, da ein potentieller Angreifer maximal Simulationen abbrechen könnte.
Falls der Wunsch besteht den Dienst über eine sichere verschlüsselte Verbindung zugänglich zu machen, können die Nutzinformationen durch ein weiteres Protokoll wie die *Secure Shell (SSH)* oder *Transport Layer Security (TLS)* verschlüsselt werden. Durch den komponentenbasierenden Aufbau ist dies auch nachträglich möglich.

3.2.2 Secure Shell Kommunikation

Die Verbindung zu den Simulations- und Dateiservern wird durch die *Secure Shell (SSH)* Sicherungsschicht hergestellt. Das SSH Protokoll setzt auf der *TCP/IP* Protokollfamilie auf. Es stellt die Vertraulichkeit, Integrität und Authentizität der zu übertragenden Informationen sicher [9]. Dabei muss sich der Nutzer zuerst beim Server authentifizieren, indem er eine Signatur an den Server sendet, welche mit seinem privaten Schlüssel zu erstellen ist. Der Server prüft in Folge mit dem öffentlichen Schlüssel des Nutzers ob die Signatur korrekt ist [7]. Anschließend beginnt der Schlüsselaustausch nach *Diffie-Hellman* zur Vereinbarung eines geheimen Schlüssels für die nachfolgende Datenübertragung [10].

Diese Authentifizierung und Verschlüsselung benötigt folglich den Namen eines Nutzers und seinen privaten Schlüssel. Der Dienst muss sich als der Nutzer der Simulation auf dem Simulationsserver authentifizieren, um die Simulation zu starten. Folglich werden die privaten Schlüssel auf dem Dienstserver benötigt. Diese können manuell auf dem Dienstserver abgelegt werden oder automatisiert durch das Nutzerprogramm geschickt werden. Das automatisierte Senden erfolgt nur, wenn der Nutzer noch nicht in der Datenbank angelegt wurde und seine Schlüsseldatei noch nicht auf dem Server existiert. Der Dateiname dabei als Hashwert der Datei umbenannt, so dass nicht klar erkennbar ist, welche Schlüssel Datei welchem Nutzer zuzuordnen ist. Die Zuordnung der Hash-Werte zum Nutzernamen wird in der Datenbank gespeichert.

Das Senden der privaten Schlüssel, bei erstmaliger Nutzung des Dienstes stellt ein Sicherheitsrisiko dar, welches genannt werden muss. Wenn ein Angreifer den Schlüssel abfängt, könnte sich dieser auf einem Simulationsserver mit dem Schlüssel authentifizieren. Wie beschrieben, können die Schlüssel manuell durch den Administrator auf dem Dienstserver hinterlegt werden. Dadurch werden die Sicherheit und der Administrationsaufwand erhöht. Es bleibt dem verantwortlichen Systemadministrator überlassen, wie er mit der Verwaltung und der Einschätzung der Sicherheitsgefährdung weiter vorgeht.

Für die SSH Verbindung wird die Bibliothek *SharpSSH* für .NET genutzt, welche einige Wrapper-Klassen enthält um die SSH-Kommunikation zu abstrahieren. So wird nur der Benutzername, die SSH-Schlüsseldatei des Nutzers, die Server IP und der auszuführende Befehl benötigt. Es handelt sich bei dem auszuführenden Befehl um einen *Shell Execute* Befehl.

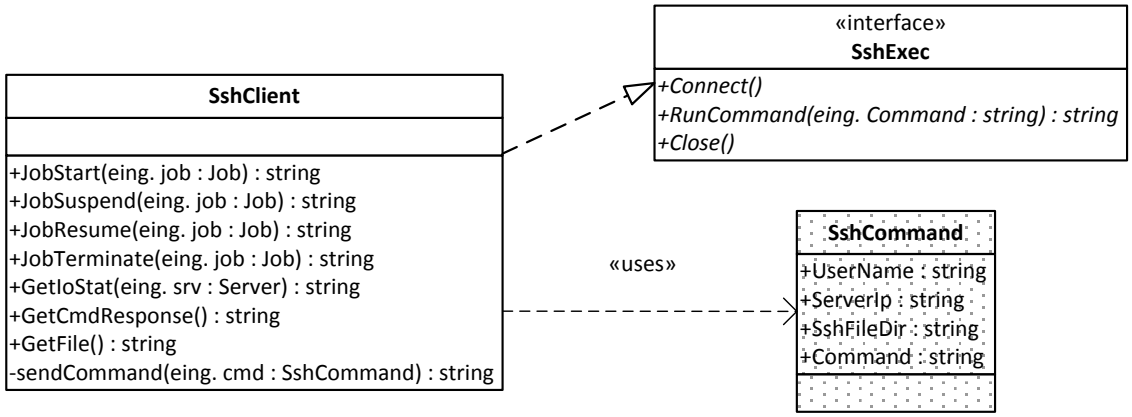


Abbildung 17: Klassendiagramm der SshClient Komponente

Die Klasse *SshClient* wird verwendet, um die Befehle vorzubereiten. Die Informationen aus dem Job werden genutzt, um die Klasse *SshCommand* mit Informationen zu befüllen. Das Interface *SshExecute* stellt die Wrapper-Klasse für die *SSH* Verbindung dar. Die *SSH* Kommunikation wird ebenso genutzt, um die Auslastung der Simulationsserver sowie einzelne Dateien auf dem Dateiserver abzufragen.

3.2.3 Lizenzstatusabfrage

Die Lizenzen werden durch den *FlexLM*-Lizenzserver verwaltet. Da leider keine Programmierschnittstelle (API) existiert, um die Lizenzverteilung abzufragen, muss das Tool *lmutil* genutzt werden.

Das Programm *lmutil* kann mit bestimmten Parametern aufgerufen werden, um den Lizenzstatus abzufragen. Dabei muss man die erhaltene Textdatei nach bestimmten Informationen parsen. Man benutzt dabei den Parameter `lmstat` um den Status aller Lizenzaktivitäten abzurufen. Mit dem Parameter `-c [port]@[host]` grenzt man die Ausgabe auf eine bestimmte Lizenzdatei auf dem angegebenen Server ein. Durch `-f abaqus` erhält man nur die Information zu Abaqus, denn mehr Informationen werden nicht benötigt. Folgende Übersicht wird dadurch erzeugt.

```
  ABAQUSLM: UP v11.6
Feature usage info:

Users of abaqus:  (Total of 15 licenses issued;  Total of 5 licenses in use)

  "abaqus" v61.2, vendor: ABAQUSLM
  floating license

    richl born /dev/tty (v61.2) (srv-lizenz/27000 622), start Tue 5/28 16:46, 5
licenses
```

Abbildung 18: Abfrage der Lizenznutzung durch lmutil

Man kann der Übersicht diverse Informationen entnehmen. *15* Lizenzen sind insgesamt vorhanden, *5* davon sind in Benutzung. Der Nutzer *richl* hat mit einer Simulation auf dem Simulationsserver *born 5* Lizenzen beantragt. Die Lizenzen wurden am *Dienstag den 28.05 um 16:46* vergeben. Die Weiteren Informationen sind zur Identifikation eines Jobs nicht von Interesse. Beim Parsen der Informationen muss darauf geachtet werden, dass das US amerikanische Zeitformat genutzt wird.

Falls es dazu kommt, dass keine Lizenzen mehr vorhanden sind, aber dennoch Jobs gestartet wurden, werden diese in einer Warteliste in der Übersicht dargestellt. Da die Warteliste von Abaqus nach dem *First-Come First-Served* Prinzip abgearbeitet wird, sollte dieser Zustand vermieden werden, da das Scheduling durch einen intelligenteren Algorithmus vom Dienst verwaltet werden soll.
Falls dennoch Simulationen existieren, welche in einer Warteliste stehen ist ein Fehler aufgetreten, oder ein Nutzer hat direkt Simulationen auf dem Simulationsserver gestartet. Um den Normalzustand wieder her zu stellen, wird das Scheduling so lange ausgesetzt bis keine Warteschlange mehr vorhanden ist.

3.3 Datenmanagement

Die persistente Datenhaltung teilt sich in zwei Systeme auf. Einerseits müssen Simulationseigenschaften, Nutzer und Server schnell abrufbar abgespeichert werden und andererseits müssen schützenswerte Daten lokal, jedoch leicht modifizierbar gesichert werden.

Die Daten, welchen keine besonderen Schutz bedürfen, werden in einer MySQL Datenbank gespeichert. MySQL wird genutzt, da dieses Datenbanksystem weit verbreitet ist und frei erhältlich ist. Ebenso existiert eine Verbindungsbibliothek für C#. Der Datenbankzugriff wird durch zwei verschiedene Benutzer realisiert. Der *Reader* erhält nur einen lesenden Zugriff auf die Datenbank. Dieser Benutzer wird für das Client Programm verwendet. Der Endnutzer darf in Kenntnis seiner Zugangsdaten zur Datenbank sein, da er direkt keine Änderungen an ihr vornehmen kann.
Der zweite Datenbanknutzer wird Service genannt und erhält lesenden und schreibenden Zugriff. Er darf alle nötigen Änderungen an den Datenbankinhalten vornehmen. Das Datenbankschema darf er jedoch auch nicht ändern.

Bei der Modellierung der Datenbank wurde darauf geachtet, eine sinnvolle und noch lesbare Abstraktion der realen Objekte zu erstellen. Die Normalisierung wurde unter dieser Bedingung durchgeführt. So wurden 4 Tabellen geschaffen, welche eine persistente Datenhaltung sicherstellen.

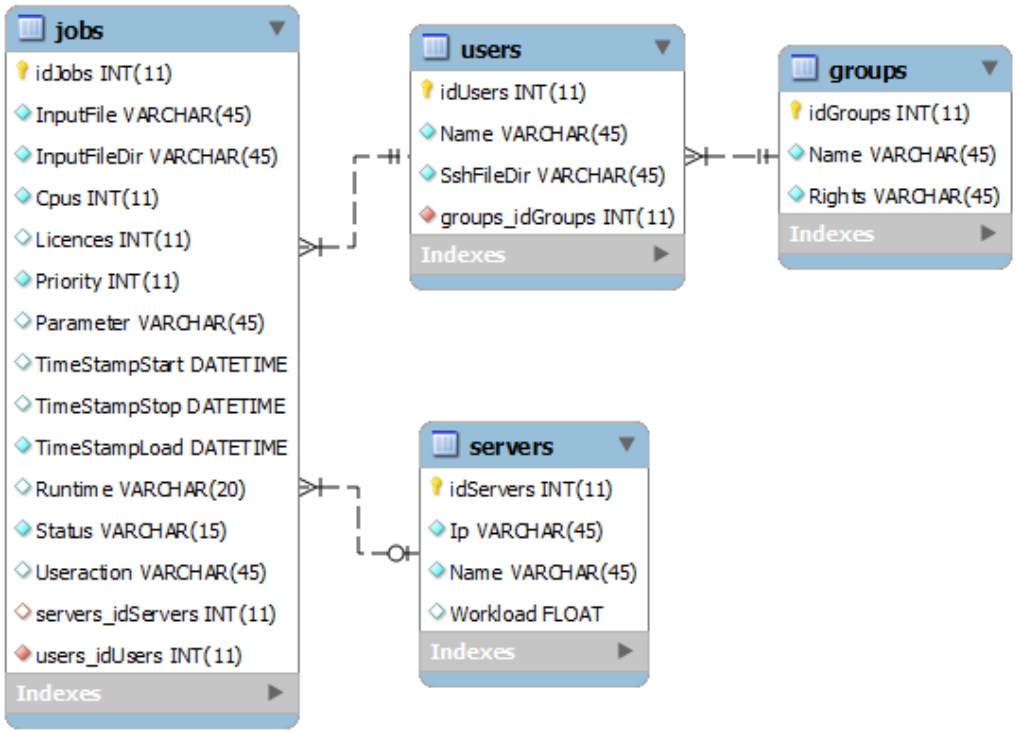


Abbildung 19: Entity-Relationship-Modell der Datenbank

Schützenswerte Daten stellen Verbindungsdaten, Nutzernamen, Passwörter und Schedulingparameter dar. Diese werden in einer Konfigurationsdatei abgespeichert. Diese Datei wird zum Programmstart von der *Common Language Runtime (CLR)* ausgewertet. Zur Laufzeit ist der Zugriff auf diese Datei ebenso möglich.
Die Anwendungskonfigurationsdatei befindet sich dabei im selben Verzeichnis wie die ausführbare Anwendungsdatei und besitzt die Endung *.config*. Die Datei ist in *XML* formatiert und damit mit einem einfachen Editorprogram schnell modifizierbar. Durch einen Neustart des Programms werden Änderungen in der *Config* eingelesen und umgesetzt. Benutzerunabhängige Daten werden dabei in der Kategorie *<applicationSettings>* gespeichert. Diese Daten gelten uneingeschränkt für jeden Nutzer der Anwendung.

3.4 Scheduling

Das Scheduling bildet das Kernstück des Dienstes. Es muss robust, effizient und fair arbeiten. Nach diesen Merkmalen wurden beide Schedulingalgorithmen entworfen.

3.4.1 Prioritätenbasierendes Scheduling

Das prioritätenbasierende Scheduling richtet sich strikt nach den durch den Nutzer vorgegebenen Prioritäten. Es existieren 2 Joblisten, in denen die einzelnen Jobs gespeichert werden. Eine Liste speichert die derzeit aktiven Jobs, eine weitere alle wartenden Jobs. Beendete oder abgebrochene Jobs werden aus der Liste gelöscht. Im Gegensatz zum entwickelten *Round-Robin*-Scheduling ist das prioritätenbasierende Scheduling ebenso dafür geeignet, mehrere Simulationen gleichzeitig rechnen zu lassen. Bei Auslieferung der Software wird jedoch aus Effizienzgründen darauf verzichtet.

Abbildung 20: Aktivitätsdiagramm Prioritätenbasierendes Scheduling

Zu Beginn des Schedulings wird eine Sicherungsvariable überprüft. Diese Variable verhindert, dass ein Scheduling einsetzt, wenn das voran gegangene Scheduling noch nicht abgeschlossen wird. Dies kann durchaus vorkommen falls die Abstände zwischen den Schedulingaufrufen zu kurz sind. Abaqus benötigt einige Zeit, um große Simulationen zu stoppen und zu starten. Diese Zeit wird im Scheduling abgewartet, um eine Rückmeldung von Abaqus zu erhalten, ob die Operation erfolgreich war. Ist die Sicherungsvariable gesetzt, wird das Scheduling abgebrochen.

Falls aktive Jobs in der aktiven Jobliste vorhanden sind, wird geprüft, ob diese mit den Jobs der Lizenzdatei übereinstimmen. Des Weiteren werden die Nutzeraktionen der aktiven Jobliste ausgeführt. Nutzeraktionen können das Ändern von Prioritäten sowie Löschanforderung einzelner Simulationen beinhalten.

Damit ist die Bearbeitung der aktiven Jobs beendet. Nun werden die Nutzeraktionen der wartenden Jobs durchgeführt und das eigentliche Scheduling eingeleitet, falls wartende Jobs existieren. Wenn im Anschluss keine aktiven Jobs existieren, wird der erst beste wartende Job aktiv gesetzt und die Datenbank aktualisiert. Existieren aktive Jobs muss genauer geprüft werden.

Dabei wird aus der aktiven Jobliste der schlechteste Job mit dem besten Job aus der Warteliste verglichen. Liegt die Priorität des wartenden Jobs dabei höher wird der derzeit aktive Job pausiert und der passive gestartet.
Durch diesen Vorgang ist es möglich auch in Zukunft ein Verfahren einzubauen, welches die Simulationen zur Laufzeit in ihrer Priorität verringert, um zu verhindern, dass alle Nutzer stets mit der höchsten Priorität arbeiten. Da dieser Mechanismus zurzeit nicht gefordert ist, wurde lediglich die Schnittstelle dazu implementiert.
Zum Ende des Scheduling wird die Sicherungsvariable wieder freigegeben.

3.4.2 Prioritätenbasierendes Round-Robin-Scheduling

Eine Erweiterung des prioritätenbasierenden Scheduling stellt dieser Algorithmus dar. Anstatt alle Simulationen strikt nach ihrem Prioritäten ablaufen zu lassen, werden nun Zeitscheiben verwendet zu denen die Simulationen gewechselt werden. Dadurch kann ein einzelner Nutzer nicht mehr alle anderen Nutzer durch eine Simulation blockieren.

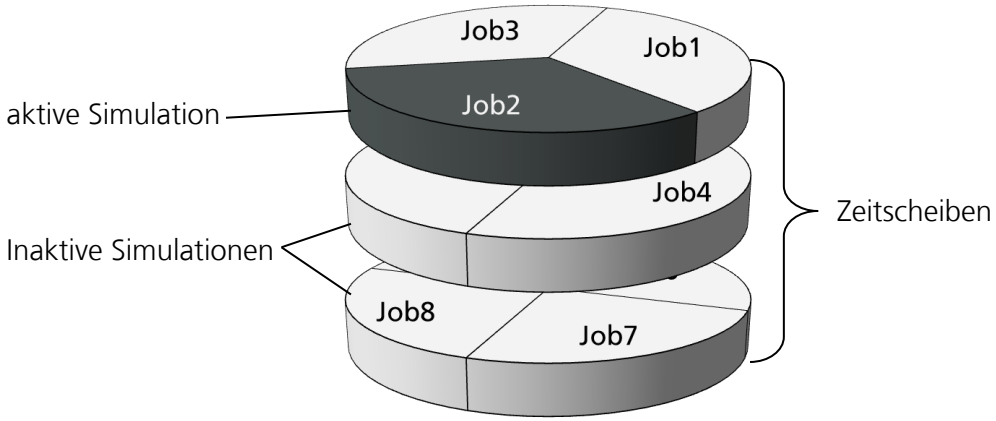


Abbildung 21: Veranschaulichung der Verwaltung der Simulationen in mehreren Zeitscheiben entsprechend ihrer Priorität

Man kann so von einem garantierten Scheduling innerhalb einer Prioritätsklasse sprechen. Da die Jobwechsel ebenfalls eine bestimmte Zeit benötigen, werden die langen Jobs einer niedrigen Priorität seltener gewechselt als die kurzen Jobs der hohen Priorität.

Das *Round-Robin*-Scheduling in Prioritätsklassen benötigt durch den höheren Verwaltungsaufwand ebenso mehr Klassen und Funktionen. An statt einer gemeinsamen Warteliste für alle Simulation ist nun für jede Priorität eine Liste vorhanden. Eine Liste für den aktiven Job bleibt dabei bestehen.

Um zu verhindern, dass gleichzeitig das Scheduling ausgeführt wird, nutzt man wieder eine Sicherheitsvariable. Das Scheduling wird erst gestartet, wenn die Sicherungsvariable frei, bzw. nicht gesetzt ist. Startet das Scheduling wird im ersten Schritt die Variable zur Sicherung gesetzt. Falls die aktive Jobliste nicht leer ist, wird der aktive Job mit den Lizenzinformationen abgeglichen und die Nutzeraktionen werden ausgeführt. Weiterhin werden die Nutzeraktionen in den einzelnen Wartelisten durchgeführt, falls sich Jobs in diesen befinden.

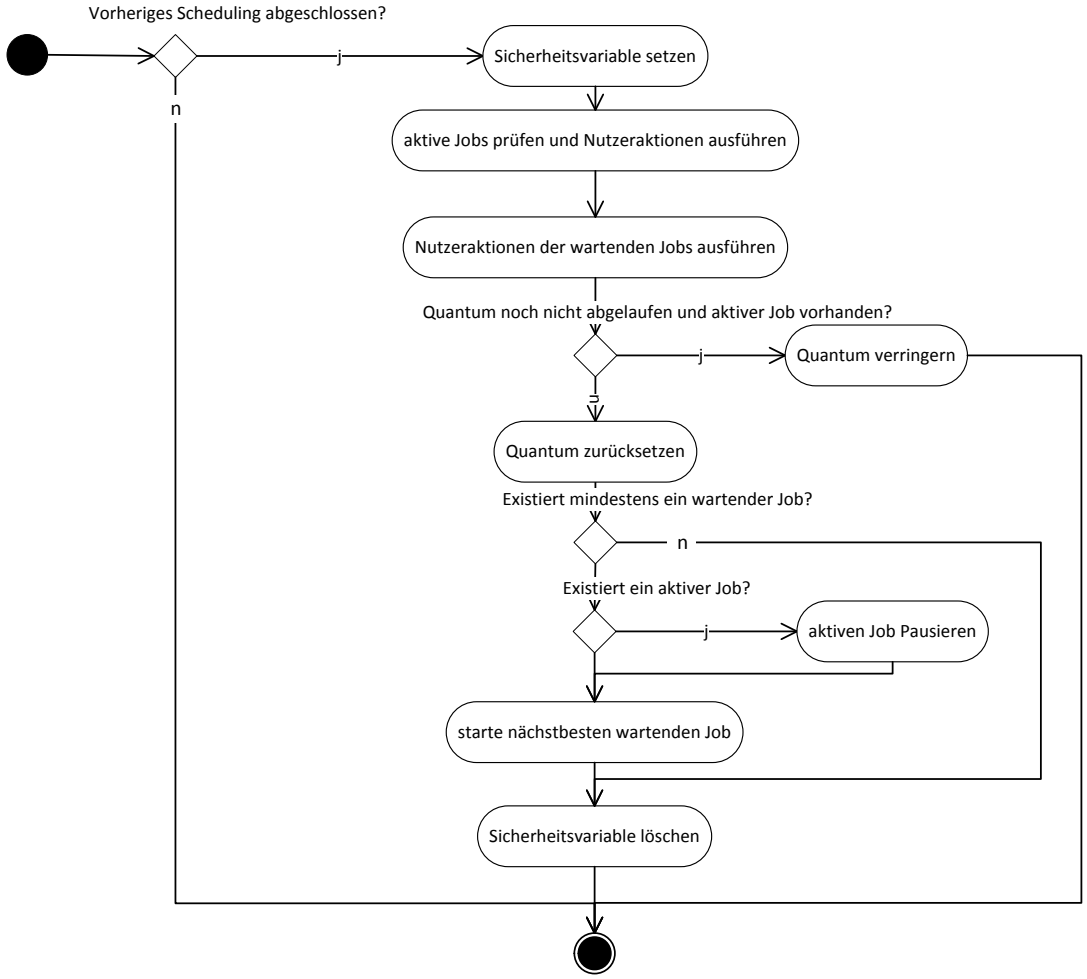


Abbildung 22: Aktivitätsdiagramm Prioritätenbasierendes Round Robin Scheduling

Wenn ein aktiver Job existiert und sein Quantum noch nicht verbraucht ist, kommt es nicht zu einem Prozesswechsel, stattdessen wird nur das Quantum dekrementiert. Trifft eine der beiden Bedingungen nicht zu, kommt es zum Prozesswechsel. Dabei wird zuerst die Sicherungsvariable gesetzt, um paralleles Scheduling zu vermeiden. Im Anschluss wird die Zeitscheibe, bzw. das Quantum zurückgesetzt. Das in der Konfigurationsdatei festgelegte Quantum wird mit der Prioritätsklasse des nächst besten Jobs multipliziert. Die niedrigste Klasse besitzt die Priorität fünf. Bei einer Zeitscheibengröße von drei Schedulingaufrufen, würde dies bedeuten, dass ein Job mit Priorität fünf genau 15 Schedulingaufrufe rechnen wird. Ein Job mit Priorität eins würde drei Schedulingaufrufe aktiv sein, bis er gewechselt werden würde. Ist die Zeitscheibengröße gesetzt, wird der derzeitig aktive Job, sofern ein Job aktiv ist, pausiert und der nächstbeste Job gestartet.
Zum Abschluss des Scheduling wird wieder die Sicherungsvariable für das nächste Scheduling freigegeben.

4 Realisierung

4.1 Nutzerprogramm

Die Oberfläche wird mittels der Programmierschnittstelle *Windows.Forms* des *.NET* Frameworks erzeugt. Ein Formular stellt in *Windows.Forms* eine Oberfläche zur Anzeige von Informationen dar.

In diesem Formular werden Steuerelemente eingefügt, welche Informationen anzeigen oder entgegen nehmen. Die Steuerelemente reagieren auf Ereignisse durch bestimmte *Ereignishandler* (engl. *Eventhandler*). So kann mit einem *MouseClickEreignishandler* ein Mausklick auf ein bestimmtes Element abgefangen und eine bestimmte Funktion aufgerufen werden. Durch die Verwendung einer *BindingSource*, einer Datenquelle zum Anbinden von Daten, können Datenbankinhalte dem Nutzer in einer Tabelle, einer DataGridView, angezeigt werden. Um zu Verhindern, dass zeitaufwendige Prozesse keinen Einfluss auf die Reaktionsfähigkeit der grafischen Oberfläche nehmen, können *BackgroundWorker*Klassen verwendet werden, welche Vorgänge im Hintergrund ausführen. [11]

4.1.1 Oberflächengestaltung

Das Nutzerprogramm soll 3 Kernfunktionalitäten erfüllen. Es soll dem Nutzer einen Überblick über die derzeitigen Simulationen verschaffen. Die Simulationen sollen mit detaillieren Eigenschaften über Status, Laufzeit, Eingabedateiname, Nutzer usw. beschrieben werden. Des Weiteren soll der Nutzer weitere Simulationen starten können, in dem er die Eingabedatei auswählt, eine Priorität wählt und den Typ der Simulation angibt. Als 3. Hauptfunktion wurde das Ändern der Simulation gefordert. Dabei ist es möglich, die Priorität zu verändern und die Abarbeitung der Simulation abzubrechen.

Das Layout der Form ist drei-teilig gegliedert. In der oberen Zeile wird ein *ToolStrip* zur Anzeige des *ToolStripButtons* „Aktualisieren“ genutzt. Dieser Button bewirkt, dass die Daten aus der MySQL Datenbank neu eingelesen und angezeigt werden. Eine abgeleitete Klasse des *Toolstrip* stellt der *StatusStrip* am unteren Rand der Form dar. In dieser Statusleiste werden Hilfsinformationen zu bestimmten Elementen der Oberfläche angezeigt, wenn man mit dem Mauszeiger über diese fährt.

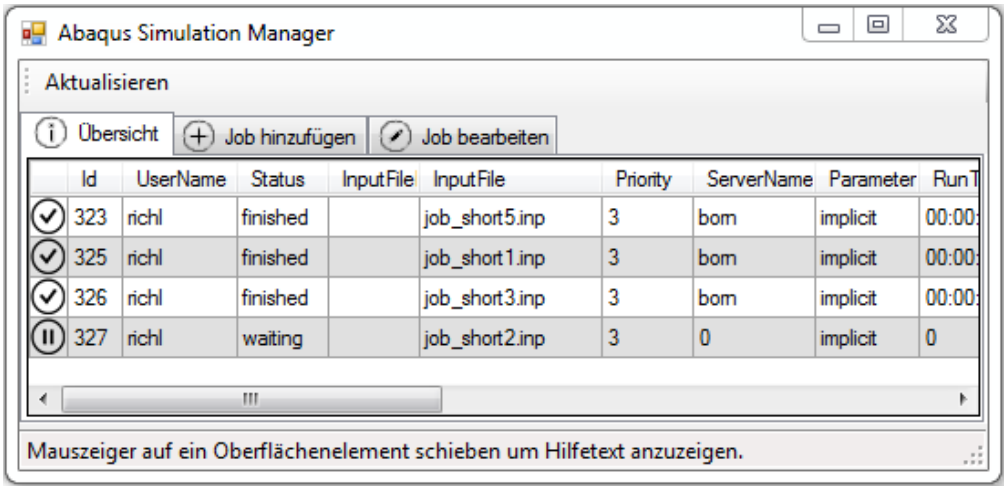


Abbildung 23: Oberfläche der Übersichtsanzeige

Mittig in der Form befindet sich das *TabControl* mit den drei Registerkarten zur Übersicht, zum Hinzufügen eines Jobs und zum Bearbeiten eines Jobs. Die einzelnen Registerkarten werden als *TabPage* Objekte dargestellt. Durch den Klick auf die Registerkartenüberschrift wird automatisch die zugehörige *TabPage* angezeigt.

Innerhalb der *TabPage* Übersicht werden die Daten der Simulationen in einer *DataGridView* angezeigt. Dies ist dabei in erster Linie eine benutzerdefinierbare Tabelle zum Anzeigen von Daten. Durch die Angabe einer *DataSource* Eigenschaft wird die *DataGridView* an eine Datenquelle gebunden und mit den Daten befüllt.

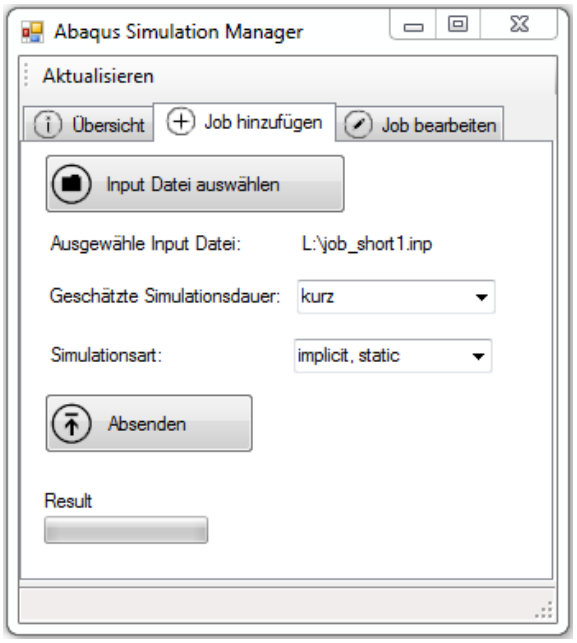


Abbildung 24: Oberfläche der Registerkarte "Job hinzufügen"

In der zweiten Registerkarte kann der Nutzer seine Simulation dem Scheduling hinzufügen. Zu Beginn muss der Nutzer eine Eingabe-Datei, auch Input-Datei genannt, vom Netzlaufwerk „L“ auswählen. Wählt er keine Eingabe Datei sondern eine andere aus oder befindet die Datei sich nicht auf dem Laufwerk L, verbleibt der Button „Absenden“ inaktiv und der Nutzer kann seine Simulation nicht an den Server senden. Bei der geschätzten Simulationsdauer wählt der Nutzer die Priorität. Aus psychologischen Gründen wurde darauf verzichtet, dem Nutzer direkt die Einschätzung der Priorität nach wichtig oder weniger wichtig zu überlassen. Kaum Jemand betrachtet seine eigene Arbeit als weniger wichtig als die Arbeit seiner Kollegen. So würden alle Simulation stets mit hoher Priorität gestartet werden.

Simulationen mit kurz geschätzter Dauer werden vorrangig ausgeführt. Bei der Simulationsart muss der Benutzer angeben, ob es sich bei der Simulation um eine statische oder dynamische Simulation handelt, da die Lizenzverteilung beider Simulationen verschieden ist.

Beim Klick des Absenden-Buttons wird der Kontakt zum Dienst über TCP hergestellt. Falls keine Verbindung entsteht, wird der Benutzer durch eine Fehlerausgabe darüber in Kenntnis gesetzt. Falls die Daten erfolgreich übertragen werden konnten, erhält der Nutzer eine Erfolgsmeldung.

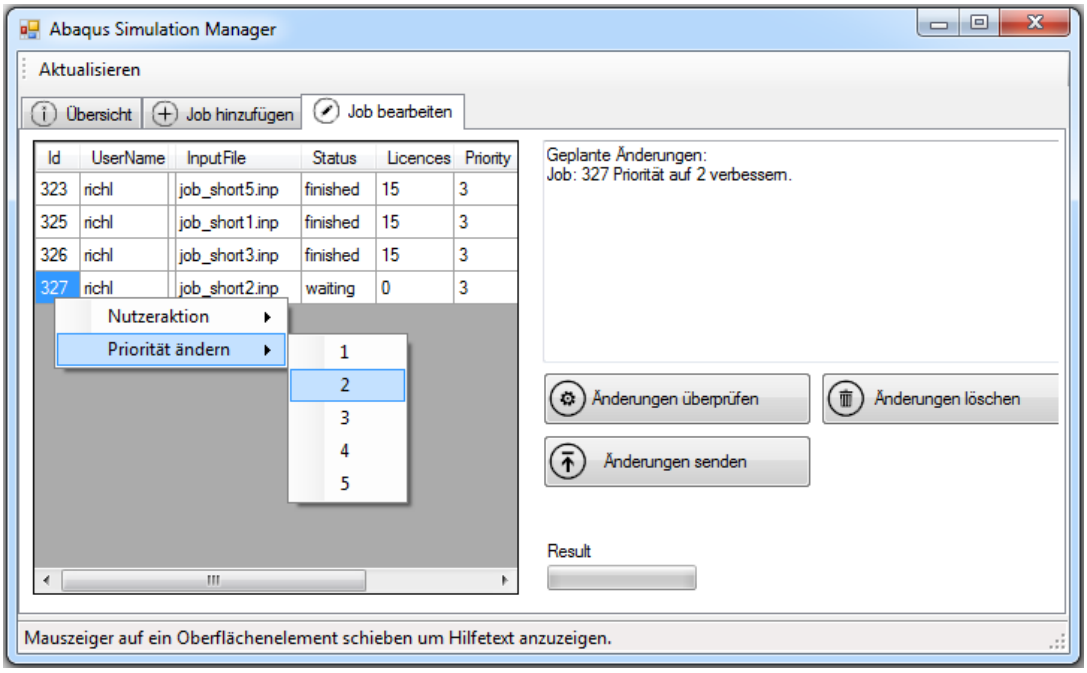


Abbildung 25: Oberfläche der Registerkarte "Job bearbeiten"

Die dritte Registerkarte bietet Werkzeuge zum Ändern der einzelnen Simulationen. Das zweispaltige Layout wird durch einen *SplitContainer* erzeugt. Dieses Steuerelement teilt den Bereich der *TabPage* in zwei Teile. Der Nutzer kann die Größe der Bereiche durch ziehen des

Mittelbalkens zwischen den Bereich ändern.
Im linken Bereich befindet sich wieder ein *DataGridView* in dem die Simulationen angezeigt werden. Durch einen Rechtsklick auf eine Zelle in der Tabelle wird ein *ContextMenueStrip*, ein Kontextmenü, an dieser Position aufgerufen. In dem Menü kann man verschiedene Änderungen an der Simulation der aktuellen Zeile vornehmen. So kann eine Lösch- oder Prioritätsänderung durchgeführt werden.
Im rechten Bereich des *SplitContainers* können die Änderungen anschließend überprüft werden. So werden Änderungen nur bei noch nicht beendeten Simulationen durchgeführt. Wenn mehrmals ein und dieselbe Simulation geändert wurde, wird nur die letzte Änderung berücksichtigt. Ist der Nutzer mit seinen Änderungen zufrieden, kann er diese an den Server durch einen weiteren Button senden, oder mit einem anderen wieder löschen. Die Änderungen werden nur vom Dienst durchgeführt und erfolgen nicht vom Nutzerprogramm. So kann es einen Schedulingrythmus andauern, bis die Änderungen umgesetzt wurden.

4.1.2 Verbindung zur Datenbank und Anzeigen der Simulationsdaten

Wie im Entwurf dargestellt werden die Simulationsdaten wie Name, Pfad, Priorität, Nutzer, Laufzeit usw. in einer MySQL-Datenbank abgelegt. Der Zugriff auf die Datenbank wird durch die IP-Adresse, den Port, den Datenbankname, den Zugangsname und das Passwort hergestellt. Diese Daten werden nicht „hartcodiert" im Quellcode des Programms abgelegt. Dadurch entfällt bei einer Änderung der IP-Adresse, das Suchen der Adresse im Quellcode, das Neukompilieren und das Ausliefern der Software an den Benutzer. Um diese Prozedur zu vermeiden, wird mit Konfigurationsdateien gearbeitet, welche der Nutzer schnell im Texteditor bearbeiten kann.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!-- . . . -->
  <applicationSettings>
    <AbaqusManager.Client.Properties.Settings>
      <!-- . . . -->
      <setting name="SqlServerIp" serializeAs="String">
        <value>10.106.64.222</value>
      </setting>
    </AbaqusManager.Client.Properties.Settings>
  </applicationSettings>
</configuration>
```

Listing 1: Auszug aus der XML Konfigurationsdatei des Nutzerprogramms

Bei dem Nutzerprogramm wird dabei eine Anwendungskonfigurationsdatei verwendet. Diese wird von der *Common Language Runtime (CLR)* bei Programmstart geladen. Die Konfigurationsdatei befindet sich im selben Verzeichnis wie die ausführbare *.exe* Datei des

Programms selber. Der Name setzt sich dabei aus dem Namen der *.exe* Datei und dem Suffix *.config* zusammen. [12]

Beim Speichern der Verbindungsdaten handelt es sich um benutzerunabhängige Daten. Alle Nutzer des Programms verwenden dieselben Zugangsdaten und Passwörter zu dem Datenbank- und Dienstserver. Diese Daten werden im Abschnitt `<applicationSettings>` gespeichert. Nutzerspezifische Daten werden den Benutzerverzeichnissen in den lokalen Einstellungen von Windows gespeichert. Diese Daten fallen hier jedoch nicht an.

Der Zugriff auf die Datenbank kann durch *ADO.NET* hergestellt werden. Der Grundgedanke des *ADO.NET* Frameworks ist es, die Daten unabhängig ihrer Datenquellen bereitzustellen. Die Daten sollen auf dieselbe Art und Weise verarbeitet werden können, unbeachtet ob es sich um lokale XML-Dateien oder SQL-Datenbanken handelt. Die erhaltenen Daten können direkt mit einem *DataReader* eines *DataProviders* verarbeitet oder in einem *DataSet* organisiert werden [13]. Das *DataSet* bildet den Datenbankinhalt lokal ab und erlaubt eine schnelle Bearbeitung dessen. Da im zu entwickelnden Dienst nur wenige Daten mit der Datenbank ausgetauscht werden und die Verwaltung der Daten in den Schedulingklassen unabhängig von der Datenhaltung abläuft, wird nur der Datenanbieter verwendet.
Die Verbindung wird durch das *Connection*-Objekt erzeugt. Das *Command*-Objekt realisiert das Senden der Datenbankbefehle und Erhalten der Rückantwort. Ein *Reader* liest dabei die Antwort aus. [14]

Da Microsoft im *.NET Framework* keinen *ADO.NET* Treiber für den Zugriff auf MySQL Datenbanken implementiert hat, muss dieser separat eingebunden werden. Dazu stellt Oracle den *Connector/Net* für MySQL Datenbanken bereit. Dieser wird in Visual Studio eingebunden und implementiert die nötigen *ADO.NET* Schnittstellen.
Um die Verbindung zur Datenbank herzustellen, werden ein *Connection String* und ein Befehl benötigt. Der *Connection String* wird mit den Verbindungsinformationen aus der Konfigurationsdatei beschrieben. Der Befehl (engl. Command*)* wird mittels des *Connection String erzeugt* und bekommt den Befehlstext zugewiesen.
Anschließend wird die Verbindung aufgebaut und der Befehl ausgeführt. Als Rückgabewert erhält man einen *Reader,* welcher nachfolgend ausgelesen werden kann.

```
MySqlConnection connect = new MySqlConnection(myReaderConnectionString);
MySqlCommand command = connect.CreateCommand();
command.CommandText = sqlQuery;
MySqlDataReader Reader;
connect.Open();
Reader = command.ExecuteReader();
while (Reader.Read())
   {
   Job job = new Job();
```

```
  job.Id = int.Parse(Reader.GetValue(0).ToString());
  job.InputFile = Reader.GetValue(1).ToString();
  job.InputFileDir = Reader.GetValue(2).ToString();
  job.Cpus = int.Parse(Reader.GetValue(3).ToString());
   . . .
  connect.Close();
  }
```

Listing 2: Verbindungsaufbau zur Datenbank und Auslesen der Antwort

Der Befehl ist dabei eine *SELECT* Anweisung über mehrere Tabellen, welche mit *JOINS* verknüpft sind. Dabei werden nacheinander alle Einträge der Tabelle *Jobs* durchlaufen. So werden für jeden Lesedurchlauf alle Informationen zu einer Simulation ausgelesen.
Um auch die Informationen des Nutzers der aktuellen Information zu erfassen, wird ein *LEFT JOIN* in die *Users* Tabelle durchgeführt in der die Nutzer-Identifikationsnummer (engl. User ID) dieselbe ist wie die ID des Nutzers der Simulation. Das Selbe wird für den Server auf dem die Simulation läuft durchgeführt. Die Ausgabe wird durch *ORDER BY … ASC* aufsteigend sortiert. So entsteht folgender, gekürzter SQL Befehl:

```
SELECT
  idJobs,InputFile,InputFileDir, . . . ,
  users.idUsers,users.Name,users.SshFileDir,
  servers.idServers,servers.Ip,servers.Name,
FROM
  jobs
  LEFT JOIN users ON jobs.users_idUsers = users.idUsers
  LEFT JOIN servers ON jobs.servers_idServers = servers.idServers ORDER
BY idJobs ASC
```

Listing 3: SQL Befehl zur Ausgabe aller Simulationsinformationen

Diese Informationen werden in einem *Job* Objekt der Klasse *Job* gespeichert. Dadurch werden die Nutzerdaten als Objekt der Klasse *User* und die Serverinformationen als Objekt der Klasse *Server* gespeichert. Dieser objektorientierte Ansatz lässt ein intuitives Programmieren durch das Erstellen von klassenbezogenen Funktionen zu. Für die Anzeige der Daten in der *DataGridView* müssen die Daten aus ihrer verschachtelten Struktur wieder in eine flache Struktur überführt werden. Nach der Initialisierung der Form kommt es zum Laden der Oberfläche. In dem Schritt wird die erzeugte Liste der Simulationsinformationen als Datenquelle für die *DataGridViews* festgelegt. Anschließend müssen die Spalten der Tabellen angeordnet, ihrer Größe angepasst und teilweise ausgeblendet werden.

4.1.3 Nebenläufiges Senden

Das Aufbauen einer Netzwerkverbindung und das anschließende Senden und Empfangen von Informationen kann teilweise länger als einige Millisekunden andauern, so dass es zu einer merklichen Verzögerung der Oberfläche kommt. Da der Nutzer nur mit der Tatsache konfrontiert ist, dass die Oberfläche nicht mehr ansprechbar ist und keinerlei Kenntnis über deren Hintergründe besitzt, wird es zu einer Verringerung des Nutzungserlebnis (engl. User Experience) kommen. Um die Arbeit mit der Software jedoch als so angenehm wie möglich zu gestalten, werden separate Threads zur nebenläufigen Ausführung von zeitintensiven Aktionen verwenden.

Die *BackgroundWorker* Klasse ermöglicht das Ausführen von Programmcode in separaten Threads. Es wird ein *BackgroundWorker* Objekt angelegt, welches die zeitintensiven Operationen ausführt. Dabei vermeldet es in festlegbaren Abständen seinen Fortschritt und eine Meldung, falls es zum Abschluss gekommen ist. Bei dem Nutzerprogramm werden diese Objekte für das Senden der Daten über *TCP* an den Dienst genutzt. Das *BackgroundWorker* Objekt besitzt die Eventlistener DoWork, *ProgressChanged und RunWorkerCompleted*.
DoWork arbeitet den zeitintensiven Prozess ab. *ProgressChanged* aktualisiert die grafische Oberfläche und *RunWorkerCompleted* wird aufgerufen, wenn der zeitaufwendige Prozess beendet ist.

Wenn ein Nutzer den Button für das Absenden der Simulation an den Dienst betätigt, wird ein *Eventhandler* für das Klickereignis auf den Button aufgerufen. In diesem wird zuerst geprüft, ob der *BackgroundWorker* noch aktiv ist. Dieser Fall kann durchaus eintreten, falls der Nutzer den Button ausversehen doppelt anklickt. Ist der *BackgroundWorker* nicht aktiv, werden die ausgewählte Priorität und der Simulationstyp aus der Form in eine Variable geschrieben, welche der *RunWorkerAsync* Funktion mit übergeben wird, da in *DoWork* keine Benutzeroberflächenobjekte mehr bearbeitet werden dürfen [15].

Durch die *RunWorkerAsync*-Funktion wird das Event *DoWork* des *BackgroundWorkers* aufgerufen. In diesem werden nun die zeitaufwendigen Schritte abgearbeitet. Zu Beginn wird das *Job*-Objekt aus der Klasse *Job* erzeugt und mit den eingegebenen Informationen befüllt. Wenn das abgeschlossen ist, wird die Funktion *ReportProgress* des *Backgroundworkers* aufgerufen. Dadurch wird wieder ein Event ausgelöst. Das *ProgressChanged*-Event gibt Rückmeldung an die Oberfläche, wie weit fortgeschritten der Prozess schon ist. In der Oberfläche wird dabei ein Textfeld aktualisiert und eine *ProgressBar* befüllt. Im *DoWork*-Event wird nachfolgend der Nutzername durch die Umgebungsvariable `Environmend.UserName` festgelegt. Weiterhin werden die Priorität und der Typ der Simulation zugewiesen. Im Falle der Nutzer hat in der Oberfläche keine Auswahl darüber getroffen, ist die Priorität 3 und der Typ „implicit“ für eine statische Simulation.
Um zu prüfen, ob der Nutzer schon in der Datenbank existiert, wird eine Datenbankabfrage nach der Nutzer-ID zum zugehörigen Nutzernamen gestartet. Existiert kein Eintrag, muss

der Nutzer neu angelegt werden. Ebenfalls ist es nötig, dass die private Schlüsseldatei an den Server übertragen wird, um Simulationen des Nutzers starten zu können. Die Schlüsseldatei wird aus dem Nutzerverzeichnis geladen. Falls keine Schlüsseldatei im Nutzerverzeichnis vorhanden ist, wird eine Fehlermeldung ausgegeben.
Im letzten Schritt wird ein *TaskSender* erzeugt und die Simulation über diesen gesendet. Die Antwort des Servers wird an das *ProgressChanged* Event übergeben und an die Oberfläche ausgegeben.

4.1.4 Überprüfung der Nutzeränderungen

Als weitere Anforderung an das Nutzerprogramm, soll man eine Simulation löschen und ihre Priorität ändern können.
Um zu verhindern, dass Nutzer die Simulationen anderer Nutzer grundlos löschen können, wurde diese Funktion nur für die nutzereigenen Simulationen erlaubt. Jedoch kann es passieren, dass ein Nutzer im Urlaub ist und vergessen hat, dass eine Simulation noch tagelang aktiv sein wird. Dazu gibt es die Möglichkeit, dass die anderen Benutzer die Simulation in ihrer Priorität verringern können. Um in Zukunft die Möglichkeit offen zu lassen, Nutzergruppen einzuführen, welche das Recht besitzen alle Simulationen zu löschen, wurden entsprechende Grundlagen in dem Datenentwurf geschaffen. So müsste, lediglich die Gruppenzugehörigkeit der Nutzer abgefragt werden.

Weitere zu beachtenden Anwendungsfälle wären der Umgang mit einer Löschanweisung an nicht mehr vorhandene Simulationen oder beendete Simulationen. In einem solchem Fall werden alle Änderungen an beendeten Simulationen abgelehnt, da sie keine Wirkung erzielen.

Die Nutzeränderungen werden in einer Liste verwaltet. Wenn der Nutzer eine neue Änderung hinzufügt, wird die Liste zuerst durchsucht, ob schon eine Änderung zu derselben Simulation existiert. Falls dem so ist, wird die alte Änderung durch die neue ersetzt. So sind stets nur die letzten Änderungen in der Liste enthalten. Bei einer Prioritätsänderung wird zudem geprüft, ob sich die Priorität erhöht oder verringert.
Wenn der Nutzer seine Änderungen prüfen lässt, wird eine zweite Liste angelegt, welche die zulässigen Änderungen speichert. In einer Schleife werden nacheinander die zu prüfenden Änderungen durchgegangen. Falls sich eine Änderung auf eine schon beendete Simulation bezieht, wird eine Fehlermeldung in einem Textfeld dazu ausgegeben. Es ist nur dem Eigentümer der Simulation gestattet, diese zu löschen und die Priorität zu erhöhen. Falls ein anderer Nutzer den Versuch prüft, wird eine Fehlermeldung ausgegeben. Die Priorität zu verringern ist Jedem gestattet. Die erlaubten Änderungen werden in der zweiten Liste gespeichert und an den Dienst übertragen, wenn der Nutzer diese sendet.

4.1.5 Implementierung mit dem Mono-Framework

Mono ist eine plattformunabhängige Laufzeit- und Entwicklungsumgebung. Es ist eine *Open Source* Entwicklung des .NET Frameworks mit dem Ziel Programme, welche in C# geschrieben wurden sind auch auf anderen Betriebssystemen als Windows lauffähig zu machen. Das Mono-Projekt ermöglicht in C# entwickelte Software auf der *Common Language Runtime* von Mono unter Linux zu starten. Anstatt die Software nur für Windows zugänglich zu machen, kann das Programm mit einer Entwicklung in Mono unter Linux, Mac OS und weiteren Betriebssystemen genutzt werden. [16]

Da das Mono-Framework jedoch eine Folgeentwicklung des .NET Frameworks darstellt sind nicht alle Funktionen des .NET Frameworks enthalten. Es muss folglich geprüft werden, welche Funktionen existieren und welche neu Implementiert werden müssen. Für die Prüfung, welche der verwendeten Programmbibliotheken unter Mono unterstützt werden, können diese mit dem *Mono Migration Analyzer* getestet werden. Dieser meldet anschließend, wenn Funktionen existieren, welche nicht in Mono existieren bzw. noch nicht umgesetzt sind.

Der Entwickler muss nun abwägen, ob die Funktionen zu vernachlässigen sind. Dies ist der Fall, wenn es sich um Interfaces oder Events handelt, welche nicht genutzt werden. Falls die Funktion benötigt wird, existieren verschiedene Vorgehensweisen. Einerseits könnte man durch Kompileranweisungen (engl. compiler conditional directives) die Verwendung verschiedener Funktionen in Abhängigkeit der Laufzeitumgebungen definieren (z.B. NET unter Windows, Mono unter Linux). Andererseits könnte auf andere Funktionen ausgewichen werden, bzw. die Funktionen könnten selbst implementiert werden. Bei der Portierung sind drei Problemquellen besonders häufig. Zum einen die Verwendung der Win32 APIs unter .NET, da diese APIs plattformspezifisch sind und nicht unter Linux existieren und zum anderen die Beachtung der Groß/Kleinschreibung und die Verwendung des Schrägstriches in Pfadangaben.

An statt der Win32 APIs können Klassen verwendet werden welche im .NET bzw. Mono Framework enthalten sind. Da das Linuxdateisystem zwischen Groß- und Kleinschreibung unterscheidet, müssen Dateinamen korrekt genutzt werden. Die Pfadtrennung in Windows erfolgt durch den Backslash „\“, unter Linux wird stattdessen der normale Slash „/“ verwendet. In Linux würden Dateien mit Windows Pfadangaben nicht gefunden werden. Um dies zu verhindern wird die `Path.DirectorySeparatorChar`–Variable in den Pfadangaben verwendet. [17]

Unter der Einhaltung dieser Grundsätze wurde das Nutzerprogramm für Linux und Mac OS X realisiert. Um weitere Kompatibilitätsprobleme zu vermeiden, wurde eine grafisch einfachere Oberfläche verwendet. Die Funktionalität bleibt unter allen Betriebssystemen bestehen.

4.2 Dienstprogramm

Das Dienstprogramm stellt den Kern des Schedulings der Simulationsauträge dar. Es nimmt die Simulationsaufträge der Nutzer entgegen, speichert die Simulationsdaten in der Datenbank und startet die Simulationen auf den Simulationsservern.

4.2.1 Dienstarchitektur

Entsprechend den Anforderungen wurde das Programm nicht als Konsolenprogrammen entwickelt, sondern als Windows-Dienst. Mit Diensten bezeichnet man Anwendungen mit langer Laufzeit, welche in eigenen Windowssitzungen ausgeführt werden. Dienste können zudem mit dem Server automatisiert gestartet werden, um den Wartungsaufwand nach einem Neustart einzuschränken.

Um einen Dienst zu starten, muss dieser installiert werden. Dazu wird in Visual Studio dem Dienstprojekt ein Setupprojekt angehängt. Der installierte Dienst kann zudem im Dienststeuerungsmanager gestartet und gestoppt werden. Da keine Anzeige über den Status des Dienstes und auftretende Fehler existiert, wird ein *EventLog* genutzt, um Ereignisse zu protokollieren.

Da ein Dienst nicht mehr in der Form debuggt werden kann, wie eine gewöhnliche Anwendung, wurde in der Realisierung ein Kommandozeilenprogramm mit selben Funktionsumfang entwickelt. Anderenfalls müsste der Dienst bei jedem Test neu installiert werden. Dies bedeutet einen hohen zeitlichen Mehraufwand. Es ist jedoch möglich an den gestarteten Dienst einen Debugger anzuhängen, sobald dieser gestartet wurde. [18]

Ein Dienst besitzt mehrere Zustände während seiner Lebensdauer. Beim Start wird die *OnStart* Methode aufgerufen. In dieser Methode wird die Konfigurationsdatei geladen. Weil die *OnStart* Methode zum Betriebssystem zurückkehren muss, wenn der Dienst ausgeführt wird, wird in dieser ein neuer Thread gestartet [19]. In diesem Thread sind Endlosschleifen erlaubt. So wird erst dann der Timer aktiviert und der *TCP Listener* nachfolgend gestartet.

4.2.2 Datenfluss vom Auftrag zur gestarteten Simulation

Wie in der Erläuterung des Nutzerprogramms beschrieben, werden die Simulationsdetails als *XML*-Datenobjekt strukturiert und über den *TCP*-Datenstream gesendet. In der Konfigurationsdatei des Nutzerprogramms sind IP- und Port-Adresse des Dienstprogrammes gespeichert. Die Port Adresse ist ebenso in der Konfigurationsdatei des Dienstprogramms enthalten. Bei einem Neustart des Dienstes wird diese, sowie viele andere Parameter der Konfigurationsdatei geladen. So können schnell und unkompliziert Änderungen an den Einstellungen vorgenommen werden.

Ebenfalls ist in der Konfigurationsdatei der gewünschte Schedulingalgorithmus angegeben. Nach diesem Parameter wird im Dienstprogramm eine Endlosschleife gestartet, welche

entscheidet, ob nach dem prioritätenbasierenden Scheduling oder dem prioritätenbasierenden Scheduling mit *Round-Robin* gearbeitet werden soll. In der Schleife wird ein *TaskListener*-Objekt erstellt. Mit der Methode *Listen* wird ein *TCP-Listener* an dem festgelegten Port gestartet und auf eingehende Verbindungswünsche gewartet. Es wird dabei die *System.net.Sockets.TcpListener*-Klasse verwendet. Wenn eine Verbindung hergestellt wurde, erhält man einen NetworkStream. Zu diesem Moment wird der Timer ausgeschalten, um zu verhindern, dass die Abarbeitung des Streams unterbrochen wird. Aus dem Stream werden zuerst die Daten gelesen. Anschließend wird über den Stream eine Meldung an das Nutzerprogramm gesendet, dass die Daten erhalten wurden. Nachfolgend wird der Timer wieder aktiviert.

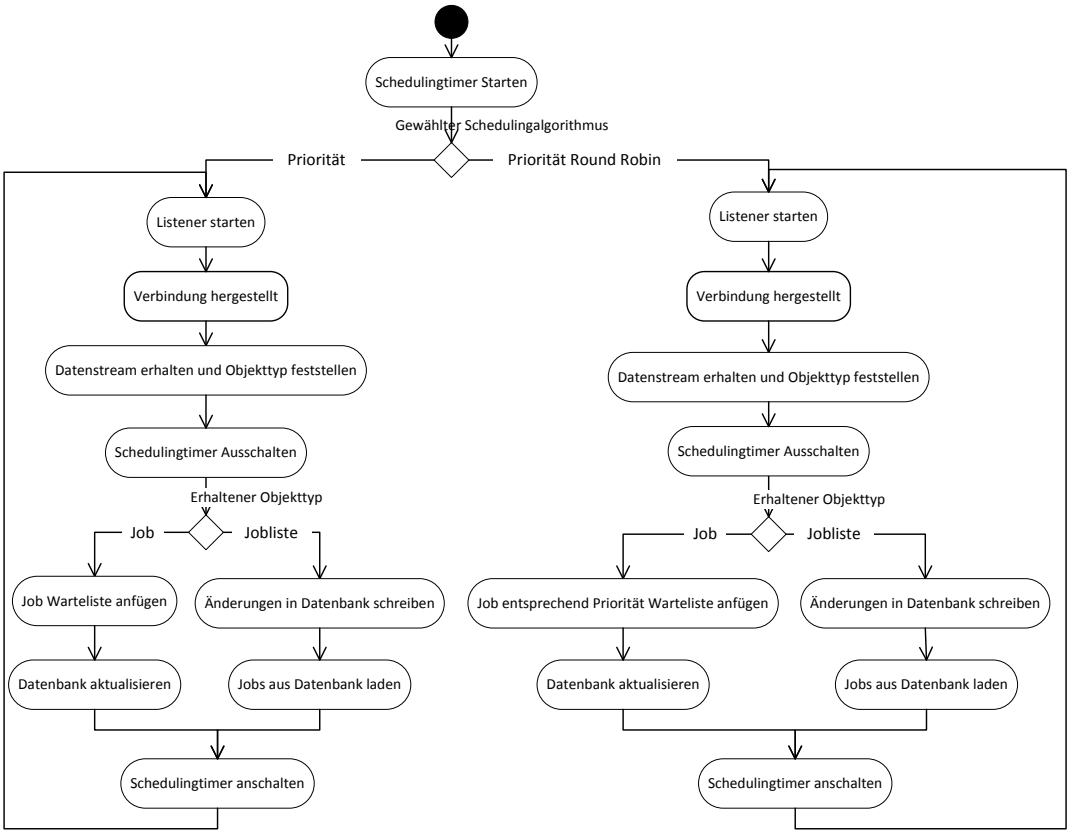


Abbildung 26: Aktivitätsdiagramm Empfang und Verarbeitung der gesendeten Objekte

Nachdem eine Verbindung hergestellt wurde, könnte sofort eine neue Verbindung hergestellt werden, um gleichzeitig mehrere Clienten zu bedienen. Um das zu realisieren müsste die Kommunikation zu dem neuen Client in einem Thread ausgelagert werden. Da es dadurch jedoch ebenfalls zu einer zeitgleichen Änderung der Datenbank kommen kann und damit Inkonsistenz in der Datenhaltung entstehen könnte, wurde darauf verzichtet. Das erwartende Nutzeraufkommen ist so gering, dass es keine Auswirkungen auf den Umgang mit der Software hat. Die Zeitdauer der einzelnen Verbindungen liegt bei wenigen

Millisekunden. Falls das das Nutzeraufkommen in Zukunft stark steigt, ist eine Zwischenspeicherung der Aufträge in einer Warteschlange zu empfehlen.

Für den Fall, dass ein Nutzer eine Simulation startet, wird diese Simulation zuerst der *Jobliste* angehängt. In der *AddJob* Funktion wird geprüft, ob schon eine Simulation mit derselben Eingabedatei im selben Verzeichnis existiert. Existiert eine solche Simulation und ist diese beendet, wird sie der Übersichtlichkeit halber gelöscht. Ist die Simulation noch aktiv, wird geprüft, ob der Nutzer identisch der der neuen Simulation ist. Falls der Nutzer gleich ist, hat er vermutlich einen Simulationsauftrag doppelt abgesendet. Um zu verhindern, dass der bisherige Simulationsfortschritt durch eine erneute Simulation überschrieben wird, wird in dem Fall abgebrochen. Eine Simulation muss so immer zuerst beendet werden bevor sie erneut gestartet werden kann.

Ist die Simulation zulässig, wird diese an die *MySQLClient* Klasse zur Speicherung übergeben. Zu Beginn wird die NutzerID anhand vom Nutzernamen abgefragt, da diese ID einen Fremdschlüssel in der *Job*-Tabelle der Datenbank darstellt. Dazu werden die Nutzer aus der Datenbank ausgelesen. Wird der Nutzer gefunden, werden die ID und der Name seiner Schlüsseldatei im *User*-Objekt des Jobs gespeichert.
Wird der Nutzer nicht gefunden, muss er angelegt werden. Dazu wird zuerst die Schlüsseldatei abgespeichert, welche mit Übertragen wurde. Um eine eindeutige Zuordnung vom Nutzer zur Schlüsseldatei, aber nicht von der Schlüsseldatei zum Nutzer, zu schaffen, wurde eine Digitale Signatur genutzt um den Dateinamen zu erstellen. Der verwendete *Secure Hash Algorithm 1 (SHA1)* erstellt einen individuellen Hashwert für die Schlüsseldatei. Es ist nicht unter vertretbaren Aufwand möglich, denselben Hashwert mit einer anderen Datei zu erstellen. Es ist weiterhin nicht möglich, die Schlüsseldatei aus dem Hashwert zu erzeugen. Fallen die Hashwerte in die Hände eines Angreifers, sind diese für ihn nutzlos. Die Schlüsseldateien werden lokal auf dem Server abgespeichert. Um diese zu erhalten, müsste der Angreifer direkt Zugang zu den Dateisystemen der Server haben. [20]

Nachdem die Schlüsseldatei unter ihrem Hashwert abgespeichert wurde wird der Nutzer mit Name und Hashwert in der Datenbank gespeichert. Anschließend wird die ID des Nutzers abgefragt, damit diese dem Nutzerobjekt zugewiesen werden kann. Da mit der Nutzer ID alle Informationen vorhanden sind, wird die Simulation in der Datenbank gespeichert. Zum Schluss wird die ID der Simulation abgefragt und dieser zugewiesen. Nun befindet sich die Simulation im wartenden Zustand.

Die Aufträge der Nutzer werden mit dem Aufruf des Timers abgearbeitet. Die Aufrufperiode wird dabei durch einen Eintrag in der Konfigurationsdatei bestimmt.

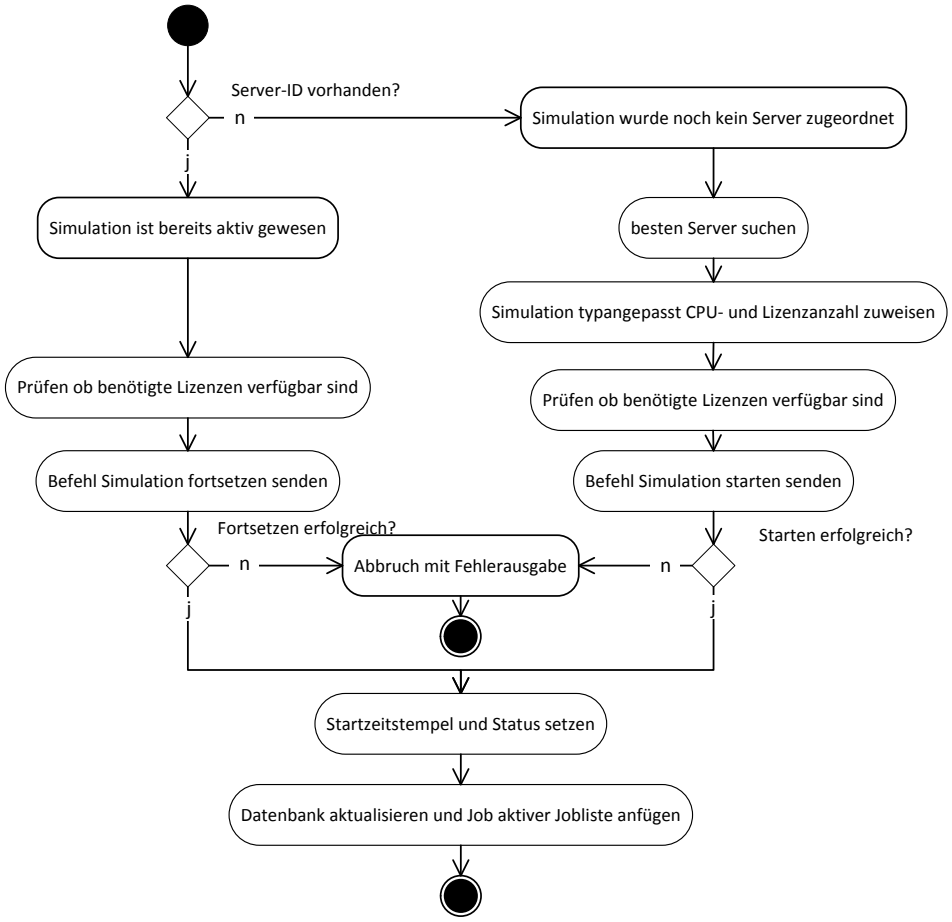


Abbildung 27: Aktivitätsdiagramm Starten einer Simulation

Wenn die Simulation gestartet wird, findet zu Beginn die Prüfung statt, ob die Simulation bereits einem Server zugewiesen wurde. Fall dies nicht der Fall ist, wird der am wenigsten ausgelastete Server genutzt. Dabei wird die aktuelle Serverliste aus der Datenbank geladen und an die einzelnen Server einen `iostat` Befehl gesendet. Mit dem Befehl `iostat 1 5` erhält man eine Input-Output Statistik der CPU fünfmal im Abstand von einer Sekunde. Der `%idle` Wert wird gemittelt und ergibt die Zeit in Prozent in der sich die CPU im Leerlauf befindet. Der Server mit dem größten Leerlauf wird der Simulation zugewiesen. Nun muss die CPU Anzahl zugewiesen werden. Aus Effizienzgründen wird dabei die Anzahl der Maximalen CPUs genutzt, wie diese in der Konfiguration angegeben sind. Bei dynamischen Simulationen wird eine Lizenz mehr beansprucht als bei statischen. Aus diesem Grund wird bei den dynamischen die CPU Anzahl um eins verringert. Die Lizenzanzahl ergibt sich aus der CPU Anzahl. Die Verteilung ist ebenfalls ein Parameter der Konfigurationsdatei.

Bevor die Simulation endgültig gestartet werden kann muss geprüft werden, ob die benötigten Lizenzen auch freigegeben sind. Es kann durchaus dazu kommen, dass der Lizenzserver Lizenzen längst gestoppter Simulationen als noch in Benutzung ansieht. Auch könnte ein Nutzer das Scheduling umgangen haben, indem er direkt Lizenzen beantragt hat. In solch einem Fall wird einfach so lange gewartet, bis die Lizenzen frei gegeben

werden. Man könnte auch von einer priorisierten Abarbeitung manuell gestarteter Simulationen sprechen.

Die *SshClient* Klasse beinhaltet alle Funktionen welche *ShellExecute*-Anweisungen an die Server stellen. Um eine Verbindung aufzubauen werden der Nutzername, die Schlüsseldatei des Nutzers und der Befehl benötigt. Die Nutzerinformationen befinden sich im User Objekt der Simulation. Der Befehl setzt sich aus mehreren Teilen zusammen. Zu Beginn muss in das Verzeichnis der Eingabedatei gewechselt werden. Anschließend wird Abaqus mit den Parametern des Dateinamen und der CPU Anzahl gestartet.

```
Befehl = "cd " + EingabeDateiPfad + "; /softw/bin/abaqus job=" +
EingabeDateiName + " cpus=" + CpuAnzahl;
```

Listing 4: Shell Execute Befehl zum Starten einer Simulation

Falls Abaqus die Datei nicht finden kann wird direkt ein Fehler ausgegeben. Wenn die Simulation startet wird der Status der Simulation auf „active“ gesetzt. Der Startzeitstempel wird auf die aktuelle Zeit gesetzt und die Datenbank wird mit diesen Daten aktualisiert.

4.2.3 Statusprüfung der Simulationen

Die Prüfung der Simulationen erfolgt nur, wenn Simulationen in den entsprechenden Listen existieren. Die aktiven Simulationen werden weiterhin nur geprüft, wenn die Anzahl der aktiven Jobs der Lizenzverwaltung unterschiedlich in der Anzahl der Jobs in der Liste ist. Wenn weniger Jobs aktiv sind als in der Lizenzliste, muss folglich eine Simulation beendet sein. Aus diesem Grund muss der Status der Simulationen überprüft werden. Dazu werden nacheinander alle Jobs der Liste abgefragt.
In der Analyse im Kapitel 2.4.2 wurde bereits auf die verschiedenen Phasen der Simulationsabarbeitung und erzeugten Dateien von Abaqus eingegangen. Im Preprozessing von Abaqus wird die Logdatei angelegt. Erst danach wird die Statusdatei erzeugt.

Zu Beginn der Abfrage werden die letzten Zeilen der Status und Logdatei ausgelesen. Das Auslesen wird über den Unix Befehl `tail` realisiert. Die letzte Zeile erhält man durch die Parameter: `tail -n 1` der Befehl wird über den *SshClient* an den Dateiserver gesendet. Als Rückantwort erhält man die letzte Zeile oder eine Fehlermeldung wenn die Datei nicht existiert. Die Antwort wird in den entsprechenden Variablen gespeichert, welche nachfolgend abgefragt werden.

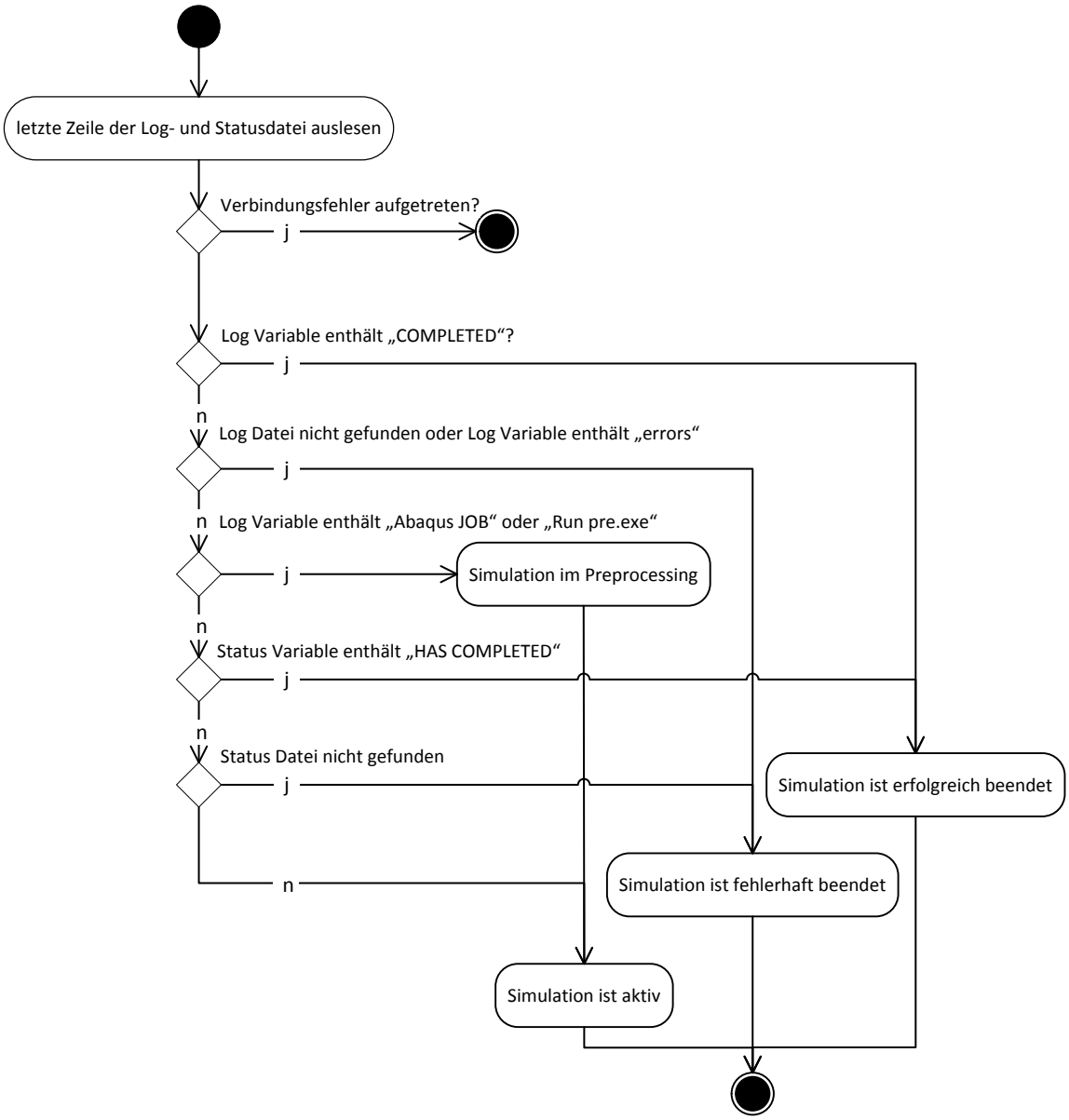


Abbildung 28: Aktivitätsdiagramm Statusüberprüfung einer Simulation

Zu Beginn wird in der Logvariable geprüft ob die Simulation beendet ist, falls dieser der Fall eingetroffen ist, wird eine Rückmeldung darüber gegeben.
Wenn in der Variable nicht das Stichwort „Completed“ existiert, wird geprüft, ob die Logvariable nicht gefunden wurde. Falls dieser Fall eintritt wurde die Simulation eventuell nicht gestartet, oder sie wurde gelöscht. Auf jeden Fall ist sie nicht mehr aktiv.
Falls in der Logvariable die Stichwörter „Abaqus Job“ oder „Run pre.exe“ existieren befindet sich die Simulation in der Vorbereitungsphase und ist aktiv. Existieren diese Begriffe nicht, wird die Statusvariable geprüft.
Existieren in der letzten Zeile der Statusdatei die Begriffe „THE ANALYSIS HAS NOT BEEN COMPLETED“ wurde die Simulation ordnungsgemäß beendet, allerdings konvergierte die Simulation nicht gegen ein bestimmtes Ergebnis. Der Nutzer muss folglich das Ergebnis prüfen und eventuell Anpassungen vornehmen. Falls in der letzten Zeile die Wortgruppe

"THE ANALYSIS HAS COMPLETED" existiert hat die Simulation ihr Ziel erreicht und der Status kann auf beendet gesetzt werden. Wurde die Statusdatei jedoch nicht gefunden kann es sein, dass der Nutzer manuell Dateien gelöscht hat und die Simulation nicht mehr aktiv ist. In diesem Fall muss der Nutzer den Sachverhalt prüfen.

Die Statusüberprüfung mit der Abfrage der Log- und Statusdateien erfolgt in der *Job* Klasse. Die Überprüfung wird von der *JobList* Klasse eingeleitet. An diese wird vom Job Objekt der Status des Jobs zurückgegeben. Der Job kann in 3 Statusmeldungen zurückgeben. Er kann fehlgeschlagen (engl. failed), erfolgreich beendet (engl. finished) und aktiv (engl. running) sein. Für die Zustände „failed" und „finished" wird der Status entsprechend gesetzt und die Zeit seit dem letzten Start auf die Gesamtlaufzeit addiert. Anschließend wird die Datenbank aktualisiert und der Job aus der Jobliste gelöscht.

4.3 Schedulingalgorithmen

Die Schedulingalgorithmen wurden entsprechend dem Softwareentwurf implementiert. Die Auswirkungen der Schedulingparameter werden detailliert in verschiedenen Testreihen analysiert.

4.3.1 Test und Optimierung

Beim Test von Schedulingalgorithmen sind verschiedene Parameter zu beachten. Die Effizienz der Algorithmen wurde im Bezug zu den Metriken Durchsatz und der Durchlaufzeit analysiert. Beim prioritätsbasierenden Scheduling werden nacheinander alle Simulationen ihrer Priorität nach abgearbeitet. Wenn viele kurze und hochpriorisierte Simulationen eine lange niedrigpriorisierte Simulation ausbremsen, ist der Durchsatz zwar hoch, jedoch die gesamte Durchlaufzeit niedrig. Trotz diesem offensichtlichen Nachteils wurde ein strikt prioritätenbasierendes Scheduling gefordert.

Da die einzelnen Simulationen innerhalb ihrer Prioritätsklassen im *Round-Robin*-Prinzip gewechselt werden, lohnt sich die nähere Betrachtung der Wechselzeiten. Für die Gesamtdauer mehrerer Simulationen oder Jobs, sind verschiedene Variablen von Bedeutung. Ein Job bezeichnet einen Simulationsauftrag und wird synonym zum Begriff Simulation verwendet.

Die Jobanzahl und die einzelnen Joblängen sind nicht durch den Administrator änderbar. Es lassen sich nur Vermutungen anstellen, wie viele Simulationen gleichzeitig gestartet werden. Auf Basis der Joblänge könnte man pro Prioritätsklasse Einschränkungen treffen. So sollten Simulationen mit einer erwarteten Simulationsdauer von weniger als 15 Minuten mit Priorität 1 und 2 gestartet werden. Simulationen welche voraussichtlich mehrere Stunden rechnen, würden die Priorität 5 erhalten. Durch solche Festlegungen könnten die Parameter an die Prioritätsklassen angepasst werden.
Die anpassbaren Parameter sind einerseits die Größe eines Quantums und die Anzahl der

Quanten die eine Simulation erhält. Nach jedem Quantum wird geprüft, ob die aktuelle Simulation noch aktiv ist, um gleich die nächste Simulation starten zu können. So muss nicht auf das Vergehen sämtlicher Zeitquanten gewartet werden. Wie groß ein solches Quantum ist, hängt primär von der Einschätzung des Systemadministrators ab. Dieser muss entscheiden, in welchem Maße die Lizenzserverabfragen akzeptierbar sind. Der entscheidende Faktor für das Scheduling ist so die Anzahl der Einzelquanten, bzw. die Gesamtdauer eines Zeitquantums für eine Simulation.
Eine optimale Quantumgröße für eine feste Jobanzahl und Jobwechseldauer kann durch das Errechnen der Gesamtrechendauer bestimmt werden. Dazu wird die reine Rechenzeit zu der Gesamtdauer der Jobwechsel addiert. Wenn man die Gesamtzeit aller verfügbaren Quantengrößen mittelt, erhält man einen Richtwert für die optimale Quantengröße für eine bestimmte Simulationsdauer.

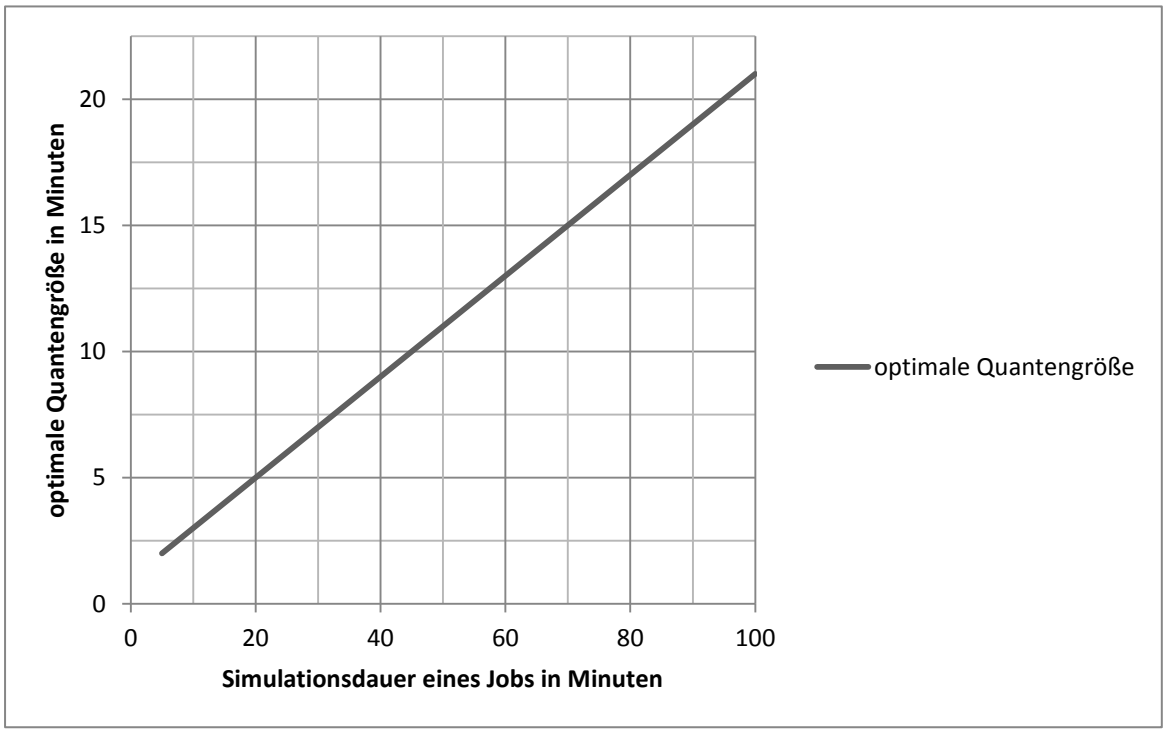


Abbildung 29: Richtwert der Quantumgröße

Die optimale Quantengröße sollte so klein wie möglich sein, um viele Simulationen in kurzer Zeit verarbeiten zu können, jedoch so groß wie nötig, dass die Prozesswechseldauer keinen zu großen Anteil an der Gesamtdauer nimmt.

Um zu Prüfen, wie sich die Größe der Zeitquanten auswirkt, wird für die Simulationen die Rechenzeit (engl. Runtime) gemessen. Die Rechenzeit wird von dem Moment an gemessen, wenn die Simulation gestartet ist, bis zu dem Moment, an dem sie beendet ist. Die Rechenzeit weicht von dem angegeben Zeitquanten ab, da diese lediglich das Timer Event bestimmen.

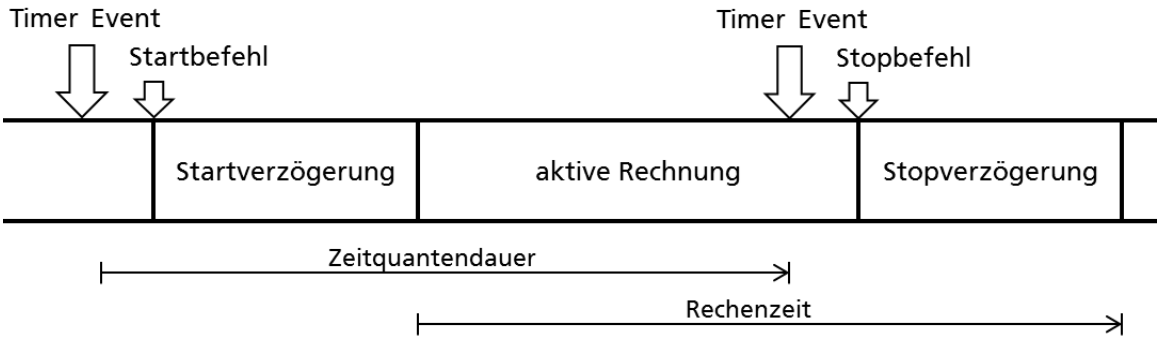


Abbildung 30: Berechnung der realen Rechenzeit

Da die Rechenzeit erst gemessen wird, nachdem die Rückmeldung über den Stop der Simulation erfolgt ist, gleichen sich die Verzögerung für den Aufbau der Verbindung bei Start und Stop aus. So kann davon ausgegangen werden, dass die Rechenzeit annähernd korrekt ist. Für eine mittelgroße Simulation wurden Startzeiten von ca. 5 Sekunden und Stoppzeiten von ca. 7 Sekunden gemessen. Diese Zeiten können, je nach Netz- und Serverauslastung sehr stark variieren.

Die Testläufe bestanden aus einem kompletten Scheduling von sechs gleichen Simulationen. Dabei wurden in drei verschiedenen Prioritätsklassen, jeweils zwei Simulationen gleichzeitig gestartet. Es wurde zu Beginn der Testreihen davon ausgegangen, dass die Zeitspanne zwischen dem Stoppen eines Jobs und dem Starten eines andern vernachlässigbar kurz ausfällt.

Tabelle 3: Übersicht der Testsimulationen bei einer Wechseldauer von 10 Sekunden

Simulation	Priorität	Quantendauer in Minuten	Wechselanzahl pro 100min	Erwartete Gesamt-wechseldauer in Minuten
Sim1	1	2	50	5
Sim2	1	2	50	5
Sim3	3	6	16,7	1,67
Sim4	3	6	16,7	1,67
Sim5	5	10	10	1
Sim6	5	10	10	1

Beim Test zeigte sich jedoch, dass die Abfrage des Lizenzservers durchaus längere Zeit in Spruch nehmen kann. Weiterhin können die Simulationen nur an bestimmten Stellen pausiert werden. Bis zu einem solchen Haltepunkt muss die Simulation aktiv sein. Dieses Verhalten von Abaqus erklärt die vereinzelt auftretende Verzögerung von mehr als 30 Sekunden beim Stoppen einer Simulation.

Ein weiterer Aspekt, welcher Einfluss auf die Auswertung der Testergebnisse nahm, ist die Tatsache, dass die Simulationsgeschwindigkeit abhängig von der Systemauslastung ist. Das heißt, die Testsimulation wurde bei geringer Systemauslastung innerhalb von 21 Minuten berechnet. Wenn der Simulationsserver durch andere Programme stark ausgelastet war, erhöhte sich die Simulationszeit auf über eine Stunde. In Folge wird die Wechseldauer im Verhältnis zur Rechendauer der einzelnen Simulationen dargestellt.

Tabelle 4: Übersicht der Ergebnisse eines Testdurchlaufs

Job Name	Quantendauer in min	Rechenzeit in min	Gesamtzeit in min	Gesamt-wechseldauer in min	Anteil Wechseldauer an Gesamtzeit
Sim1	2	01:13:05	01:23:32	00:10:27	12%
Sim2	2	01:11:34	01:21:03	00:09:29	11%
Sim3	6	00:50:48	00:53:35	00:02:47	5%
Sim4	6	00:48:52	00:52:15	00:03:23	6%
Sim5	10	00:23:15	00:23:51	00:00:36	2%
Sim6	10	00:23:23	00:23:54	00:00:31	2%

Die Wechseldauer liegt im Mittel nicht wie erwartet bei 5 Sekunden, sondern bei ca. 10 Sekunden. Dadurch ergibt sich die höhere Wechseldauer. Wie bereits in der Analyse des *Round-Robin*-Schedulings erläutert, steigt der Anteil der Wechselzeit an der Simulationszeit mit kleineren Zeitquanten sehr stark an.

4.3.2 Ergebnisse der Testreihen

Die Tests haben gezeigt, dass die Simulationszeit stark von der Auslastung des Servers abhängt. An dieser Stelle wird deutlich, wie wichtig es ist, dass der Scheduler für den Start der Simulation den am wenigsten ausgelasteten Server nutzt. Der Nutzer profitiert von dieser Serverauswahl, denn seine Simulation wird schneller gerechnet werden. Die Standardabweichung der Gesamtdauer einzelner Simulationen beträgt bis zu 20 Minuten. Bei niedriger Serverauslastung werden die Simulationen, unabhängig ihrer Quantengröße, in unter 30 Minuten gerechnet. Mit zunehmender Auslastung des Servers steigt die Simulationsdauer auf über 50 Minuten.

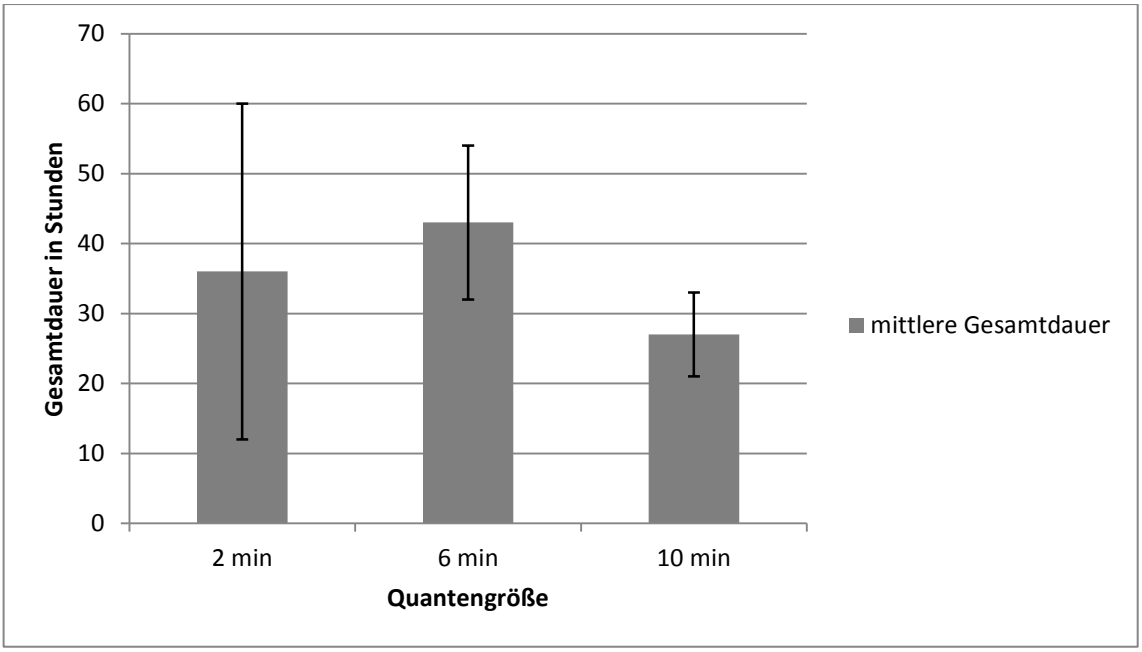


Abbildung 31: Mittlere Gesamtdauer pro Simulation bei verschiedenen Quantengrößen

Die Tests ergaben genaue Zeiten der Prozesswechsel. Es ließ sich beweisen, dass Simulationen mit kurzen Zeitquanten insgesamt signifikant mehr Zeit dafür benötigen, gestartet und pausiert zu werden, als Simulationen mit längeren Quanten. Dass bei einem Zeitquantum von 10 Minuten nur noch ca. 2% der gesamten Rechenzeit für die Wechsel benötigt werden, ist besonders positiv, da bei 10 Minuten ein zufriedenstellender Durchsatz von 10 Simulationen die Stunde erreicht werden kann.

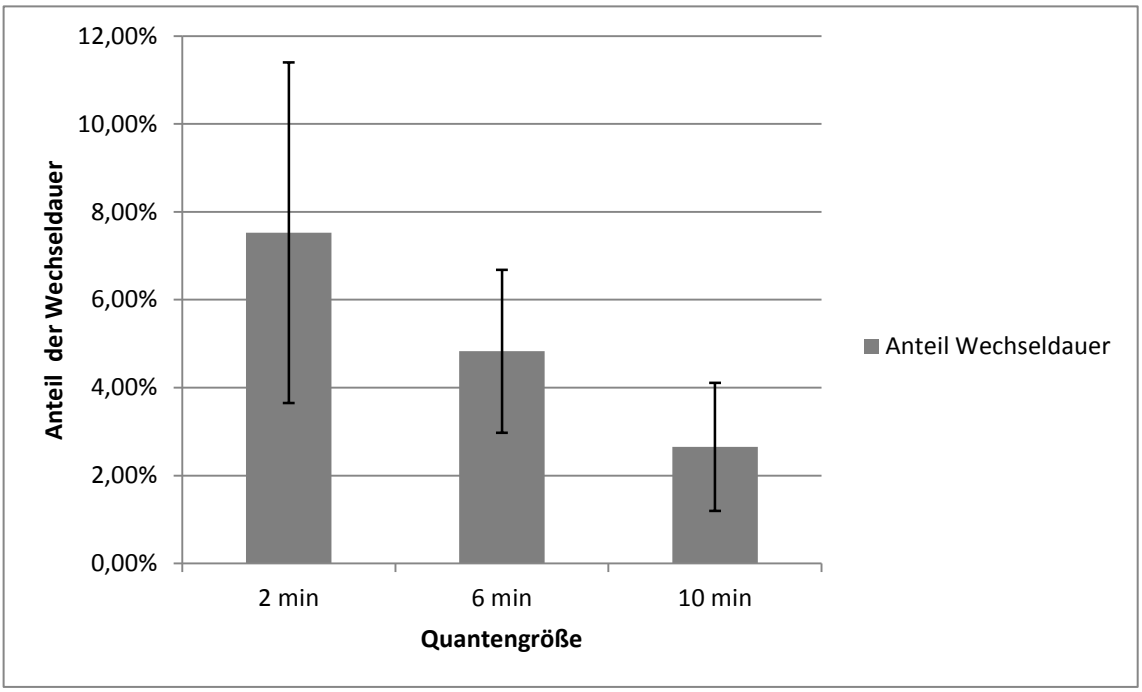


Abbildung 32: Anteil der Wechseldauer an der Gesamtsimulationsdauer pro Simulation

5 Fazit

5.1 Ergebnisse

Während der Analyse der Simulationssoftware Abaqus und der Optimierung der Lizenznutzung, um Engpässe zu beseitigen, wurden verschiedene Meilensteine erreicht.
Das komplexe Softwaresystem, bestehend aus Datei-, Simulations- und Lizenzservern wurde tiefgehend analysiert. Durch prototypisches Vorgehen wurden Schnittstellen zu diesen Komponenten verifiziert. Mit der Realisierung der Prototypen konnte frühzeitig festgestellt werden, welche Softwarekomponenten welchen Entwicklungsaufwand besitzen und so konnte auf diese Erkenntnisse mit der Verteilung der Arbeitszeit reagiert werden. Innerhalb des gesetzten Zeitrahmens wurde eine funktionierende und erweiterbare Softwarelösung zur Verwaltung der Abaqus-Simulationen geschaffen.
Durch die theoretische Betrachtung verschiedener Schedulingalgorithmen konnten zwei Algorithmen ausgewählt werden, welche den Anforderungen und Rahmenbedingungen des Softwaresystems gerecht werden. Die theoretisch erbrachten Erkenntnisse über die Gesamtdauer der Simulationen bei verschiedenen Quantengrößen, konnten in der Realität durch Tests des entwickelten Softwaresystems bestätigt werden.

Am Fraunhofer Institut für Werkstoffmechanik in Halle wurde ein Softwaresystem aufgebaut, welches den Umgang mit dem Simulationsprogramm Abaqus durch die grafische Oberfläche und Statusanzeige erleichtert und die Lizenzverteilung optimiert.

5.2 Ausblick

Die realisierte Client-Dienst Architektur wurde nach objektorientierten Merkmalen auf beste Wartbarkeit und Erweiterbarkeit entwickelt. Künftige Anpassungen auf Grund neuer Abaqus-Versionen und Startparameter können zielgerichtet im Nutzerprogramm, in der *SshClient*-Komponente und der Datenbank vorgenommen werden. So reicht es aus, die Spezifikationen der Schnittstellen zu kennen, um Modifizierungen zu implementieren.

Falls weitere Nutzer im Betrieb auf die Nutzung von Abaqus angewiesen sind und es vermehrt zu Fehlentscheidungen beim Setzen der Prioritäten kommt, kann auf die Nutzung der Benutzergruppen zurückgegriffen werden. Dabei können einzelnen Benutzern übergreifende Rechte zum Löschen von Simulationen anderer gewährt werden. Ebenfalls könnte die Absprache getroffen werden, dass neue Mitarbeiter nur bestimmte Prioritäten vergeben dürfen.
Als Weiterentwicklung könnte die Anbindung an das Simulationssystem „Ansys“ realisiert werden.

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] A. S. Tanenbaum, „Moderne Betriebsysteme," Pearson Studium, München, 2003.

[2] U.-P. D. M.-R. Wolff, 23 02 2005. [Online]. Available: http://winfor.uni-wuppertal.de/fileadmin/wolff/Downloads/Grundstudium/EWI/Phasenmodelle.pdf. [Zugriff am 31 07 2013].

[3] S. Kleuker, Grundkurs Software-Engineering mit UML, Wiesbaden: Vieweg+Teubner Verlag, 2011.

[4] J. Seemann und J. Wolff von Gudenberg, Software Entwurf mit UML 2, Heidelberg: Springer Verlag, 2006.

[5] P. Steinke, Finite-Element-Methode, Berlin: Springer-Verlag, 2010.

[6] Dassault Systèmes, „Abaqus 6.12 Getting Started with Abaqus," Dassault Systèmes Simulia Corp., Providence, RI, USA., 2012.

[7] T. Ylonen und C. Lonvick, „RFC4252: The Secure Shell (SSH) Authentication Protocol," The Internet Society, 2006.

[8] G. Bengel, C. Baun, M. Kunze und K.-U. Stucky, Masterkurs Parallele und Verteilte Systeme, Wiesbaden: Vieweg+Teubner, 2008.

[9] J. Swoboda, S. Spitz und M. Pramateftakis, Kryptographie und IT-Sicherheit, Wiesbaden: Vieweg+Teubner, 2008.

[10] T. Ylonen und C. Lonvick, „RFC4253: The Secure Shell (SSH) Transport Layer Protocol," The Internet Society, 2006.

[11] Microsoft, „Übersicht über Windows Forms," Microsoft, [Online]. Available: http://msdn.microsoft.com/de-de/library/8bxxy49h.aspx. [Zugriff am 02 08 2013].

[12] A. Kühnel, Visual C# 2010, Bonn: Galileo Press, 2010.

[13] Microsoft, „Übersicht über ADO.NET," Microsoft, [Online]. Available: http://msdn.microsoft.com/de-de/library/vstudio/h43ks021.aspx. [Zugriff am 01 08 2013].

[14] Microsoft, „ADO.NET-Architektur," Microsoft, [Online]. Available: http://msdn.microsoft.com/de-de/library/vstudio/27y4ybxw.aspx. [Zugriff am 20 08 2013].

[15] Microsoft, „BackgroundWorker-Klasse," Microsoft, [Online]. Available: http://msdn.microsoft.com/de-de/library/system.componentmodel.backgroundworker%28v=vs.110%29.aspx. [Zugriff am 01 08 2013].

[16] „What is Mono," Xamarin, [Online]. Available: http://mono-project.com/What_is_Mono. [Zugriff am 01 08 2013].

[17] „Guidelines:Application Portability," Xamarin, [Online]. Available: http://www.mono-project.com/Guidelines:Application_Portability. [Zugriff am 01 08 2013].

[18] Microsoft, „Einführung in Windows-Dienstanwendungen," Microsoft, [Online]. Available: http://msdn.microsoft.com/de-de/library/d56de412%28v=vs.80%29.aspx. [Zugriff am 01 08 2013].

[19] Microsoft, „Exemplarische Vorgehensweise: Erstellen einer Windows-Dienstanwendung im Komponenten-Designer," Microsoft, [Online]. Available: http://msdn.microsoft.com/de-de/library/zt39148a%28v=vs.80%29.aspx. [Zugriff am 01 08 2013].

[20] Eastlage und Jones, „RFC 3174: US Secure Hash Algorithm 1 (SHA1)," The Internet Society, 2001.

Eidesstattliche Erklärung

Name:	Richter	Vorname:	Lars
Matrikel-Nr.:	17875	Studiengang:	Bachelor Angewandte Informatik

Hiermit versichere ich, Lars Richter, an Eides statt, dass ich die vorliegende Bachelorarbeit mit dem Titel „Entwicklung einer Anwendung zur Steuerung der Stapelverarbeitungsfunktionen in Abaqus“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Auszug aus dem Strafgesetzbuch (StGB)

§ 156 StGB Falsche Versicherung an Eides Statt

Wer von einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Ort, Datum

Unterschrift