Mirko Caspar

Lastgetriebene Validierung Dienstbereitstellender Systeme

Wissenschaftliche Schriftenreihe

EINGEBETTETE, SELBSTORGANISIERENDE SYSTEME

Band 11

Prof. Dr. Wolfram Hardt (Hrsg.)

Mirko Caspar

Lastgetriebene Validierung Dienstbereitstellender Systeme

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Universitätsverlag Chemnitz
2013

Impressum

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über http://dnb.d-nb.de abrufbar.

Zugl.: Chemnitz, Techn. Univ., Diss., 2013

Technische Universität Chemnitz/Universitätsbibliothek
Universitätsverlag Chemnitz
09107 Chemnitz
http://www.bibliothek.tu-chemnitz.de/UniVerlag/

Herstellung und Auslieferung
Verlagshaus Monsenstein und Vannerdat OHG
Am Hawerkamp 31
48155 Münster
http://www.mv-verlag.de

ISSN 2196-3932

ISBN 978-3-941003-84-2

URL: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-110257

VORWORT ZUR WISSENSCHAFTLICHEN SCHRIFTENREIHE ”EINGEBETTETE, SELBSTORGANISIERENDE SYSTEME”

Der elfte Band der wissenschaftlichen Schriftenreihe Eingebettete, Selbstorganisierende Systeme behandelt die Validierung komplexer, verteilter Systeme. Betrachtet man beispielsweise ein Mobilfunknetz, so kann eine Validierung nicht im Labor stattfinden. Eine Vielzahl von Diensten ist sicher und performant zu erbringen.

Herr Mirko Caspar greift dieses Problem auf und führt ein automatisiertes Testkonzept ein, das durch die Generierung vorgegebener Lasten Testfälle erzeugt. Mit der Konzentration auf die zu testenden Dienste gelingt es Herrn Caspar, eine Abstraktion einzuführen, die Automatisierung und Flexibilität miteinander verbindet.

Lasten für das zu testende System werden als Testfälle mit Hilfe heterogener Klienten systematisch erzeugt. Die technischen Eigenschaften der Klienten werden ebenso berücksichtigt wie die Größe der Testlast. Das entwickelte Verfahren wird als automatisierte Testbench implementiert und angewandt. So können realistische Bewertungen ermittelt werden. Diese werden dann im Rahmen von Simulationen weiter verwendet.

Das Ergebnis dieser Arbeit ist ein Konzept zur Systemvalidierung, das generisch auf nahezu jede Art Dienstbereitstellender Systeme angewandt werden kann und damit zur Verbesserung des Entwicklungsprozesses von komplexen verteilten Systemen beiträgt.

Ich freue mich, Herrn Caspar für die Veröffentlichung der Ergebnisse seiner Arbeit in dieser wissenschaftlichen Schriftenreihe gewonnen zu haben, und wünsche allen Lesern durch die Lektüre aufschlussreiche Erkenntnisse in diesem Themengebiet.

Prof. Dr. Wolfram Hardt
Professur Technische Informatik
Direktor Universitätsrechenzentrum
Technische Universität Chemnitz
April 2013

TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Lastgetriebene Validierung Dienstbereitstellender Systeme

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur
(Dr.-Ing.)

vorgelegt

der Fakultät für Informatik
Technische Universität Chemnitz

von:	Dipl.-Inf. Mirko Caspar
geboren am:	19.10.1980
geboren in:	Karl-Marx-Stadt, jetzt Chemnitz
Gutachter:	Prof. Dr. Wolfram Hardt Prof. Dr. Ulrich Heinkel

Chemnitz, den 17. Dezember 2012

Danksagung

Da es sich bei dieser Promotion um ein Vorhaben handelte, dessen Bearbeitung sich über einige - nach Meinung einiger: zu viele - Jahre hinzog, ist es nahezu unmöglich, all denen namentlich zu danken, die fachlich oder ideell bei der Bewältigung der Forschungsarbeiten und der Erstellung dieser Dissertation beigetragen haben. Ich möchte die Chance an dieser Stelle nutzen, den wichtigsten Begleitern namentlich zu danken.

An erster Stelle stehen natürlich die, die zur Idee und fachlichen Ausgestaltung dieses Promotionsvorhaben beigetragen haben. Ich danke Prof. Dr. Wolfram Hardt, der als Doktorvater zahllose Impulse für diese Arbeit gegeben und beständig Wert darauf gelegt hat, daß die finalen Ziele nicht zugunsten von Lösungsdetails aus dem Fokus geraten. Als Vorgesetzter ermöglichte er mir die Arbeit in interessanten Projekten und förderte auch nicht alltägliche Lehrveranstaltungen, die mir nicht nur beruflich neue Erfahrungen ermöglichten.

Ebenso gilt mein Dank Prof. Dr. Ulrich Heinkel sowie Dr. Vasco Jerinic von der Core Mountains GmbH. Erst durch ihr Engagement kam ein Entwicklungsprojekt zustande, welches als Ideengeber für die vorliegende Arbeit diente. Einige technologische Grundlagen konnten im Rahmen dieses Projektes erarbeitet werden.

Mit der Niederschrift dieser nun (endlich) vorliegenden Dissertation stieg auch der Aufwand für Kollegen und Freunde, für deren Unterstützung ich mich bedanken möchte. Matthias Vodel, Rico Günther und Jan Zückmantel seien hier als die fleißigen Helfer genannt, die mit großem Engagement die Arbeit inhaltlich und orthographisch dringend notwendigen Revisionen unterzogen.

Selbstverständlich gab es auch zahlreiche Menschen, die nicht direkt an Wort und Bild dieser Arbeit beteiligt waren, aber mir durch ihr Wirken das Leben „neben“ der Dissertation einfacher gemacht haben. Ich danke insbesondere meiner Familie aber auch meinen engen Freunden für die Hilfe, die gelegentlich notwendige Ablenkung und auch das nervige Nachhaken „Hast’e denn nun mal abgegeben?“.

In der Schlußphase der Arbeit kam es häufiger vor, daß ich einigen meiner beruflichen, ehrenamtlichen oder privaten Verpflichtungen nicht in dem Maße nachkommen konnte, wie das von mir erwartet wurde. Allen Kollegen und Freunden, die dafür Verständnis hatten und mich unterstützt haben, sei hiermit herzlichst gedankt.

Jeder der meint, direkt oder indirekt zum Gelingen dieser Arbeit beigetragen zu haben, aber hier nicht genannt wurde, sollte nicht beleidigt sein... sondern sich auf die anstehende Feier zum (hoffentlich erfolgreichen) Abschluß der Promotion freuen.

. . . 42! . . .

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Die von Menschen entwickelten Systeme werden immer komplexer. Wachsende Erfahrungen der Entwickler, Verbesserung der Entwurfsmethodiken - bspw. durch leistungsfähige Entwicklungswerkzeuge - sowie massive Verbesserungen in den Fertigungstechnologien und -prozessen führen dazu, daß immer aufwändigere Systeme mit einem größeren und besseren Funktionsumfang entstehen. Dies trifft auf nahezu alle Ingenieursdomänen zu.

Parallel dazu kommt es auch zur Verschmelzung verschiedener Ingenieursdisziplinen. Vor allem die programmgesteuerte Rechentechnik hielt in den letzten Jahrzehnten Einzug in nahezu allen anderen Ingenieursdisziplinen. Derartige softwarebasierte Systeme bieten die Möglichkeit, einen großen Funktionsumfang mit verhältnismäßig wenigen, standardisierten Komponenten (Hardware) bereitzustellen. Änderungen sind ohne physische Eingriffe und folglich ohne große Aufwände möglich.

Die Spezifikation, Entwicklung und vor allem auch die Verifikation von zustandsdiskreten und besonders von softwarebasierten Systemen sind jedoch nicht trivial und stehen auch nach vielen Jahrzehnten der Disziplin „Software Engineering“ (Vgl. [64], S. 8) noch immer vor zahlreichen ungelösten Problemen. Diese Probleme werden nun in Systeme anderer Ingenieursdisziplinen und immer weitere Anwendungsfelder getragen. Hinzu kommt, daß die Koordination zwischen den Entwicklern der verschiedenen Teilgebiete bei komplexer werdenen Systemen zwangsläufig auch schwieriger wird.

Welch drastische Auswirkungen Fehler in Teilen der Entwicklungsphasen haben können, zeigt das wohlbekannte und vielzitierte Beispiel des misslungenen Jungfernflugs 501 der Trägerrakete Ariane 5 im Jahre 1996. Rund 40 Sekunden nach dem erfolgreichen Start änderte diese abrupt den Kurs und zerbrach in Folge der hohen mechanischen Kräfte, die auf sie einwirkten.

Als Ursache wurde ermittelt (Vgl. [56]), daß ein unverändert aus der Vorgängerversion Ariane 4 übernommenes Steuergerät beim Abbilden einer 64-Bit Fließkommazahl, die die horizontale Geschwindigkeit repräsentierte, auf eine 16-Bit vorzeichenbehafteten Ganzzahl wegen Wertebereichsüberschreitung eine Ausnahme generierte. Diese Ausnahme wurde von den ursprünglichen Entwicklern nie für möglich gehalten, da die Vorgängerversion diese Geschwindigkeiten nicht ansatzweise erreichen konnte. Entsprechend wurde darauf nicht reagiert. Das Steuergerät wechselte in einen Fehlermodus, in dessen Folge Diagnoseinformationen an den Rechner, der für die Steuerung der Rakete zuständig war, gesendet wurden. Dieser interpretierte die Diagnosedaten wiederum als Flugstatusinformationen und folgerte auf starke Abweichungen der Fluglage, die er zu korrigieren versuchte. Die resultierende Zerstörung der Rakete verursachte einen Schaden von rund 360 Millionen US-Dollar und führte zu einer einjährigen Verzögerung des Ariane-Programms.

Besondere Tragik liegt in der Tatsache, daß die Funktion des Steuergerätes, die zur Ausnahme und in Folge zur Zerstörung führte, keine für die Ariane 5 relevanten Informationen berechnete - es handelte sich um ein „Überbleibsel“ von der Nutzung in der Ariane 4.

Es liegt in der Interpretation des Betrachters, welcher Schritt in der Entwicklung der Rakete zum Fehler führte. Die Softwareentwickler haben aus ihrer Sicht ein korrektes Programm - zumindest an der entsprechenden Stelle - erstellt, das für alle gängigen Zustände der Ariane 4 funktioniert und sich auch bei zahlreichen Starts bewährt hatte. Allerdings war es nicht übermäßig „visionär“ dimensioniert, was wegen Ressourcenmangel in der Rechentechnik - zumindest in früheren Zeiten - schon fast symptomatisch war (man bedenke die Jahr-2000 Problematik). Die Systemingenieure wiederum folgten dem populären und an sich durchaus praktikablen Ansatz „never touch a running system“ und übernahmen das bewährte System in die Ariane 5 (Vgl. [10], S. 64). Die anderen Ingenieursdisziplinen konstruierten eine leistungsfähigere - und damit schnellere - Rakete.

In dem konkreten Fall der Ariane 5 hätten vielleicht verbesserte Dokumentationsstände und organisatorische Abläufe den Fehlschlag des Jungfernfluges verhindert. Im Allgemeinen stellt sich jedoch die Frage, wie die Funktion komplexer und heterogener Systeme über die Grenzen verschiedener Ingenieursdisziplinen hinaus überprüft und sichergestellt werden kann. Hierfür sind zahlreiche Ansätze für verschiedene Phasen des Entwurfsprozesses denkbar. Im Mittelpunkt dieser Arbeit steht die Verifikation bzw. Validierung durch Testen.

Komplexe Systeme erschweren klassische Methoden des Tests - insbesondere wenn es sich um heterogene und verteilte Systeme handelt. Der Systemtest, der den kompletten Entwurf bestehend aus allen Komponenten und Relationen zwischen diesen in die Überprüfung einbezieht, gewinnt deutlich an Bedeutung. In Bezug auf die Entwicklung von Flugzeugen schrieben Wise et.al. ([97], S. VII):

> The instability makes the verification/validation process even more important than it has been in the past, while the coupling makes traditional modular testing obsolete. As complex systems become more coupled, interdisciplinary issues also become more critical.

Mit dem Bedeutungsgewinn des Systemtests ändern sich zwangsläufig auch die Testmethoden. Schon die triviale Tatsache, daß in komplexen Systemen häufig keine physische oder logische Entität existiert, an die klassische Testvektoren angelegt werden bzw. an der die Ausgaben zum Vergleich aufgezeichnet werden können, führt dazu, daß die Testfälle stark abstrahiert werden müssen.

Handelt es sich um ein hybrides System, das aus Komponenten verschiedener Ingenieursdisziplinen besteht, ist es außerdem ratsam, den Systemtest derart zu definieren und durchzuführen, daß Vertreter aller Fachbereiche diesen verstehen und bewerten können. Entsprechend abstrakt und vor allem technologieunabhängig müssen die Modelle und Verfahren sein.

Auch wenn die softwarelastige Steuerung häufig massiven Einfluß auf die Funktion des Gesamtsystems hat (siehe Ariane 5), sind reine Softwaretestansätze denkbar ungeeignet, den kompletten Systemtest zu bewerkstelligen. Diese decken per se nur das Zustandsmodell der

Software selbst ab. Außerdem werden die technologiespezifischen Konzepte von anderen Fachdisziplinen nicht verstanden bzw. können in ihrer Wirksamkeit nicht eingeschätzt werden.

Doch nicht nur die Definition dessen, was getestet werden soll, ist relevant. Auch die Ausführung von Tests in komplexen Systemen stellt Entwickler vor Herausforderungen. Häufig müssen bspw. viele parallele Testfälle gleichzeitig ausgeführt werden, um das System realistischen Belastungen auszusetzen. Handelt es sich um ein auch räumlich verteiltes System, stellt sich die Frage, wie diese Testfälle koordiniert werden können. Eine automatisierte Ausführung oder sogar ein automatischer Test auf Basis eines Modells sind wünschenswert bzw. sogar zwingend notwendig.

Bei der Entwicklung komplexer Systeme kommen in der Praxis verschiedene, problemspezifische Ansätze für einen zumindest teilautomatisierten Systemtest zum Einsatz. Zur Verallgemeinerung und Weiterentwicklung der Ansätze kann also auf praktische Erfahrungen zurückgegriffen werden. So basiert auch die hier vorgestellte Arbeit auf einem gemeinsamen Entwicklungsprojekt der Coremountains GmbH Chemnitz sowie der Professur Technische Informatik der Technischen Universität Chemnitz, dessen Ziel die Erstellung einer Testbench zum zentralisierten System- und Feldtest eines Mobilfunknetzwerkes war. Das komplette Netz stellt dabei das zu testende System dar. Aufgrund der aufwändigen und heterogenen Architektur, bestehend aus zahlreichen Servern, Netzwerken und dedizierten Komponenten zur drahtlosen Kommunikation, kann es zweifelsohne als komplexes System betrachtet werden.

Es war bereits gegebene Praxis, das Netz direkt durch Nutzung von Mobilfunkgeräten, also Telefonen und Datenmodems, zu testen. Diese wurden an Rechner angeschlossen und mittels Netzwerk von einem zentralen Testsystem aus ferngesteuert. Dadurch gelang es, das Netz zu belasten und zu überprüfen, inwieweit es sich für den Nutzer stabil verhält. Es ist also ersichtlich, daß Teile des Systemtests in der Praxis als Leistungs- bzw. Lasttests durchgeführt werden.

Das Problem besteht jedoch zum einen in der geringen Dynamik dieses Testszenarios. Die Mobilfunkgeräte waren wegen ihrer Verbindung zu Arbeitsplatzrechnern ortsfest und erlauben damit keinen realitätsnahen Test. Zum zweiten sind die Kosten für die Installation und Wartung dieser Systeme hoch, da viele Rechner angeschafft und gepflegt werden müssen.

Die Testbench des Entwicklungsprojektes soll genau diese Mängel beheben. Zur Fernsteuerung der Mobilgeräte soll kein dediziertes Netz mehr dienen, sondern das Datennetz des Mobilfunknetzwerkes selbst. Damit können die Geräte mobil genutzt werden und es entfällt die zusätzliche Rechentechnik.

Jedoch ergeben sich durch diese Variante des Tests auch Einschränkungen, die elementare und weitreichende Anforderungen an die Testbench stellen. In erster Linie muß die Heterogenität der Testgeräte berücksichtigt werden. Mobiltelefone basieren auf verschiedene Plattformen und verfügen über differierende Leistungsparameter. Dies hat Auswirkungen auf die Durchführung des Tests. Desweiteren kann nicht garantiert werden, daß jedes Mobilgerät während des kompletten Tests verfügbar ist - Funklöcher oder mangelnde Akkukapazitäten können hier zu Schwankungen führen. Im Mittelpunkt steht deshalb die Entwicklung der zentralen Komponenten zur Verwaltung der Klienten und der laufenden Tests.

Eine Automatisierung des Tests war jedoch nicht Bestandteil dieser Entwicklung. Im Kapitel 6, in dem die Implementierung dieses Beispielsystems erläutert wird, werden die Ergebnisse dieser Basisarbeit als Ausbaustufe 1 bezeichnet.

1.1. Zielsetzung dieser Arbeit

Ziel dieser Arbeit ist es, einen Prozeß sowie eine Testbench bereitzustellen, mit deren Hilfe die abstrakten Dienste eines beliebigen Systems validiert werden können. Zur Validierung werden physische oder logische Klienten verwendet, die die einzelnen Dienste durch Anfragen nutzen können. Die Anfragen, die ein Klient am zu testenden System ausführt, sind quantifizierbar und werden als *Last* bezeichnet. Auf dieser Architektur wird eine Validierung eines Dienstes durch die Vorgabe einer zeitabhängigen Soll-Last definiert. Es ist Ziel des Validierungsvorganges, eben jene Soll-Last als Summe der Lasten einzelner Klienten zu generieren.

Dieser lastbasierte Ansatz wurde bisher teilweise in Form manueller Feldtests durchgeführt. Ein wesentlicher Bestandteil dieser Arbeit ist jedoch die Automatisierung des Vorganges, so daß weitaus komplexere Systeme unter Nutzung großer Zahlen von Klienten validiert werden können. Hierfür wurde eine Testbench erarbeitet, deren Grundstruktur in Abbildung 1.1 zu sehen ist. Eingabe für eine Validierung bildet ein Szenario, das die Soll-Last in Abhängigkeit von der Zeit beschreibt. Zu dedizierten Zeitpunkten wird diese Last von einem Algorithmus auf die verfügbaren Klienten verteilt. Diese Aufträge werden an den Klienten übermittelt und dort in Form von Anfragen der entsprechenden Last an den Dienst des zu testenden Systems ausgeführt. Die Ergebnisse und Güteparameter der Ausführung werden an die Testbench übermittelt und dort ausgewertet.

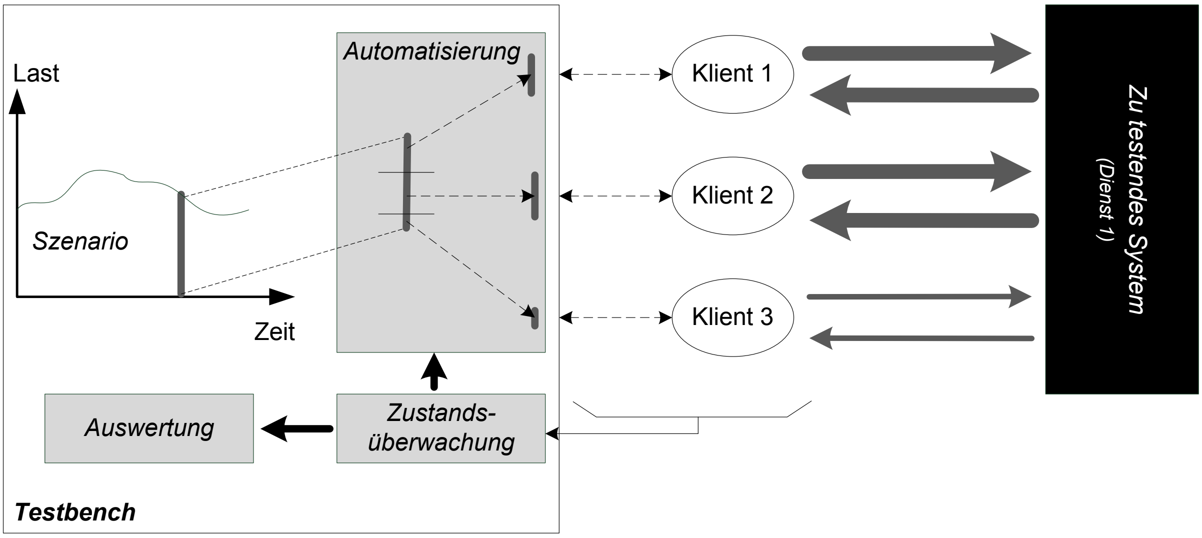


Abbildung 1.1.: Konzept zur automatisierten, lastgetriebenen Validierung

Aus möglichen Anwendungsfeldern und der Strukur der Testbench ergeben sich folgende Kernanforderungen, die berücksichtigt werden müssen:

- Heterogene Klientenarchitektur (Plattformen, Dienste, Ressourcen)
- Instabile Klientenverfügbarkeit (Mobilität, eingeschränkte Kommunikationsfähigkeit)
- Große Anzahl zu verwaltender Klienten und Dienste
- Beherrschbare Verzögerungszeiten bei der Ausführung

Insbesondere die Heterogenität der Klienten und deren schwankende Verfügbarkeit bedingen ein aufwändiges Modell und führen zu einem komplexen Automatisierungsalgorithmus, der das Problem der *Testpartitionierung* löst.

Im Rahmen dieser Arbeit werden alle benötigten Modelle zur Umsetzung der automatisierten, lastgetriebenen Validierung vorgestellt. Das Testpartionierungsproblem wird beschrieben und eine Heuristik erarbeitet, die dieses Problem in polynomieller Laufzeit lösen kann. Die Heuristik wird in ihrer Leistungsfähigkeit validiert.

Dieses Konzept ist anwendbar auf sämtliche Systeme, deren Funktionen sich in Dienste kapseln lassen. Zur Veranschaulichung der abstrakten Überlegungen dient das Beispiel einer zu validierenden Mobilfunkinfrastruktur. Eine Implementierung der Testbench für eben dieses Beispiel wird ebenso vorgestellt, wie die darauf basierenden Evaluierungen des gesamten Konzeptes.

1.2. Aufbau dieser Arbeit

Die gesamte Arbeit stellt ein allgemein anwendbares Validierungskonzept vor und erläutert dieses anhand des Beispiels Mobilfunknetzwerk.

Im folgenden Kapitel werden zunächst Grundlagen diskutiert. Im Mittelpunkt steht die Klärung der Begriffe Validierung, Verifikation und Testen, der Kategorisierung der Tests sowie die Definition der Klasse der Dienstbereitstellenden Systeme. Desweiteren wird die Technik zellulärer Mobilfunknetze vorgestellt.

Ein Überblick über Vorarbeiten und den wissenschaftlichen bzw. technischen Stand wird in Kapitel 3 gegeben. Im Mittelpunkt stehen Verfahren zum automatischen Testen und zu Leistungstests, wie sie im klassischen Entwurf von Softwaresystemen zum Einsatz kommen. Auch Ansätze zum Test hardwarelastiger Systeme sowie Verfahren für zelluläre Mobilfunknetze werden erläutert. Unabhängig davon werden Algorithmen zur Lösung kombinatorischer Optimierungsprobleme sowie zum Scheduling in Parallelrechnern vorgestellt, da diese Ähnlichkeiten zum in dieser Arbeit ausführlich betrachteten Testpartitionierungsproblem haben.

Das eigentliche Validierungskonzept dieser Arbeit und dessen Einordnung in den Entwurfsprozeß werden in Kapitel 4 eingeführt. Aus den Anforderungen ergibt sich ein Modell zur Beschreibung der einzelnen Komponenten und der Tests sowie ein mehrteiliges Testbenchsystem, das zur automatisierten Ausführung des Modells verwendet werden kann.

Kapitel 5 beschäftigt sich ausführlich mit der zur Automatisierung notwendigen Testpartitionierung. Das Problem wird formuliert und Lösungsansätze vorgestellt. Im Falle dieser Arbeit kommt eine eigenentwickelte Heuristik zum Einsatz, die in ihrer Funktion erläutert wird. Zur Evaluierung werden verschiedene Testreihen präsentiert, die zum Vergleich auch mit einem Optimierungsalgorithmus in Matlab gelöst wurden.

Die Implementierung des Beispielssystems zum Test des Mobilfunknetzwerkes steht im Mittelpunkt von Kapitel 6. Zum Überprüfen des Konzeptes dieser Arbeit wird das Beispielsystem mit einem Simulator gekoppelt. Die durchgeführten Testreihen für statische und dynamische Szenarien sind in Kapitel 7 dargestellt. Die Messergebnisse sowie die Konzepte werden qualitativ und quantitativ diskutiert.

Eine Zusammenfassung über das vorgestellte Konzept sowie ein Ausblick über denkbare Weiterentwicklungen schließen die Arbeit ab.

2. Grundlagen

Das Testen von Systemen umfaßt eine Vielzahl an Kategorisierungen, Ansätzen und Begrifflichkeiten. Besonders verschiedene Domänen und Architekturen stellen unterschiedliche Anforderungen an den „Test“ und bieten verschiedene Konzepte und Technologien zur Umsetzung. Die Aufteilung in Abstraktions- und Komplexitätsniveaus der zu testenden Entitäten sowie die Kategorisierung nach verschiedenen technischen oder ökonomischen Zielsetzungen eröffnen weitere Dimensionen, die eine vollständige Betrachtung des Themas Testen schwierig machen.

In diesem Kapitel werden deshalb dieser Arbeit zugrundeliegende Begriffe und Verfahren des Entwurfsprozesses definiert und bei Bedarf erläutert. Die Klasse der Dienstbereitstellenden Systeme wird eingeführt und kategorisiert.

Ergänzt wird das Kapitel um Ausführungen zu zellulären Mobilfunknetzen und deren Einordnung als Dienstbereitstellendes System. Mobilfunknetze dienen als Beispiel zur Veranschaulichung der Konzepte dieser Arbeit.

2.1. Testen, Verifizieren, Validieren

Wie bereits in der Einführung erläutert, stehen in erster Linie interagierende, softwarelastige und damit zustandsbasierte Systeme im Fokus dieser Arbeit. Demzufolge liegt es nahe, Verfahren und Begrifflichkeiten aus dem Bereich des Softwaretests hierfür heranzuziehen. Es ist hilfreich, daß dieser Bereich des Testens, der häufig als eine Disziplin des Sofwareengineerings gesehen wird, einige anwendungsfeldabhängige aber anerkannte Verfahren, Normen oder Standards hervorgebracht hat.

Hierzu sollten zunächst alltägliche Begriffe in ihrer genauen Bedeutung definiert werden. Begriffe wie Testen, Verifizieren oder Validieren werden im akademischen und betrieblichen Alltag häufig verwendet und intuitiv verstanden. Allerdings fällt eine Abgrenzung zwischen ihnen relativ schwer. Da die Begriffe auch in der Primärliteratur, vermutlich begünstigt durch Ungenauigkeiten von Übersetzungen, nicht einheitlich verwendet werden, bietet zumindest der IEEE[1] - Standard 610.12-1990, Standard Glossary of Software Engineering Terminology, eine internationale und anerkannte Terminologie englischsprachiger Begriffe im Softwareengineering ([1]).

---

[1] Institute of Electrical and Electronics Engineers

Allgemein wird ein „Test“ im IEEE Glossar wie folgt definiert ([1], S. 74):

> (1) An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.
>
> (2) To conduct an activity as in (1).

Wörtlich ist ein Test also ein Vorgang, bei dem ein System, oder Teile davon, unter wohldefinierten Randbedingungen ausgeführt wird. Die Ergebnisse werden überwacht und bezüglich einiger Aspekte des System oder dessen Komponenten beurteilt.

Die wörtliche Übersetzung wirft die Frage auf, was unter einer „Ausführung eines Systems“ und was unter der „Beurteilung“ zu verstehen ist. Hierzu kann eine weitere, etwas spezifischere Definition des Vorgangs „Testen“ herangezogen werden (Vgl. [62], S. 6):

> Testing is the process of executing a program with the intent of finding errors.

Übersetzt ist Testen also „[...] die Ausführung eines Programms mit dem Ziel, Fehler zu finden“ ([77], S. 23).

Diese Definitionen bezieht sich nur auf einen Teil des Systems: das „Programm“ - also einer sequentiellen Folge von Befehlen. Der „Beurteilungsaspekt“ aus erstgenannter Definition wird in der zweitgenannten zur Suche nach „Fehlern“.

In Bezug auf Tests ist unter einem Fehler folgendes zu verstehen ([77], S.24):

> [...] Damit wäre ein Fehler die Abweichung des Programmverhaltens von dem in der Spezifikation vorgeschriebenen Verhalten.

Zusammengefasst ergibt die Kette der Definitionen, daß ein Test also die Ausführung eines (Teil-)Systems unter wohldefinierten Randbedingungen ist - mit dem Ziel, Abweichungen des realen Systemverhaltens von dem in der Spezifikation vorgeschriebenen Verhalten zu finden. Der Test ist ein Verfahren bzw. eine Methode zur Generierung und Bewertung von Ist-Verhalten in Relation zum Soll-Verhalten.

Es stellt sich die Frage, woher die Definitionen oder Maßgaben für das Soll-Verhalten abgeleitet werden. Wird der Systementwurf als ganzes betrachtet, unterscheidet der Forschungsbereich der Qualitätssicherung zwischen (nach [88], S. 207f):

- getting the right product
- getting the product right

Dieses Wortspiel zielt genau auf die Generierung des Soll-Verhaltens und den daraus folgenden Entwurfsvorgang ab. Der Normalfall „getting the product right“ (das Produkt richtig machen) ist das Qualitätsziel, das Soll-Verhalten einer gegebenen Spezifikation korrekt umzusetzen.

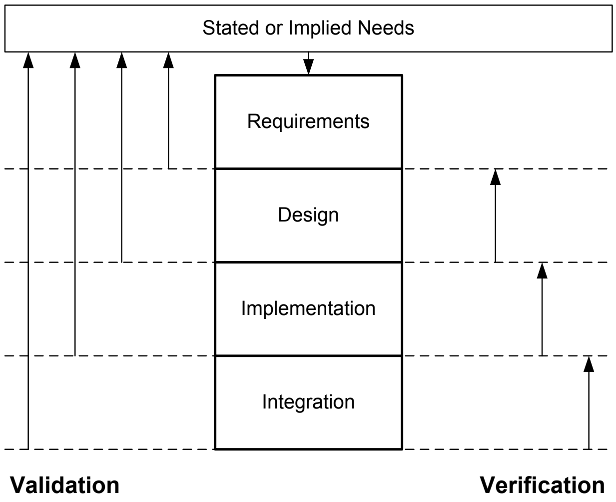


Abbildung 2.1.: Validierung und Verifikation im Entwurfsprozeß (Vgl. [88], S. 208)

Demzufolge wird auch das Soll-Verhalten des zu testenden Systems aus der Spezifikation abgeleitet. Alle Aktivitäten zur Sicherstellung der korrekten Spezifikationsimplementierung, die auf verschiedenen Abstraktionsniveaus des Entwurfs durchgeführt werden können, werden Verifikation genannt (Vgl. [88], S.208; [1], S. 81). Häufig wird verifiziert, ob die Ergebnisse einer Phase des Entwicklungsprozesses die Anforderungen erfüllt, die durch die vorhergehende Phase definiert wurden (nach [1], S. 81).

Validierung beschäftigt sich hingegen damit, sicherzustellen, daß das „richtige Produkt“ erstellt wird (Vgl. [88], S.207; [1], S. 80). Dies bezieht sich auf die Wünsche und Ideen des Kunden oder Anfordernden und damit nicht zwingend auf erstellte Spezifikationsdokumente. Vielmehr soll durch Validierung die Einhaltung von Anforderungen auf Systemebene gewährleistet werden (Vgl. [88], S. 209). Diese beinhalten ebenso die „Behandlung aufkommender Eigenschaften [während des Entwurfs] und die Identifizierung impliziter Anforderungen, die zu Beginn nicht formuliert wurden“ (nach [88], S.209). In Folge dessen ist eine umfassende Validierung häufig nur in einem fertiggestellten bzw. vollständigen System möglich (Vgl. [88], S. 209).

Bestenfalls gelingt es im Rahmen der Spezifikation des zu entwickelnden Systems, alle Anforderungen des Kunden zu erfragen und mehr oder weniger formal niederzuschreiben. Aus diesen Anforderungen können dann technologiebasierte Spezifikationen erstellt werden. Könnte dieser Vorgang als vollständig zuverlässig angesehen werden, so wären Verifikation und Validierung in einem einzigen Prozeß vereinbar. Aus vielerlei Gründen ist es jedoch langfristig unwahrscheinlich, einen solchen zuverlässigen Spezifikationsprozeß zu finden.

Das Testen steht als ein Mechanismus zur Verifikation und Validierung zur Verfügung. Es dürfte sich wohl, in welcher Ausprägung auch immer, um den gebräuchlichsten Mechanismus handeln und ist für eine zielgerichtete Überprüfung entwickelter Systeme unerlässlich.

Jedoch genügt das Testen in vielen Fällen als einziges Verfahren nicht ([14], S. 3):

> Testing is not the only validation method you should use [...].

2.2. Testkategorisierung

Tests als Mittel zur Verifikation und Validierung können nach verschiedenen Kriterien kategorisiert werden, die aber voneinander abhängig sind.

Eine der wichtigsten Charakterisierungen ist die Sicht auf das System. Ähnlich wie in den einschlägigen Entwurfsprozeßmodellen, bspw. dem Y/X-Diagramm (Vgl. [68], S. 15), dem Double-Roof Modell (Vgl. [86], S. 10) oder dem P-Chart (Vgl. [45], S. 48f) wird hier zwischen der Funktionalen[2] und der Strukturellen[3] Sicht unterschieden.

Im Falle der Funktionalen Sicht wird das Verhalten des Systems bewertet. Dazu werden Testfälle, also Kombinationen aus Eingaben und erwarteten Ausgaben, in erster Linie aus der Spezifikation abgeleitet, da diese die Anforderungen an das Verhalten eines Systems (oder Teilsystems) beschreiben muß (Vgl. [77], S. 26f, S.92ff). Details über Architektur oder Implementierung des Systems sind hierfür nicht relevant bzw. sogar nicht erwünscht. Demzufolge sind Informationen über Interna auch nicht notwendig, so daß der Name Black-Box-Test hierfür gebräuchlich ist.

Wird die Struktur der Implementierung in die Definition von Testfällen einbezogen, so wird von White-Box-Tests[4] gesprochen (Vgl. [77], S. 27, S. 104ff; [87], S. 61ff). Testfälle werden hier in erster Linie genutzt, um Eigenschaften bzw. Parameter der Implementierung zu prüfen. Im Falle zu testender Software ist es bspw. das Ziel, möglichst viele Kontrollflußpfade zu durchlaufen. Die Granularität der Pfade richtet sich nach der Testebene (Code, Modul, ...). Zur Generierung entsprechender Testfälle sind Strukturkenntnisse unabdingbar. Ziel ist, möglichst jeden Teil der Logik (Programm, Software) zu durchlaufen, da in jedem Schritt/ Zweig Fehler entstehen können. Dies trifft auch auf andere Domänen außerhalb des Software-Tests zu.

Beide Strategien des Tests sind einzusetzen (Vgl. [14], S. 8) und haben für ihr jeweiliges Abstraktionsniveau im Entwurf entsprechende Bedeutung. Auf der Ebene von Systemtests, wo also das komplette System im Zusammenspiel aller Einzelkomponenten zu überprüfen ist, ist jedoch fast ausschließlich Black-Box-Testen üblich. Insbesondere komplexe und heterogene Systeme erlauben kaum domänenübergreifende White-Box-Tests.

Neben der Unterscheidung nach Sichten auf das zu testende (Teil)System ist auch die Unterscheidung nach dem Abstraktionsniveau der betrachteten Systeme üblich. Implizit ergibt sich daraus auch die Einordnung in den kompletten Entwurfsprozeß. Im Bereich der Softwareentwicklung hat sich das V-Modell als eines der gebräuchlichsten Prozeßmodelle etabliert, das die einzelnen Entwurfs- und Testschritte iterativ anordnet und in Relation zueinander setzt.

Eine graphische Veranschaulichung des V-Modelles, das 1979 erstmalig veröffentlicht wurde ([16]) und mittlerweile durch verschiedene Adaptionen in vielen Bereichen zum Einsatz

---

[2] Englisch: Behaviour

[3] Englisch: Structure

[4] Auch eine weiße Box verhindert das Hineinblicken - deshalb wird häufig auch der Begriff Glass-Box-Test (Vgl. [35], S. 57) verwendet

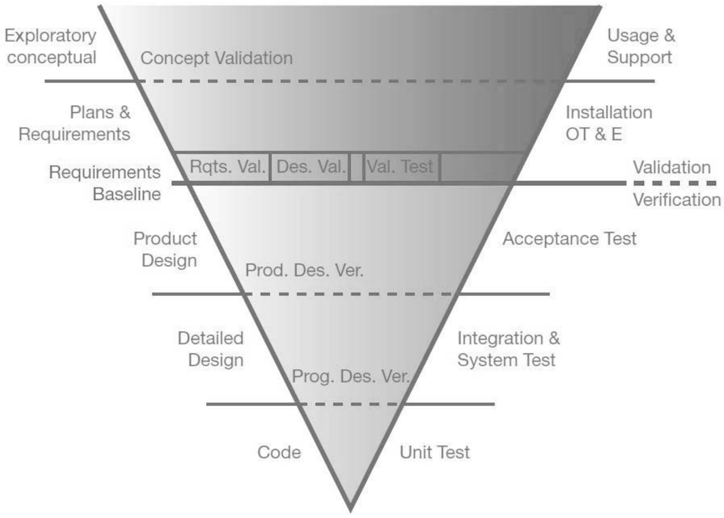


Abbildung 2.2.: V-Modell ([40], S. 375)

kommt, ist in Abbildung 2.2 zu sehen (Vgl. [40], S. 375f). Kerngedanke bildet die Teilung des Entwurfsprozesses in eine Realisierungsphase, die sowohl Spezifikation wie auch Implementierung beinhaltet (linker Schenkel des „V“), und eine Verifizierungs- bzw. Inbetriebnahmephase (rechter Schenkel). Beide Phasen werden in verschiedene Schritte unterteilt, die sequentiell ausgeführt werden.

Entscheidend ist die Aussage, daß die beiden Stränge abhängig voneinander sind. Auf jeder Ebene werden bereits in der Realisierungsphase die zugehörigen Tests der Verifikationsphase definiert und diese dann in der Verifikationsphase ausgeführt. Damit wird das Testen als Methodik zur Verifikation bzw. zur Validierung von Systemen deutlich aufgewertet, da es nicht mehr als „notwendiges Übel“ im Anschluß an die Implementierung gesehen wird, sondern als elementarer Bestandteil des gesamten Entwurfsprozesses, der zur Sicherstellung der Qualität des Systems zwingend notwendig ist. Abbildung 2.3 verdeutlicht noch einmal den Einfluß des Testens auf den Entwurfsprozeß.

Desweiteren wurden mit dem V-Modell erstmalig die hierarchischen Teststufen (auch Testzyklen genannt) eingeführt (Vgl. [40], S. 376). Diese verdeutlichen, daß verschiedene Schritte in der Implementierung und Inbetriebnahme auch verschiedene Testansätze und -fälle bedingen.

Das V-Modell unterscheidet zwischen vier Stufen. Die feingranularste ist der Komponententest[5]. Dieser sieht den isolierten Test einzelner Modellierungsprimitive vor. Je nach Programmiersprache werden diese Tests bspw. an Unterprogrammen, Klassen oder Modulen durchgeführt und umfassen Prüfungen auf Lauffähigkeit und korrekt umgesetzte Logik. Komponententests werden häufig vom Entwickler selbst durchgeführt. Für einige Programmiersprachen und Entwicklungsumgebungen

---

[5] Englisch: Unit Test

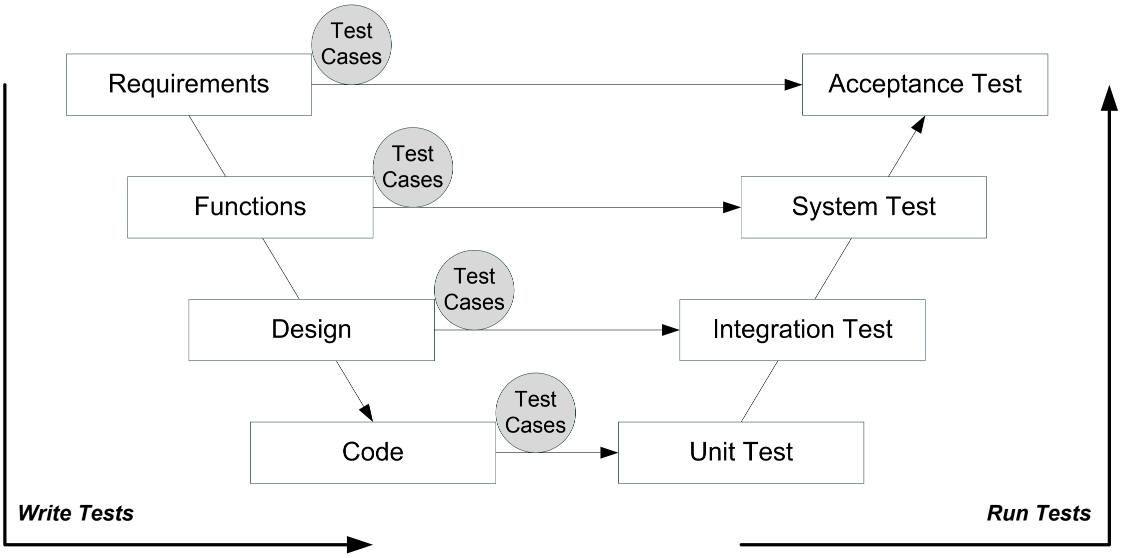


Abbildung 2.3.: Testfälle erzeugen und ausführen laut V-Modell (Vgl. [32], S. 7)

existieren auch Programme für eine teilautomatisierte Erstellung und Durchführung der Komponententests. Ein Beispiel für die Programmiersprache Java ist JUnit (Vgl. [48]).

Nachdem die Komponenten einzeln und isoliert getestet wurden, wird im Integrationstest das Zusammenspiel verschiedener Konstellationen abhängiger Komponenten überprüft, wobei der Test von Syntax und Semantik der Schnittstellen im Vordergrund steht.

Beim anschließenden Systemtest wird das System komplett integriert und überprüft. Die zu prüfenden Anforderungen werden meist der Spezifikation entnommen, wobei dies bereits in der Realisierungsphase geschehen sein sollte. Die Tests umfassen sowohl die Erfüllung funktionaler wie nichtfunktionaler Anforderungen und werden üblicherweise durch den Systemhersteller durchgeführt. Um insbesondere nichtfunktionale Anforderungen sinnvoll bewerten zu können, wird versucht, die spätere Produktivumgebung möglichst genau nachzubilden.

In den genannten Teststufen ist es stets das Ziel, durch strukturierte Testprozesse Fehler in den Entwürfen zu finden. Im Gegensatz dazu dient die letzte Stufe des V-Modells, der Abnahmetest, nicht mehr dem Auffinden von Fehlern. Vielmehr will der Hersteller in dieser Stufe die Funktionsfähigkeit und in gewissen Grenzen auch die Fehlerfreiheit seines Produktes demonstrieren, während der Auftraggeber sicherstellen will, daß das System den vorher festgelegten Anforderungen entspricht. Häufig wird das System dazu bereits in der Produktivumgebung eingesetzt, wird aber noch nicht oder nur eingeschränkt von den künftigen Nutzern verwendet.

Abhängig vom jeweiligen Projekt lassen sich die einzelnen Teststufen nicht immer sauber voneinander abtrennen. Vor allem bei sehr komplexen Systemen werden häufig System- und Abnahmetest kombiniert. Häufig kommen neben reinen Labortests, in denen das System also unter statischen, kontrollierbaren Umweltbedingungen getestet wird, auch Feldtests zum Einsatz. Hier wird das System unter weitestgehend produktiven Bedingungen überprüft. Die Zahl der Nutzer wird häufig eingeschränkt und diese auch auf die Eigenarten des Systems hin geschult. Zum Teil werden auch Nutzungsauflagen erteilt, so daß Nutzer - im Gegenzug für Vergünstigungen - bspw. Evaluierungsberichte erstellen müssen.

Der Begriff „Testen“ ist stark an die Bedeutung der Funktionsüberprüfung von Software gebunden. Unabhängig davon ist jedoch jedes entwickelte System, ganz gleich welcher Ingenieursdisziplin(en), zu verifizieren und zu validieren. Entsprechend werden auch bei Kompositionssystemen ähnliche Teststufen Anwendung finden, wie es beim V-Modell in der Softwaretechnik vorgeschlagen wird. Handelt es sich bei dem Gesamtsystem um ein heterogenes System, das aus vielen Komponenten verschiedener Ingenieursdisziplinen besteht, so ist mit großer Wahrscheinlichkeit auch ein programmgesteuertes, zustandsdiskretes Teilsystem enthalten, das Überwachungs- und Kontrollaufgaben übernimmt oder elementare Funktionen des Systems überhaupt erst bereitstellt. In diesem Falle beeinflussen sich die Teststufen derart, daß ein vollständiger Systemtest erst nach Komponenten- bzw. Integrationstests der Teilsysteme möglich ist.

2.3. Dienstbereitstellende Systeme

Grundlage dieser Arbeit und des präsentierten Konzeptes zur Validierung bildet der Begriff der *Dienstbereitstellenden Systeme*. Dieser scheint zunächst intuitiv klar, wird jedoch zum allgemeinen Verständnis im Folgenden detailiert erläutert.

2.3.1. Dienst

Der Begriff Dienst[6] wird in verschiedenen Bereichen der Informatik für verschiedene Aspekte und auf verschiedenen Abstraktionsebenen genutzt. Ein Web*service* ist bspw. nicht mit einem *Dienst* bzw. *Daemon* eines Betriebssystems zu vergleichen.

Allgemein ist unter einem Dienst eine eigenständige, logische Einheit zu verstehen, die in der Lage ist, die Abarbeitung „einer fest umrissenen Aufgabe“ verschiedenen Dienstnutzern zur Verfügung zu stellen (Vgl. [23], S. 198; [31], S. 33). Meist handelt es sich bei dieser Aufgabe um Möglichkeiten zur Manipulation, Gewinnung oder Speicherung von Informationen. Auch die Kommunikation, egal über welches Medium oder welche Technologie, ist aus Sicht des Klienten als Dienst aufzufassen.

Abhängig von der bereitstellenden Domäne kann ein Dienst eine „Sammlung von Funktionen“ oder auch ein im „Hintergrund aktiver Prozeß“ (Vgl. [33], S. 225) sein. Während es sich bei der Funktionssammlung umgangssprachlich eher um eine Bibliothek handelt, sind die „Prozesse“[7] die im Rahmen dieser Arbeit bevorzugte Realisierung eines Dienstes.

---

[6] Englisch: Service

[7] Im folgenden *Aktivtäten* genannt

Die Funktionalität eines Dienstes umfaßt in aller Regel keine komplette Applikation. Vielmehr werden abgeschlossene Teilaufgaben bzw. -funktionen in Diensten gekapselt, die dann durch andere Applikationen oder auch andere Dienste genutzt werden können[8]. Meist werden Dienste so ausgelegt, daß sie quasiparallel von mehreren Klienten genutzt werden können. Handelt es sich beim Dienst um eine Aktivität, können jedoch die zur Ausführung benötigten Betriebsmittel die Parallelnutzung einschränken.

Der Aufruf von bzw. der Zugriff auf Dienste ist über definierte Schnittstellen möglich. Eine hinreichende Spezifikation oder Dokumentation der Schnittstellen ist notwendig, um auch externen Entwicklern von Klienten die Nutzung der Dienste zu ermöglichen. Neben der eigentlichen Funktion des Dienstes müssen auch Übergabe- und Rückgabeparameter, Fehlercodes und auch das Zeitverhalten angegeben werden. Da Dienste und Klienten meist auf verschiedene Systeme verteilt sind, werden auch Angaben zu unterstützten Kommunikations- oder Aufrufprotokollen benötigt.

Läuft ein Dienst als Aktivität, erlaubt er den Zugriff auf interne Zustände bzw. Daten nur über die definierten Schnittstellen. Laufen mehrere Dienste parallel auf einer Plattform, schützt das verwendete Betriebssystem den Speicher der einzelnen Dienste genau wie den von Applikationen.

Die Analyse der genannten Eigenschaften legt den Vergleich zum wohlbekannten Unterprogrammkonzept von Hochsprachen nahe. Auch hier werden einzelne Funktionen bzw. Teilaufgaben in Funktionen, Prozeduren oder Methoden gekapselt, die dann von der Applikation, zu der die Unterprogramme gehören, genutzt werden können. Hierzu verfügen Unterprogramme über eine Schnittstelle, die je nach verwendeter Programmiersprache mehr oder minder typenrein definiert ist. Im Gegensatz zu Diensten, die als Aktivität laufen, sind Unterprogramme jedoch Bestandteil der Applikation selbst und verfügen weder über eigene Betriebsmittel noch über geschützte Speicherbereiche. Ein paralleler Betrieb und asynchrone Aufrufe, wie sie bei Diensten und Klienten üblich bzw. sogar notwendig sind, sind bei Unterprogrammen kaum möglich. Spezielle Programmierparadigmen und -technologien, bspw. die Objektorientierte Programmierung, erlauben aber zumindest die logische Kapselung der Daten und Zustände von Unterprogrammen.

Ein Dienst im Sinne dieser Arbeit erfüllt die o.g. Eigenschaften und Anforderungen.

2.3.2. Dienstbereitstellendes System

Dienstbereitstellende Systeme (SPS[9]) im Sinne dieser Arbeit sind eine Kombination aus einem bzw. mehreren Diensten und entsprechenden physischen oder logischen Ressourcen, die alle Betriebsmittel zur Ausführung der Dienste und zur Kommunikation mit Klienten bereitstellen. Die kombinierten Dienste haben hinsichtlich ihrer Funktionen meist semantische Gemeinsamkeiten, so daß das SPS eine hierarchische und logische Zusammenfassung dieser Funktionen darstellt. Unter Umständen werden weitere, interne Dienste benötigt, die für externe Klienten nicht zugreifbar sind aber für die Bereitstellung weiterer externer Dienste benötigt werden.

---

[8] Dienstnutzer werden im folgenden *Klienten* genannt

[9] Englisch: Service Providing System

Bei den physischen Ressourcen muß es sich nicht zwangsläufig um eine einzelne Plattform, wie bspw. einen Server, handeln, auf der die Dienste als Aktivitäten laufen. Vielmehr kann die Systemarchitektur eine Menge stark verteilter und heterogener Komponenten sein, auf denen auch die Dienste verteilt ausgeführt werden können. Auch dynamische Architekturen, die zusätzliche Ressourcen je nach Betriebszustand allokieren können, sind denkbar. Der Aufbau und die Funktion des SPS ist für Klienten allerdings transparent, da diese lediglich über die Schnittstellen auf die einzelnen Dienste zugreifen.

Die einzelnen Dienste des SPS arbeiten logisch unabhängig voneinander und greifen auf Zustände bzw. Daten anderer nur über die dafür vorgesehenen Schnittstellen zu. Allerdings kann die Bindung an Ausführungssysteme und deren Ressourcen sowie mögliche Nutzung weiterer Dienste zu Betriebsmittelkonkurrenzen zwischen diesen führen, die nur durch zeitliches Multiplexing aufgehoben werden können und damit zu indirekten Abhängigkeiten führen.

Hauptaufgabe eines SPS besteht in der Bereitstellung der Dienste für die Nutzung durch Klienten. Erfordert das Einsatzgebiet parallele Nutzung durch mehrere Klienten, müssen entsprechende Ressourcen durch Entwickler bestimmt und zur Verfügung gestellt werden, so daß eine angemessene Qualität der Dienstbereitstellung gewährleistet werden kann.

Basierend auf dieser Überlegung ist es für das Konzept dieser Arbeit notwendig, eine weitere Anforderung an ein SPS zu stellen. Nutzt ein Klient einen Dienst des SPS, so muß diese Nutzung quantitativ oder qualitativ bewertbar sein. Es muß also möglich sein, jeder Art von Dienstaufruf einen sinnvollen Wert zuzuordnen, der beschreibt, wie stark der Klient den Dienst mit dem Aufruf belastet. Dieses absolute Maß wird im folgenden als „*Last*“ bezeichnet.

Die Last eines Dienstes kann theoretisch mit jeder positiven, rationalen Zahl beschrieben werden. Da es sich sowohl bei den SPS als auch den Klienten in aller Regel um digitale und damit zustandsdiskrete Systeme handelt, wird ein Großteil der praktischen Anwendungsfälle auch zu diskreten Lastwerten führen. Es wird deshalb die Festlegung getroffen, daß die Last ${}^{t}l_{s,i}$, die ein Klient s am Dienst i zum Zeitpunkt t generiert, eine natürliche Zahl ist: ${}^{t}l_{s,i} \in \mathbb{N}$. Die Definition oder Ableitung der Last für bestimmte Dienstaufrufe ist stark vom jeweiligen Anwendungsszenario bzw. auch Betriebmodus abhängig. Da es sich um ein Maß für die Leistungsbelastung eines Dienstes handelt, muß die Quantifizierung auch die realen Verhältnisse dieser Belastung abbilden.

Die Gesamtlast für den Dienst ergibt sich aus der Summe der Einzellasten aller Klienten, die zum Zeitpunkt t zugreifen. Greift kein Klient zu, befindet sich der Dienst im Leerlauf und die Last sollte offensichtlich einen Wert von 0 haben bzw. minimal sein. Andererseits sind dem Dienst durch begrenzte Ressourcen der Plattform des SPS Grenzen in der qualitativen oder quantitativen Diensterbringung gesetzt. Dies entspricht einer maximalen Last. Entsprechend anteilig sollten auch die Lastwerte für Dienstanfragen einzelner Klienten gewählt werden.

2.3.3. SPS und SOA

SPS ist ein Begriff, der zur Systemklassifikation im Rahmen dieser Arbeit eingeführt wurde. Im Gegensatz dazu fand der Begriff der *Dienstorientierten Architekturen*[10] (SOA) in den letzten Jahren weite Verbreitung in der Informatik. Auch wenn die Bezeichnungen Dienstbereitstellende Systeme und Dienstorientierte Architekturen ähnlich klingen, bestehen deutliche Unterschiede, die im folgenden kurz erläutert werden.

Eine einheitlich anerkannte Definition für SOA existiert nicht. Allerdings wird häufig die 1996 veröffentlichte Definition der OASIS[11] herangezogen. Diese besagt, SOA sei ein Paradigma zur Strukturierung und Anwendung verteilter Ressourcen, die u. U. von verschiedenen Besitzergruppen kontrolliert werden (Vgl. [60], S. 8). Die Strukturierung der Ressourcen erfolgt hinsichtlich eines Kontextes. Häufig wird dieser im Bereich von Geschäftsprozessen angesiedelt (Vgl. [31], S.33).

Bei SOA handelt es sich also um ein Architekturkonzept zur Anwendungsentwicklung. Ziel ist es, die Anforderungen der Anwendung („needs“, Vgl. [60], S. 8) auf verschiedene Dienste („capabilities“) abzubilden und diese durch Komposition so zu verknüpfen, daß die Anforderungen erfüllt werden (Vgl. [60], S. 8).

Ein SPS, wie es in dieser Arbeit vorgestellt wird, ist im Gegensatz dazu kein Architekturkonzept sondern stellt eine Klasse von Systemen dar. Diese umfassen die physischen Plattformen und Dienste, die unter Nutzung dieser Plattformen verwendet werden. Applikationen der Klienten und die Relationen, wie diese die Dienste nutzen, sind für die Klassifizierung irrelevant. Vereinfacht kann gesagt werden, daß ein SPS nur beschreibt, welche Dienste „bereitgestellt“ werden, wohingegen SOA auch Konzepte zu deren Nutzung umfassen.

Das Konzept der SPS beschreibt Dienste als logische Entitäten. Auch die Plattformen, auf denen diese Dienste ausgeführt werden, sind nur abstrakt als Menge an Betriebsmitteln definiert. Konkrete Vorgaben oder Anforderungen an die technische Umsetzung werden nicht gestellt. Gleiches trifft auf SOA zu. Auch hierbei handelt es sich „nur“ um ein abstraktes Paradigma für eine verteilte Architektur (Vgl. [31], S. 2). Nur in Kombination mit weiteren, detailiert spezifizierten Implementationskonzepten, wie bspw. SOAP und den darauf aufbauenden Web-Services (Vgl. [31], S. 73), ergeben sich konkrete Umsetzungsmöglichkeiten.

2.4. Ein Beispiel: Zelluläre Netze

Es existieren zahlreiche Beispiele für Dienstbereitstellende Systeme. Ein naheliegendes ist ein Internetserver. Es ist zu beachten, daß im alltäglichen Sprachgebrauch sowohl die auf Dauerbetrieb ausgelegte Hardwareplattform als auch die einzelnen Dienste, wie HTTP oder FTP, als Server bezeichnet werden. Im Sinne von SPS wird jedoch die Kombination aus beiden betrachtet. Dieser

---

[10] Englisch: Service-Oriented Architectures

[11] Organization for the Advancement of Structured Information Standards

Internetserver stellt die Serverdienste Klienten zur Verfügung, die über definierte Protokolle auf diese zugreifen können. Klienten sind bspw. Arbeitsplatzrechner, die über ein hierarchisches Netzwerk mit dem Server kommunizieren können.

Wird ein Dienst durch einen Klienten gerufen, verursacht dieser Aufruf eine Last. Diese Last kann verschieden quantifiziert werden. Denkbar ist bspw. im Falle des Dienstes FTP, jede zum Zeitpunkt t bestehende Verbindung zwischen Klient c und Server mit einer pauschalen Last ${}^{t}l_{c,FTP} = 1$ zu bewerten. Diese einfache Quantifizierung berücksichtigt allerdings nur bedingt die interne Betriebsmittelauslastung des Servers. So kann ein Klient, der über ein lokales, breitbandiges Netz angebunden ist, beim Herunterladen einer Datei wesentlich höhere Bandbreiten nutzen und damit sowohl den Server als auch das Kommunikationsnetz zwischen Klient und Server stärker belasten als das einem Klienten möglich ist, der über ein schmalbandiges Mobilfunknetz zugreift. In Abhängigkeit der Rahmenbedingungen ist daher eine Lastquantifizierung mittels der zur Verfügung stehenden bzw. genutzten Bandbreite denkbar und sinnvoll.

Aus Sicht des Klienten ist die Struktur des SPS unbekannt. Realisiert werden kann ein solcher FTP-Server sowohl als einzelner Rechner als auch als ein verteiltes System, auf dem bspw. das Dateisystem in ein externes Speichersystem ausgelagert ist. Für den Klienten ist dies nicht bekannt und auch nicht relevant. Allerdings ist es durchaus wahrscheinlich, daß der FTP-Serverdienst wiederum interne Dienste der Plattform nutzt. Beispielsweise werden Betriebssystemfunktionen zum Zugriff auf das lokale Dateisystem verwendet. Folglich kann auch die hierarchische Kombination aus Rechner und Betriebssystem wieder als SPS betrachtet werden und der lokale FTP-Serverdienst als entsprechender Klient.

Das genannte Beispiel des Internetservers ist einfach, anschaulich und durch weitere Protokolldienste auch leicht erweiterbar. Als zentrales Beispiel eines SPS wird im Rahmen dieser Arbeit ein darauf aufbauendes, komplexeres System dienen.

Wie bereits erwähnt geht die Motivation für diese Arbeit auf eine Industriekooperation mit einem Hersteller von Mobilfunkinfrastruktur zurück. Dieser steht vor der Herausforderung, daß Software- und Hardwareänderungen an einer oder mehrerer Komponenten des komplexen Mobilfunksystems detailliert auf Funktionsfähigkeit aber auch auf Leistungsfähigkeit getestet werden müssen. Können nach einem Update des beim Kunden produktiven Systems vertraglich geregelte Leistungswerte nicht erbracht werden, drohen erhebliche Vertragsstrafen und damit schwerer betriebswirtschaftlicher Schaden. Entsprechende Tests zur Ermittlung der Leistungsparameter sind also bereits vor dem Update auf das eigentliche Kunden- bzw. Produktivsystem wünschenswert. Um die Herausforderung dieser Art der Validierung zu verstehen, ist ein Einblick in die Architektur aktueller Mobilfunknetzwerke notwendig.

2.4.1. Entwicklung der Mobilfunksysteme

Prinzipiell handelt es sich bei Mobilkommunikation um eine „Form der elektronischen Kommunikation über räumliche Distanzen“, bei der „drahtloser - also leiterungebundener - Datenaustausch zwischen ortsveränderlichen Kommunikationspartnern“ ([57], S.9) ermöglicht wird. Technologisch haben sich Systeme durchgesetzt, bei denen nur die nutzerseitigen Endgeräte drahtlos und mobil sind. Die nötige Infrastruktur zur Bereitstellung der elektronische Kommunikation wird jedoch in Form ortsunveränderlicher und meist kabelangebundener Basisstationen realisiert.

Erste, analoge Systeme dieses Bereichs ähnelten stark den Systemen zur Radio- und Fernsehübertragung. Wenige starke Sender werden räumlich günstig errichtet und übernehmen die Verbindung zu allen mobilen Endgeräten. Diese Architektur bringt allerdings viele Einschränkungen. Zum einen ist es durch die physikalische Ausbreitung elektromagnetischer Wellen schwierig, die Erreichbarkeit der Mobilgeräte bzw. des Senders großflächig bzw. in schwieriger Topographie zu gewährleisten. Zum zweiten werden - im Gegensatz zu Rundfunksendern - für jeden Nutzer des Netzes auf Seiten des Senders auch Ressourcen, wie bspw. der Zugriff auf Telefonleitungen, benötigt. Diese können nicht in beliebiger Zahl zur Verfügung gestellt werden, so daß die maximale Anzahl der Nutzer stark beschränkt ist. Das erste dieser Art in Deutschland betriebene Mobilfunksystem war das A-Netz, welches 1958 gestartet wurde. Da es sich hierbei noch um ein handvermitteltes Netz handelte, war auch ein Wechsel des Senders während eines Gesprächs nicht möglich (Vgl. [57], S. 27).

Um diese Mängel zu beseitigen, wurde die Architektur der zellulären Netze eingeführt. Hier wird das abzudeckende Areal in Funkzellen aufgeteilt, die jeweils über eine eigene Basisstation verfügen. Jede der Funkzellen verfügt über einen eigenen Satz an physikalischen Frequenzen oder logischen Kanälen, die überlappungsfrei zu Nachbarzellen sind und somit auch in Übergangsbereichen eine zuverlässige Kommunikation zwischen Endgerät und Basisstation erlauben.

Durch die Verkleinerung der Zellen kann zum einen die physikalische Netzabdeckung gesteigert werden, und zum anderen müssen die lokalen Ressourcen auch nur den lokal anwesenden Nutzern zur Verfügung gestellt werden. In Summe aller Funkzellen kann somit die Zahl der möglichen Nutzer massiv gesteigert werden. Als positiver Nebeneffekt kann durch die geringen Distanzen zwischen Mobilgerät und Basisstation auch die Sendeleistung deutlich reduziert werden, so daß der Energiebedarf der Mobilgeräte geringer ausfällt. Die damit einhergehende Verkleinerung der Batterien und Endgeräte waren vermutlich ein entscheidender Faktor für den Erfolg der Mobilkommunikation. Während das deutsche A-Netz 1970 über 11.000 Teilnehmer (Vgl. [57], S. 27) verfügte, mußten in allen digitalen Mobilfunknetzen Deutschlands im Jahre 2010 bereits Ressourcen für 108,85 Millionen Teilnehmer bereitgestellt werden (Vgl. [3], S. 86).

Mit dem Wechsel zu zellulären Netzen stieg jedoch gleichzeitig der technische Aufwand zur Umsetzung und Verwaltung. Durch die geringere räumliche Ausbreitung der Zellen mußten insbesondere Möglichkeiten geschaffen werden, Nutzer von einer Zelle in eine benachbarte zu transferieren, ohne daß der Nutzer dies bemerkt. Im Gegenzug ist es notwendig, die Position eines Endgerätes im Netzwerk zu kennen, so daß dieses von anderen gerufen werden kann.

Diese Anforderungen führten zur Entwicklung und zum Aufbau eines mehrschichtigen Systems, dessen Grundstruktur unabhängig vom jeweiligen Standard auch heute noch besteht (Vgl.[57],

S. 25 ff). Die drahtlose Kommunikation mit den Endgeräten[12] wird vom Base Station System (BSS) ermöglicht. Dieses besteht aus einzelnen Basisstationen, die jeweils als zentrale Empfänger und Sender in einer Zelle des Mobilfunknetzes agieren. Als eine Art „Rückgrat“[13] sind mehrere Basisstationen untereinander über Mobile Switching Center (MSC) verbunden. Diese MSCs dienen als Gateways in andere Netzwerke (Festnetz, Netze weiterer Provider) und koordinieren das Weiterreichen[14] von Endgeräten zwischen verschieden Basisstationen. Außerdem werden Datenbanken verwaltet, die Informationen über die derzeitige Basisstation enthält, über die jedes Endgerät erreichbar ist. Die MSCs sind wiederum über ein Betriebs- und Wartungsnetzwerk[15] (OSS) untereinander verbunden. Dieses erlaubt den Informationsaustausch zwischen den MSCs, bspw. um ein Handover über die Grenze eines MSCs zu bewerkstelligen, und erlaubt eine zentralisierte Wartung und Betriebsführung des gesamten Netzes. Die Infrastruktur zur Kommunikation zwischen den einzelnen Komponenten ist in aller Regel dediziert aufgebaut.

Bereits durch die Einführung dieses hierarchischen Systems entstanden komplexe, heterogene und verteilte Architekturen, deren Entwicklung und Test schwierig sind. Die Einführung weiterer Dienste und Standards führte jedoch zu noch wesentlich komplexeren Architekturen, die im Entwurf und Betrieb immer schwieriger zu handhaben sind.

Mitte der 1980er Jahre erreichte der analoge Mobilfunk seine Grenzen. Um mehr Nutzer zu integrieren und die Sprachqualität zu erhöhen, wurden digitale Mobilfunkstandards entwickelt. In Europa setzte sich der Standard Global System for Mobile Communication (GSM) als entsprechender Standard der 2. Generation durch (Vgl. [57], S. 31). Im Gegensatz zu den analogen Systemen der 1. Generation werden die Audioströme hier digitalisiert und als Bitstrom übertragen. Da zu diesen Zeiten Datendienste auch im ortsgebundenen Umfeld noch nicht weit verbreitet waren, standen jedoch auch bei diesen ersten Digitalstandards die leitungsvermittelten Sprachdienste im Vordergrund der Entwicklung (Vgl. [57], S. 31). Der Short Message Service (SMS) führte jedoch zu ersten Erweiterungen, die auch im Betriebsnetzwerk durch entsprechende Infrastruktur realisiert werden mußten. Die internationale Verbreitung des GSM-Standards erlaubte außerdem eine anbieter- und nationenübergreifende Nutzung entsprechender Mobilfunknetze. Zu Betriebs- und Abrechnungszwecken mußte hierfür die Kopplung der Netzwerke und Datenbanken der einzelnen Anbieter ermöglicht werden.

Die Nutzung von Datendiensten war mit der kanalvermittelnden GSM-Technik aufwändig und belegte viele Ressourcen in der Kommunikation zwischen Endgerät und Basisstation. Mit der Einführung des General Packet Radio Service (GPRS) wurde die paketorientierte Übertragung in den 1990er Jahren auch in die bestehenden GSM-Systeme integriert. Dieses nutzt freie Zeitschlitze nicht genutzter Kanäle („Anrufe“) zum Pakettransfer zwischen Endgerät und BSS (Vgl. [57], S. 46). Im Gegensatz zur GSM-basierten Datenübertragung, wo bei Bedarf erst ein Übertragungskanal geöffnet und verifiziert sowie im Anschluß offen gehalten werden muß, erlaubt GPRS eine ständige Verbindung ins Datennetz („always on“). Dies ist durch die Nutzung

---

[12] Mobilstationen, Vgl. [57]

[13] Englisch: Backbone

[14] Englisch: Handover

[15] Englisch: Operation and Service System

von Signalisierungskanälen möglich, so daß für nicht genutzte Datenverbindungen auch kaum Medienzugriffe notwendig sind.

Zur Umsetzung von GPRS genügte es jedoch nicht, die Funktschnittstelle zwischen Endgeräten und BSS anzupassen. Vielmehr mußte ein zweites Netzwerk aufgebaut werden, das den Basisstationen entsprechenden Zugriff auf Datennetze, wie das IP-basierte Internet, bietet. Dieses Netz arbeitet parallel und weitestgehend unabhängig vom eigentlichen GSM-Backbone. Nur im BSS und im Medienzugriff zu den Endgeräten kollidieren beide Systeme (Vgl. [57], S. 47).

GPRS erlaubte erstmals verschiedene Datenapplikationen, die sich im Desktop-Heimbereich durch die aufkommenden Internet-Breitbandanschlüsse schnell verbreiteten, auch auf Mobilgeräten zu nutzen. Allerdings verhinderten technische Einschränkungen des Standards und der verfügbaren Endgeräte sowie kundenunfreundliche Abrechnungsmodelle den großen Durchbruch von GPRS im Endkundenbereich. Insbesondere die geringen Datenraten von theoretisch 171,2 kBit/s und meist praktisch verfügbaren Maximalraten von 53,6 kBit/s sowie Einschränkungen in der Parallelnutzung von kanalvermittelnden („Anrufe“) und paktetvermittelnden Diensten (Vgl. [57], S. 46ff) machten Überlegungen für eine Folgetechnik notwendig. Verbesserungen waren jedoch nur durch eine grundlegende Änderung der Mediennutzung, die durch die Basistechnologie GSM noch auf Kanalvermittlung ausgelegt ist, und der bestehenden Infrastruktur möglich.

Bereits Anfang der 1990er Jahre wurden weltweit verschiedene lokale Vorschläge für ein neues Mobilfunksystem zusammengetragen. Das Ziel, ein weltweit einheitliches System für den Mobilfunk der 3. Generation zu etablieren, scheiterte schon im Ansatz auf Grund des Fehlens weltweit einheitlicher, verfügbarer Frequenzspektren. Unter einer Menge von Standards, die als IMT-2000[16] von der International Communication Union (ITU) gesammelt wurden, befand sich für den Europäischen Raum und Teile Asiens das vom European Telecommunication Standards Institute (ETSI) spezifizierte Universal Mobile Telecommunications System (UMTS) (Vgl. [11], S. 23f). Es sollte als Standard das GSM-System erweitern und ablösen.

2.4.2. Aktuelle Mobilfunkgeneration

Die in Europa verbreiteten Infrastrukturen zu Beginn des 21. Jahrhunderts bestehen in aller Regel aus Mischsystemen zweier Standards. Der bekanntere ist der Mobilfunk der 3. Generation, der hier durch den Universal Mobile Telecommunications System (UMTS) Standard realisiert wird. Er stellt gleichberechtigte Dienste zur Sprach- und Datenkommunikation bereit. Da mit dem Aufbau von UMTS erst zu Beginn der 2000er Jahre begonnen wurde, konnte eine flächendeckende Einführung nicht bewerkstelligt werden. Vielmehr wurde das neue System mit dem bereits etablierten und flächendeckend ausgebauten Global System for Mobile Communication (GSM) kombiniert. GSM ist die europäische Umsetzung der 2. Mobilfunkgeneration, bei der qualitativ hochwertige, drahtlose Sprachkommunikation im Vordergrund steht. In einer Übergangsphase zu UMTS wurden verschiedene Standards, wie GPRS oder EDGE, ergänzt, die auch paketorientierte Datendienste erlauben.

---

[16] International Mobile Communication - Standards die voraussichtlich ab dem Jahr 2000 zum Einsatz kommen

Beide Systeme bilden zelluläre Netze, in denen die zu versorgende Fläche in Abhängigkeit von Nutzerzahlen und geographischen Bedingungen in kleine Zellen aufgesplittet wird. Jede Zelle wird von einer eigenen Basisstation versorgt. GSM und UMTS nutzen verschiedene Frequenzbänder und Medienzugriffsverfahren, um die Kommunikation zwischen mobilen Endgeräten und der Basisstation zu realisieren. Demzufolge gibt es beim Parallelbetrieb der Netze keine bzw. kaum Beeinflussungen beim Zugriff auf das Medium Luft.

Die Basisstationen müssen an ein Infrastruktursystem angeschlossen werden, um zum einen die Nutzerdienste bereitstellen zu können und zum anderen auch basisstationsübergreifende Betriebs- und Wartungsoperationen zu ermöglichen. Eine der wichtigsten Funktionen ist das „Handover“, das ein Endgerät je nach Empfangsstärke von einer Basisstation an eine benachbarte weiterreicht. Auch der Wechsel von einer UMTS-Verbindung in das GSM-Netz wird unterstützt. Diese Wechsel erfolgen für den Nutzer unbemerkt und werden auch bei aktiver Dienstnutzung, bspw. einem Anruf, ermöglicht.

Zur Umsetzung dieser Betriebsoperation und insbesondere der Bereitstellung der Datendienste erfolgt die Vernetzung der Basisstationen meist über ein dediziertes Netzwerk auf Basis des Asynchonous Transfer Mode (ATM). Kupfer- und Glasfaserkabel kommen ebenso als Medium zum Einsatz wie Richtfunksysteme. Aus Kostengründen arbeiten einige Netzanbieter derzeit an der Umstellung auf IP-basierte Technologien ((Vgl. [37]).

An das Netzwerk der Basisstationen sind verschiedene, hierarchische Systeme angeschlossen, die von einzelnen Basisstationen bzw. Endgeräten angeforderten Dienste vermitteln oder bereitstellen. Hierbei wird auch bei UMTS noch zwischen leitungsvermittelnden und paketvermittelnden Diensten unterschieden. Hinzu kommen Server- und Datenbanksysteme, die zur Verwaltung der Endgeräte, bspw. das Auffinden aber auch das Führen von Abrechnungsdaten, benötigt werden. Die Server- und Vermittlungssysteme sowie das Netzwerk werden häufig als „Backbone“ bezeichnet.

Kostentreiber bei Mobilfunknetzen sind die Immobilien zur Installation der Basisstationen und das dedizierte Netzwerk zur Vernetzung der Basisstationen und des Backbones. Um die Wirtschaftlichkeit zu gewährleisten, ist bereits bei der Spezifikation von UMTS verankert worden, daß einige Teile des bereits existierenden GSM-Systems verwendet werden sollen. Die Server- und Vermittlungssysteme wurden den gestiegenen Anforderungen, bspw. an Datenraten und -volumen, angepaßt und sind durch Parallelsysteme vom eigentlichen GSM getrennt. Das dedizierte Netzwerk zur Verbindung der Komponenten im Backbone von GSM und UMTS wird hingegen gemeinsam genutzt. Ein Großteil der Endgeräte implementiert den Zugriff auf beide Netze, so daß sich je nach Verfügbarkeit in ein geeignetes Netz eingewählt werden kann. Ein Netzwechsel ist auch während der Nutzung verschiedener Dienste, bspw. einem laufenden Anruf, möglich. Hierfür müssen Handover-Mechanismen vorgesehen werden, die standardübergreifend arbeiten. Entsprechend gibt es auch operative Verflechtungen zwischen den eigentlich unabhängigen UMTS- und GSM-Netzen.

2.4.3. Zelluläre Netze als SPS

Im Abschnitt 2.3.2 wurde die Klasse der Dienstbereitstellenden Systeme eingeführt und erläutert. Wird ein aktuelles UMTS/GSM-Netz als ein solches System betrachtet, sind zunächst die Dienste zu definieren. Wie bereits erwähnt, sind diese abhängig von der Sicht auf das System. Die offensichtlichste und im Rahmen dieser Arbeit bevorzugte Sicht ist die des Nutzers, der mittels eines Endgerätes (Mobiltelefon, Laptop, ...) auf die Dienste des Mobilfunknetzes zugreift.

Welche Dienste genau angeboten bzw. genutzt werden, hängt von der jeweiligen Ausbausstufe des Netzes und den technischen Möglichkeiten der Endgeräte ab. Typische Dienste sind:

- **Sprachanruf**
  Audioübertragung zwischen Endgerät und zweitem Telefon im selben Mobilfunknetz oder in anderen Telefonnetzdomänen
- **Videoanruf**
  wie Sprachanruf, jedoch Audio- und Videoübertragung
- **Short Message Service** (SMS)
  Übertragung kurzer Textnachrichten vom Endgerät
- **Multimedia Message Service** (MMS)
  Übertragung von Nachrichten mit multimedialem Inhalt
- **Paketorientierte Datendienste**
  Übertragung von Datenpaketen auf Basis verschiedener Protokolle

Diese Einteilung der Dienste entspricht zum einen der natürlichen Wahrnehmung des Mobilfunksystems durch den Nutzer, spiegelt allerdings auch in Teilen strukturelle Eigenschaften des UMTS/GSM-Netzes wieder. Zur Bereitstellung werden Strukturen in Form von Servern, Gateways oder speziellen Netzwerken vorgehalten. Bspw. werden Sprach- und Videoanrufe durch ein kanalvermittelndes Teilsystem realisiert, während die Datendienste paketvermittelt bereitgestellt werden. Dies hat sowohl Auswirkungen auf den Medienzugriffsmechanismus bei der Kommunikation zwischen Endgerät und Basisstation als auch bei der Ressourcenverwendung im Backbone.

Allerdings ist diese Aufschlüsselung in Dienste nicht eindeutig. Es ist bspw. ebenso denkbar, die Datendienste in weitere, protokollspezifische Dienste, wie z. Bsp. HTTP, SIP, ..., aufzuteilen.

Mit der Abstraktion vom UMTS/GSM-Netz hin zu abstrakten Diensten ist aus Sicht des Dienstnutzers oder des Endgerätes die zugrundeliegende Architektur des Gesamtsystems unbedeutend. Ein Aufruf einer entsprechenden Applikation durch den Nutzer oder ein Ruf einer Systembibliothek auf dem Endgerät selbst startet eine asynchrone Anfrage an den entsprechenden Dienst, die erfolgreich oder nicht erfolgreich nach einer bestimmten Zeit beendet wird. Das Beenden erfolgt entweder durch die Abarbeitung eines Dienstaufrufes, bspw. nach erfolgtem

Senden aller Zeichen einer Textnachricht, durch Nutzereingriff, bspw. das Beenden eines Telefongesprächs, oder durch einen Fehler, bspw. einen Verbindungsverlust.

Formal muß der Nutzung eines Dienstes i durch einen Klienten s - in diesem Falle ein mobiles Endgerät - eine entsprechende Last ${}^{t}l_{s,i}$ zugeordnet werden. Diese muß in logischer Relation zur Nutzung von Ressourcen stehen und wird systemweit zu einer Gesamtlast des SPS summiert.

Im Falle der leitungsvermittelnden Anrufdienste ist die Zuordnung einer Last recht einfach und offensichtlich. Jedem laufenden Anruf wird eine Last von 1 zugeordnet. Demzufolge ergibt sich die Gesamtlast für die Dienste Sprachanruf und Videoanruf, die von einem Klienten s erzeugt werden, aus der Summe der derzeit laufenden Anrufe. Ein Großteil der verfügbaren Endgeräte wird technisch und ergonomisch nur in der Lage sein, zum Zeitpunkt t höchstens einen Sprachanruf oder höchstens einen Videoanruf auszuführen. Üblicherweise schließt auch ein Sprachanruf einen Videoanruf (und umgekehrt) aus, so daß auch Abhängigkeiten zwischen den Lasten beider Dienste bestehen. Für die Betrachtung der Last ist es unerheblich, ob ein Anruf vom Klienten initiiert oder angenommen wurde.

Auch eine Lastquantifizierung für die Datendienste ist naheliegend. Hier bestimmen die transportierten Datenmengen pro Zeiteinheit die Last für das SPS. Demzufolge wird der Last eines Klienten die Summe der Datenraten für Download und Upload zugeordnet. Die verwendeten Einheiten, wie Byte/s oder kByte/s, sind dabei interpretativ und müssen derart gewählt werden, daß mit natürlichzahligen Lasten aussagekräftige Beschreibungen möglich sind.

Schwieriger ist die Lastdefinition für Nachrichtendienste. MMS-Nachrichten sind durch die multimedialen Inhalte relativ groß und bedingen demzufolge ein hohes Datenaufkommen zum Versenden bzw. Empfangen. Demzufolge ist auch hier eine Lastdefinition auf Basis der Datenrate sinnvoll. Da viele Mobilfunkanbieter verschiedene Server für Datendienste und MMS-Nachrichten verwenden, ist eine Trennung beider Dienste bei der Betrachtung als SPS jedoch durchaus sinnvoll. SMS-Nachrichten hingegen bestehen ausschließlich aus Zeichen. Demzufolge beschränkt sich die Größe auf wenige hundert Byte. Da derartige Datenmengen üblicherweise in Bruchteilen von Sekunden übertragen werden können, belasten die Nachrichten die Ressourcen des Mobilgerätes und der Basisstation kaum. Allerdings werden alle SMS-Nachrichten eines Mobilfunknetzes an eine Nachrichtenzentrale gesendet. Bei einigen Millionen Nutzern kann die Anzahl der Nachrichten an dieser Zentrale durchaus zu großen Lasten führen. Im Jahr 2010 sind in deutschen Mobilfunknetzen ca. 41,3 Milliarden SMS versendet worden (Vgl. [3], S. 88). Dies entspricht rund 1300 Kurznachrichten, die durchschnittlich pro Sekunde (und verteilt auf verschiedene Anbieter) bearbeitet werden mußten. Zu Spitzenzeiten, bspw. am Neujahrsmorgen, dürfte es zu einem Vielfachen dieses Wertes gekommen sein.

Die Last, wie sie im Abschnitt 2.3.2 eingeführt wurde, ist aber aus Sicht der Klienten definiert. Eine Einheit von mehreren Nachrichten pro Sekunde erscheint ergonomisch unrealistisch. Um die Last so zu definieren, daß sie in der Summe auch zu den o.g. Lastsummen führt und bezogen auf einen Klienten ganzzahlig sinnvoll ausgedrückt werden kann, ist eine Erweiterung des betrachteten Zeitraums notwendig. So kann bspw. die SMS-Last in Nachrichten pro Stunde angegeben werden. Es ist jedoch zu beachten, daß durch die kurze Übertragungzeit einer einzelnen Nachricht mit der

Angabe der Last implizit eine Wahrscheinlichkeitsverteilung für Einzelereignisse (eine SMS) über die betrachtete Dauer (Stunde) eingeführt wird.

Werden Erfolg bzw. Mißerfolg sowie die Lasten aus Nutzersicht interpretiert, darf der Einfluß des Endgerätes nicht vernachläßigt werden. Nur durch die Nutzung eines Mobilgerätes ist überhaupt der Zugriff auf die Dienste des Mobilfunknetzes möglich. Konkret greift der Nutzer auf Dienste des Endgerätes zu, welches wiederum Dienste des Mobilfunknetzes nutzt. Diese Endgerätdienste unterliegen wiederum Einschränkungen der verwendeten Technologien und der verfügbaren Ressourcen. Entsprechend müssen maximale Lastwerte, die ein bestimmtes Endgerät generieren kann, so gewählt werden, daß die Einschränkungen des Endgerätes das Ergebnis der Dienstnutzung nicht zu stark beeinflussen oder verzerren. Auch Fehler in der Umsetzung der Endgeräte können nicht ausgeschlossen werden und müssen bei der Interpretation der tatsächlich ausgeführten Dienstaufrufe berücksichtigt werden.

2.4.4. Herausforderung Systemtest

Moderne Mobilfunknetze sind, wie oben beschrieben, verteilte Systeme, die aus heterogenen Einzelkomponenten bestehen und über verschiedene Netzwerktechnologien miteinander verbunden sind. Im Falle der in Europa verbreiteten UMTS/GSM-Netze handelt es sich sogar um zwei Systeme unterschiedlicher Standards, die an einigen Stellen gemeinsame Komponenten nutzen, Ressourcen teilen oder Schnittstellen zum operativen Austausch von Informationen vorsehen.

Auf Grund der Konzeption als zelluläres Netz müssen einige Komponenten des Netzes, bspw. die Basisstationen, nur auf die Handhabung einiger Dutzend bis einiger Hundert Klienten ausgelegt werden. Diese Szenarien lassen sich zum Zwecke des Systemtests im Labor oder in Feldtests nachstellen. Andere zentrale Komponenten hingegen verwalten die Klienten vieler Basisstationen und müssen somit weitaus mehr handhaben können. Desweiteren stellen zwar verschiedene Komponenten des Mobilfunknetzes verschiedene Dienste bereit, nutzen dafür aber meist gemeinsame physische oder logische Ressourcen. Dadurch kann die intensive Nutzung eines Dienstes auch zu Wechselwirkungen mit anderen, eigentlich unabhängigen Diensten führen. Die Komplexität dieser Sachverhalte stellt immense Herausforderungen für die Ausführung eines Systemtests, denn erst auf dieser Ebene können derartige Abhängigkeiten überprüft werden.

Im Labor werden auf kleinem Raum sehr viele Klienten zum Zugriff auf das Mobilfunknetzwerk - meist ein bis zwei Basisstationen - genutzt. Die Klienten können über ein spezielles System ferngesteuert und auf diese Weise die Lasterzeugung zentral gesteuert werden. Damit lassen sich hauptsächlich die Kommunikation zwischen Endgeräten und Basisstation sowie bestimmte Netzaktionen, wie Handover, automatisiert testen. Allerdings führen die verwendeten Testskripte meist zu synthetischen, regelmäßigen Testmustern, die nur bedingt reales Nutzerverhalten wiedergeben. Desweiteren werden gewisse dynamische Systemparameter, wie die Bewegung von Endgeräten, ausgeblendet.

Als Ergänzung zum Labortest kommen deshalb Feldtests zum Einsatz. Hier werden komplexere Netze samt Backbone aufgebaut und einer eingegrenzten Nutzergruppe Endgeräte zur Nutzung zur Verfügung gestellt. Meist dienen Firmenzentralen oder Universitätscampus als Umfeld. Die Nutzer werden je nach Testfall aufgefordert, die Endgeräte und das Mobilfunknetz nach Vorgaben zu nutzen und damit Last im Netz zu erzeugen. Zwangsläufig ergeben sich daraus unregelmäßigere und dynamischere Tests. Testingenieure können allerdings die Menge der Klienten nicht mehr fernsteuern, und damit bestimmte Testmuster nicht bewußt herbeiführen. Das Verhalten der Testnutzer spielt also eine entscheidende Rolle für die Qualität des Tests.

In der Praxis setzt man üblicherweise auf kombinierte Labor- und Feldtests.

Wie wichtig es aber sein kann, auch nicht alltägliches Nutzerverhalten zu simulieren, zeigt sich an vielen Beispielen beim Einsatz der Mobilfunknetzwerke. Beispielsweise führte von 2008 bis 2010 die Einführung preiswerter Tarifstrukturen und die Verfügbarkeit ansprechender Smartphones mit zugehörigen Anwendungen zu einem exponentiellen Anstieg des Datenvolumens in deutschen Mobilfunknetzen (Vgl. [3], S. 90). Systeme und Ressourcen müssen bereits mit großem Vorlauf für derartige Anforderungen vorbereitet werden, was wiederum adäquate Tests des zukünftig erwarteten Nutzerverhaltens notwendig macht. Gelingt dies nicht, ist die Netzintegrität in Gefahr. So brachte in Japan eine einzige Anwendung für das weitverbreitete Betriebssystem Android ein ganzes Betreibernetz zum Zusammenbruch (Vgl. [73]).

Natürlich kann ein automatisiertes System zur Unterstützung von Tests derartige Entwicklungen nicht voraussehen - aber es kann helfen, derartige Szenarien, die nicht von aktuellem Nutzerverhalten provoziert werden, zu überprüfen. An diesem Punkt setzt die vorliegende Arbeit an.

2.5. Zusammenfassung

In diesem Kapitel wurden die Begrifflichkeiten Testen, Verifikation und Validierung abgegrenzt sowie eine grobe Kategorisierung der Tests vorgenommen. Die für diese Arbeit maßgebliche Klasse der Dienstbereitstellenden Systeme (SPS) wurde eingeführt. Es handelt sich im Groben um Systeme, deren funktionales Verhalten sich in Form von Diensten abgrenzen läßt. Die Dienste werden logischen oder physischen Klienten bereitgestellt. Die Nutzung eines Dienstes durch einen Klienten ist als Last quantifizierbar.

Beispiele eines SPS sind Mobilfunknetzwerke, deren Entwicklung ebenso im zurückliegenden Kapitel beschrieben wurde, wie die Interpretation als SPS sowie die Herausforderungen, die sich durch einen Systemtest ergeben. Im Laufe der Arbeit dient die Validierung eines Mobilfunknetzwerkes als anschauliches Beispiel für die allgemeinen Konzepte.

3. Stand der Technik

Verifikation und Validierung von Systemen, vor allem von zustandsdiskreten, beschäftigt Forschung und Entwicklung seit den Anfängen der programmgesteuerten Rechner. Entsprechend wurden vielfältige Konzepte und Technologien für verschiedene Anwendungsfelder, Entwurfsebenen und Entwurfstechnologien entwickelt. Dieses Kapitel wird einen Überblick über praktische und theoretische Ansätze für Verifikation und Validierung heterogener Systeme geben.

Aus dem Konzept dieser Arbeit ergibt sich das algorithmische Problem *„Testpartitionierung“*, welches auf einigen wohldokumentierten Standardproblemen basiert. Zum Verständnis werden in einem Abschnitt dieses Kapitels deshalb einige Algorithmen und deren Lösungsansätze vorgestellt.

3.1. Automatisiertes Testen

Das vollständige Testen durch Abprüfen aller denkbaren Eingaben und aller möglichen Zustände ist bekanntlich aufgrund der Anzahl der resultierenden Testfälle selbst bei einfachen Systemen kaum möglich. Es existieren im Grunde zwei Möglichkeiten, diesem Problem zu begegnen. Üblicherweise müssen beide Verfahren angewendet werden.

Zum ersten kann durch eine gezielte Auswahl erfolgsversprechender Testfälle[1] die Menge der auszuführenden Tests stark eingeschränkt werden. Dieser Vorgang bedarf meist abstraktem oder kreativem Denken und ist stark vom zu testenden System abhängig (nach [32], S. 17f; [7]). Eine Automatisierung dieser Vorgänge ist kaum möglich.

Die zweite Möglichkeit besteht darin, die Anzahl der pro Zeit (bzw. pro Kosten) definierten und ausgeführten Testfälle zu erhöhen. Dies wird meist durch den Einsatz von Automatisierungsprogrammen erreicht. Diese können zum einem bei der Generierung von Testfällen, zum anderen bei der Ausführung von Tests sowie der Verifikation verwendet werden (Vgl. [32] S. 19ff.; [40] S. 332ff.). In der Praxis kommt die letztgenannte Variante deutlich häufiger vor (Vgl. [32], S. 25).

Es existieren unzählige Testautomatisierungsansätze und Programme. Jedes ist dabei auf eine bestimmte Testebene hin konzipiert. Häufig kommen auch Anpassungen an technologische Rahmenparameter, bspw. das verwendete Betriebssystem und die Programmiersprache, oder auch den Anwendungskontext selbst hinzu.

Die naheliegendste Möglichkeit der automatisierten Testausführung sind Testroboter (Vgl. [81], S. 210ff). Diese häufig auch als „Capture & Replay“ -Tools bezeichneten Programme erlauben das Aufzeichnen und wiederholte Ausführen von Aktivitäten manueller Tester (Vgl. [32], S.

---

[1] Es gilt zu beachten, daß „erfolgsversprechend“ im Falle des Testens dem Auffinden von Fehlern im zu testenden System dient.

34f). Diese Aktivitäten sind häufig Eingaben per Tastatur, Maus oder anderer Eingabegeräte. Die Ausgaben des zu testenden Systems werden ermittelt und mit den Soll-Werten verglichen.

Es existieren zahlreiche kommerzielle und nicht-kommerzielle Testrobotersysteme mit verschiedenen Schwerpunkten und Leistungsumfängen (Vgl. [34], S. 213ff). Bekannt sind bspw. HP QuickTest Professional (vormals von Mercury Interactive), JStudio SiteWalker Professional oder auch das unter Apache-Lizenz verfügbare Selenium.

Gemein ist den meisten Ansätzen, daß der Aufwand zur Erstellung und Pflege von Testfällen recht hoch ist. Insbesondere die Ermittlung der Ausgaben ist technisch schwierig, besonders bei aufwändigen graphischen Oberflächen. Änderungen in der Oberfläche, und sei es nur die Sprache, haben großen Aufwand zur Folge. Auch kann im Falle der Wiedergabe der Testfälle nicht wirklich von einer Automatisierung gesprochen werden - selbst kleinere Fehler, wie bspw. ein sich unerwartet öffnendes Fenster einer graphischen Oberfläche - können zum Testabbruch bzw. unkontrollierten Verhalten führen (Vgl. [32], S. 62ff). Zustands- oder sitzungsbasierte Systeme sind schwer zu testen.

Demzufolge sind alternative Verfahren zur Testautomatisierung wünschenswert, die mehr bieten als nur die Wiedergabe vorab aufgezeichneter oder geskripteter Nutzereingaben. Im allgemeinen setzt sich jeder Testprozeß aus einer Sequenz von Aktivitäten zusammen (nach [32], S. 13ff; [89], S. 17):

- Identifizierung und Priorisierung der Testparameter
- Auswahl und Entwurf der Testfälle
- Implementierung der Testumgebung und der Testfälle
- Ausführung jedes Testfalls
  - Eingabedaten bereitstellen
  - Test ausführen
  - Ergebnisse ermitteln, vergleichen und speichern
  - Optional: Zurücksetzen des zustandsbasierten Systems
- Ergebnisse analysieren

Im Gegensatz zur weit verbreiteten Meinung führt die Testautomatisierung jedoch nicht zwangsläufig zu schnelleren (und billigeren) Testphasen im Entwurfsprozeß (Vgl. [40], S. 332). In einer Studie zur Testautomatisierung der graphischen Oberfläche eines GSM-Basisstationskonfigurationsprogramms wurde ermittelt, daß die Vorbereitung des automatischen Tests durchschnittlich doppelt so lange dauert wie die Vorbereitung des manuellen Tests (Vgl. [29], S. 52ff). Dem gegenüber ist die Ausführung eines einzelnen Testfalls - wie zu erwarten - im automatisierten Falle wesentlich schneller.

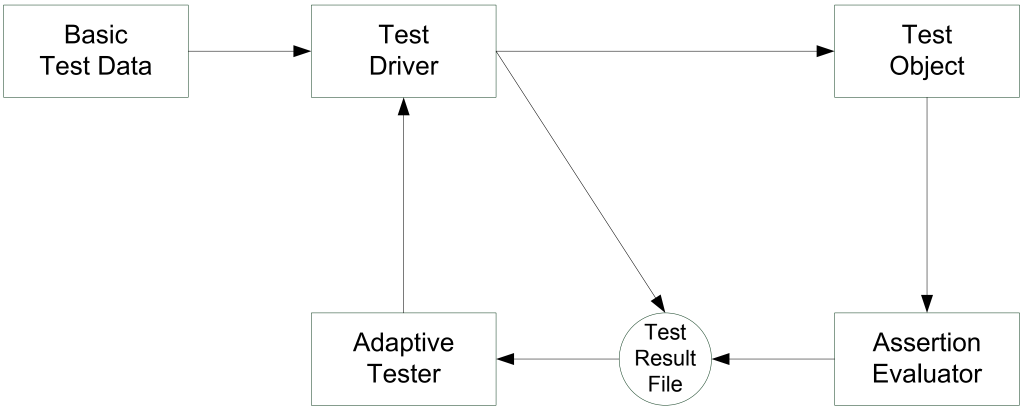


Abbildung 3.1.: Architektur eines einfachen Testausführungssystems, Vgl. [7]

Dementsprechend ist die Automatisierung von Testvorgängen besonders für folgende Konstellationen geeignet bzw. wünschenswert (nach [32] S. 9f):

- Häufige Wiederholung gleicher Tests auf verschiedene Systemversionen (Regressionstests)
- Notwendigkeit hoher Testpräzision (Wiederholung, Zeitverhalten, ...)
- Manuell nicht beherrschbare Tests (bspw. große Server-Klienten-Strukturen)

Es gibt zahlreiche Softwaresysteme, die einzelne Aktivitäten unterstützen bzw. automatisieren. Neben der reinen Verwaltung der meist recht umfänglichen Testfälle liegt die Hauptschwierigkeit in der Definition des Soll-Verhaltens von zu testenden Systemen sowie ein automatisiertes Ermitteln des Ist-Verhaltens und Vergleichen mit dem Soll.

Ein früher Ansatz zur Handhabung dieser Probleme und automatischen Ausführung von Testfällen mit dem Ziel der funktionalen Korrektheitsprüfung wird in [7] vorgestellt. Die Idee besteht darin, das Soll-Verhalten nicht in Form von erwarteten Ausgaben eines Systems zu spezifizieren sondern direkt im Programmcode des zu testenden Systems die Generierung normierter Debugging-Nachrichten zu integrieren. Diese „Executable Assertions“ genannten Konstrukte, die nur unter bestimmten zu prüfenden Bedingungen ausgeben werden, soll auf abnorme Systemzustände hindeuten. Bei der Ausführung von Testfällen, die nur noch aus einer Menge von Eingaben besteht, werden auftretende Assertions gespeichert.

Ebenso wird in [7] eine allgemeine Architektur des Testausführungssystems beschrieben, wie es in vielen Adaptionen heute verwendet wird. Auch das Konzept dieser Arbeit basiert auf den Grundzügen des Systems, das in Abbildung 3.1 dargestellt ist. Es wird ein Testtreiber verwendet, der ein zu testendes System mit den Eingaben beaufschlagt und die Ausgaben - in dem Konzept die Assertions - aufnimmt und speichert. Im Falle der Arbeit von [7] wird die Generierung der Testfälle von der Zahl und der Art der Assertions beeinflußt.

Basierend auf dieser Idee wurden zahlreiche weitere Konzepte und Programme entwickelt. Assertions wurden in modernen Programmiersprachen in das Konzept der Exceptions übernommen und dienen dort der Fehlerfallbehandlung (Vgl. [26]). Das Testausführungssystem wurde in weitere Testframeworks integriert. Bekannte Beispiele sind das „IBM Software Testing Automation Framework“ ([80]; [69]) oder „DejaGNU“ von der Free Software Foundation[2]. Das „Test Environment Toolkit“[3] von „The Open Group“ ist ein ähnliches System, das auch die Testfallgenerierung durch verteilte Klienten unterstützt (Vgl. [89], S. 21f).

Mit komplexer werdenden Systemen steigen auch die Anforderungen an die automatisierten Testansätze. Eine besondere Herausforderung stellt die Vernetzung, Verteilung, Parallelisierung und Virtualisierung von Rechnersystemen dar. Daraus resultierende dynamische Laufzeiteffekte erschweren das Testen, da statische Testfälle nur noch unzureichende Zustandsräume abdecken. Auch die Testautomatisierung wird unter Umständen durch die Architektur der Systeme erschwert, da ein zentraler Zugriff schwierig möglich ist bzw. nur Teile des zu testenden Systems abdeckt.

Entsprechend beschäftigte sich die Forschung auch mit Ansätzen, derartige dynamische Systeme automatisiert zu testen. Da sich die Charakteristik der Systeme unterscheidet, sind auch jeweils Ansätze für die entsprechende Architektur notwendig. Bspw. wird in [28] über einen praxiserprobten Ansatz zum Test dienstorientierter Architekturen (Vgl. Abschnitt 2.3.3) berichtet. Der Ansatz beruht in erster Linie auf der Aufteilung des Testtreibers (Vgl. Abbildung 3.1) in einen Master-Agent und eine Menge von Test-Agents, die dann die Log-Files der jeweiligen Dienstinstanzen auswerten. Im Zentrum der Analysen stehen vor allem funktionale Eigenschaften, wie die korrekte Sequentialisierung von Transaktionen. Einem ähnlichen Problem, nämlich der Priorisierung von Testfällen für Regressionstest mit dem Ziel, Fehler auf verschiedenen Ebenen des Entwurfsprozesses zu finden, widmet sich [61].

Allerdings zielt bei dynamischen und verteilten Systemen ein automatisierter Test auf Systemebene häufig nicht mehr in erster Linie auf funktionale Korrektheit ab (nach [58]). Es stehen nichtfunktionale Fragen, wie bspw. das Verhalten von Leistungsparametern und Ressourcenbedarf unter bestimmten Belastungssituationen, im Vordergrund.

3.2. Leistungstests

Leistungstests werden üblicherweise als nichtfunktionale Tests klassifiziert (Vgl. [40], S. 327f) und dienen dazu, Leistungsfähigkeit, Skalierbarkeit und Ressourcenanforderungen des zu testenden Systems zu überprüfen. Häufig wird ermittelt, ob vorab definierte Anforderungen oder Ziele eingehalten werden. Dazu werden Leistungstests üblicherweise in Last- und Streßtests unterschieden. Beim Lasttest wird das System mit Lasten beaufschlagt, die Teil der Spezifikation oder Anforderungen sind, und es wird überprüft, ob das System diese zu jeder Zeit funktional korrekt abarbeitet. Im Falle von Streßtests wird die Belastung weiter erhöht, bis das System unter Umständen ausfällt oder Anforderungen nicht mehr erfüllt (Vgl. [40], S. 330).

---

[2] Informationen: http://www.gnu.org/software/dejagnu

[3] Informationen: http://tetworks.opengroup.org

Bei dieser Art von Tests wird die manuelle Durchführung weitaus schwieriger. Das einfachste Beispiel sind Leistungstests bei Servern. Hier müssen zahlreiche Anfragen von Klienten zur gleichen Zeit koordiniert und ausgewertet werden - was manuell fast unmöglich ist. Es ist also notwendig, die Ausführung der Leistungstests (zum Teil) zu automatisieren. Für verschiedene, verbreitete Systemklassen oder Architekturen, bspw. Web Services, Server-Klienten-Architekturen oder Grid-Systeme, existieren entsprechend auch verschiedene Ansätze.

Ein Ansatz zum Streßtest, der auf den Testroboteransätzen beruht, wird in [53] vorgestellt. Nutzerskripte von Klienten werden jedoch derart aufgearbeitet, daß auch der Umgang mit sitzungsbasierten, also zustandsbehafteten, Systemen möglich ist. Die HTTP-Rufe an das zu testende System werden mittels httperf erzeugt - einen System von HP, zur Messung der Leistungsfähigkeit von Serversystemen[4]. Der Fokus des Ansatzes beschränkt sich auf HTTP-basierte Webanwendungen.

Ein ähnlicher Ansatz, hier jedoch für Systeme, die auf virtuellen Plattformen ausgeführt werden, wird in [36] präsentiert. VATS - Virtualized-Aware Automated Test Service - berücksichtigt schwankende Ressourcen der virtuellen Systeme - allerdings auf Seiten der Testfallausführung. Es erlaubt, in virtuellen Umgebungen Ressourcen für die Ausführung von Tests an einem anderen System zu allokieren sowie zu deallokieren und stellt entsprechende Mechanismen dafür bereit. Auch in VATS wird zur Generierung der Zugriffe ein externes System, HP Load Runner[5], verwendet, so daß auch hier der Fokus auf Webanwendungen liegt. Ziel der vorliegenden Arbeit ist ein System, das dynamisch Tests unter Berücksichtigung existierender Ressourcen auf den Klienten verteilt. VATS arbeitet mit dem inversen Konzept zur Allokation von Ressourcen um Tests ausführen zu können.

Auch für die weitverbreiteten Web-Services existieren zahlreiche Leistungstestansätze. In [58] wird der gesamte Testprozeß analysiert und eine traditionelle Architektur zum automatisierten Leistungstest vorgeschlagen. Ähnlich wird in [75] eine Menge automatisierter, hierarchischer Tests beschrieben, die verschiedene Ebenen und Kompositionen der zu testenden Web-Services testet. Als Sprache zur Modellierung der Tests wird die „Testing and Test Control Notation“ (TTCN-3) (Vgl. [96]; [74]) verwendet. Schwerpunkt des Ansatzes ist die Generierung von Web-Service-Anfragen aus TTCN-3.

Ein überaus interessanter Ansatz zum automatisierten Testen von Web-Services wird auch in [98] vorgestellt. Das Ziel hier ist, jeden Web-Service einzeln und ohne Einfluß durch Netzwerkstrukturen und Transportprotokolle zu testen. Dies ist nach Meinung der Autoren insbesondere für die Bestimmung von Leistungsparametern wichtig. An einem zentralen Testrechner werden hierfür Agenten erzeugt, die eine Menge elementarer Tests „tragen“. Diese werden dann per Socket an den lokalen Rechner, der auch den Web-Service ausführt, übertragen und ausgeführt. Ergebnisse werden durch die Agenten geloggt. Nach der Abarbeitung der Tests „wandert“ der Agent zurück zum Testrechner, wo eine Auswertung erfolgt. Das Konzept wurde in Form einer Testplattform umgesetzt und evaluiert.

---

[4] Informationen: http://www.hpl.hp.com/research/linux/httperf

[5] Informationen: www8.hp.com/us/en/software-solutions/software.html?compURI=1175451

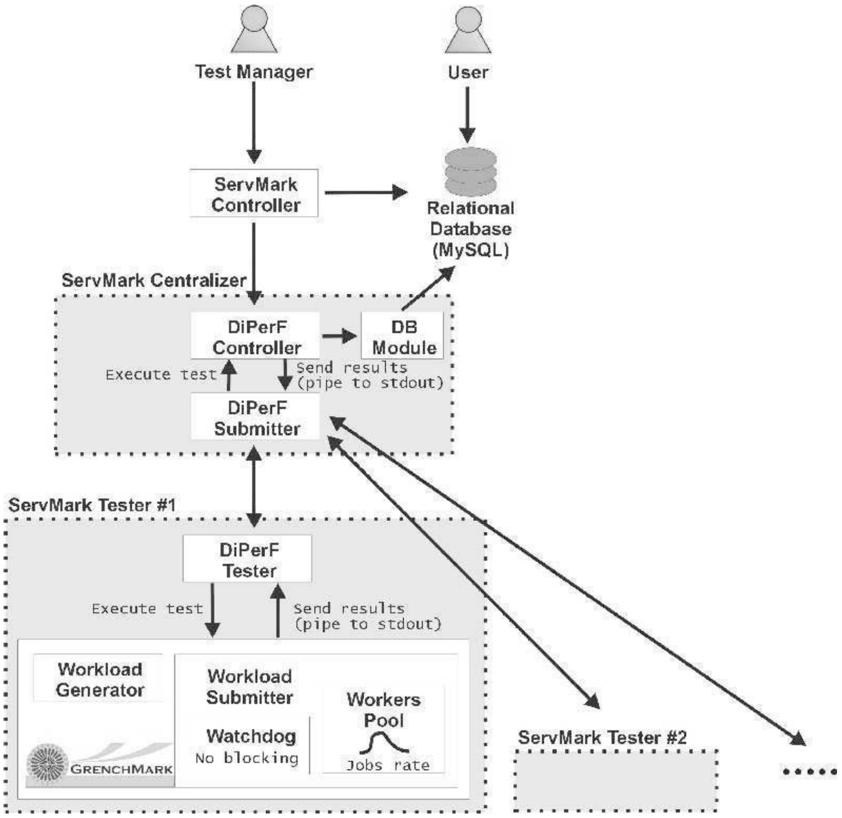


Abbildung 3.2.: Architektur von ServMark zum Test von Grid-Systemen, Quelle [5]

Das „Distributed Performance Testing Framework“ (DiPerF) (Vgl. [27]) und ServMark (Vgl. [27]; [6]) sind Konzepte, die durch das EU-Projekt „European Research Network on Foundations, Software Infrastructures and Applications of large scale distributed, GRID and Peer-to-Peer Technologies“, kurz CoreGrid[6] (Vgl. [25], S. V), entstanden sind. Im Zentrum der Arbeiten steht der Test von Grid-Systemen, also verteilten und vernetzten, heterogenen Rechnersystemen.

Die Architektur von ServMark ist in Abbildung 3.2 dargestellt. Ausgeführt werden die Tests in Form von Lasten für die Dienste des Grids durch Tester, die auf verschiedenen Hosts laufen. Von einem zentralen System aus werden die Testaufträge errechnet und verwaltet. DiPerF dient dazu, Verbindung zwischen dem zentralen System und den verteilten Testern zu halten. Die einzelnen Workload-Generatoren, also die Programme, welche die Zugriffe auf die Grid-Dienste ausführen, werden mittels RPC vom DiPerF-Submitter aus gestartet. Damit einher geht die Forderung, daß auf allen Systemen RPC-kompatible Plattformen verfügbar sein müssen. Die Tester wiederum melden Ergebnisse der Tests an den DiPerF-Submitter, wo Auswertungen generiert und in eine zentrale, externe Datenbank gespeichert werden.

Die vorgestellten Ansätze zum Leistungstest sind zumeist auf spezielle Architekturen zugeschnitten. Insbesondere Netzanwendungen stehen im Vordergrund, so daß auch die zugrundeliegenden Technologien (TCP, HTTP, ...) Anwendung finden. Ein Großteil der Systeme kann Dynamik während der Testausführung, bspw. ausfallende Klienten zum Generieren von Tests,

---

[6] Das Projekt endete 2008 - Information sind noch verfügbar unter: http://coregrid.ercim.eu

nicht handhaben. Auch heterogene Strukturen werden kaum berücksichtigt oder, im Falle von VATS, als rekonfigurierbar (Allokation) angesehen. An diesen Punkten knüpft das Konzept dieser Arbeit an.

3.3. Ansätze jenseits des Softwaretests

Die zurückliegenden Abschnitte beschäftigten sich vorrangig mit dem Testen von softwarelastigen Systemen. Selbstverständlich gibt es auch in anderen Ingenieursdisziplinen Ansätze zur Überprüfung der entwickelten Systeme. Auch gibt es Systeme, die zwar zum Teil programmgesteuert sind aber deren Funktion nur in Kombination mit weiteren Domänen nutzbar ist, für die entsprechende Testansätze entwickelt werden müssen. Ein bekanntes Beispiel hierfür sind Eingebettete Systeme.

Auch bei diesen Systemen spielen die im letzten Abschnitt erwähnten Aktivitäten zum Test eine Rolle. Entsprechend existieren auch Ansätze zur Automatisierung bzw. Formalisierung einzelner Aktivitäten. Die Testausführung ist natürlich stark von der Charakteristik des jeweiligen Systems abhängig, so daß allgemeine Ansätze rar sind. Häufig steht die Generierung von Testfällen im Vordergrund.

In [24] wird beispielsweise beschrieben, wie aus einer Petrinetz-Modellierung (Vgl. [67], S. 49ff) eines verteilten Systems mittels mathematischer Analyse Testfälle abgeleitet werden können, die optimal in einer Bewertungsfunktion sind. Mehrere Verfahren zur Modellbildung und Ableitung von Testfällen für kombinierte Hardware-Software Systeme sind in [70] aufgeführt.

Die Arbeiten in [9] evaluieren verschiedene Algorithmen zur Generierung von Testfällen, die für den Leistungstest von softwaregetriebenen Telekommunikationssystemen benötigt werden. Alle Algorithmen basieren auf einer Markov-Kette (Vgl. [22], S. 1ff), die ein Zustandsmodell des zu testenden Systems repräsentiert. Die drei Algorithmen wurden an verschiedenen realen Programmen getestet, die interne Verwaltungaufgaben im Telekommunikationssystem übernehmen.

Ein Ansatz zur Erzeugung von Testfällen für ASICs[7], die MPEG2-Dekodierung in Multimediasystemen, bspw. bei DVD-Playern, durchführen, wird in [63] erläutert. Der Test dieser Systeme wird als Scheduling-Problem zwischen den einzelnen, interagierenden Komponenten aufgefasst und dann als lineares Optimierungsproblem beschrieben.

Für ähnliche Anwendungsfelder des digitalen Schaltungsentwurfs, jedoch viel eher im Entwurfsprozeß, wird in [46] ein Ansatz vorgestellt, bei dem das Verhalten des Systems aus der Spezifikation in eine formale Beschreibung aus Zeit- und Wertedarstellungen überführt wird. Diese formale Spezifikation wird zur automatischen Generierung von Stimuli und der automatischen Überprüfung der Systemausgaben verwendet. Die dynamische Berechnung erfolgt durch Lösung eines linearen Optimierungsproblems mit einem Simplex-Algorithmus. Die

---

[7] Application Specific Integrated Circuit

erzeugten Stimuli dienen als Eingabe für die Simulation der Systemimplementierung in der Hardwarebeschreibungssprache VHDL[8]. Die Ausgaben werden durch die ebenso automatisch generierte Testbench ermittelt und mit den erwarteten Ausgaben verglichen.

Zueigen ist allen Ansätzen, daß zunächst ein mehr oder minder formales Modell vom eigentlichen System gebildet wird und dieses dann als Grundlage zur Generierung genutzt wird.

3.4. Validierung zellulärer Netze

Zelluläre Netze, ganz gleich welcher Generation, sind die Ergebnisse intensiver Entwicklungsarbeit der jeweiligen Infrastrukturanbieter und -betreiber. Folglich ist es schwierig, Details über Entwurfs- und Testprozesse der Unternehmen zu bekommen.

Ein Kernkonzept zur Leistungsvalidierung eines zellulären Netzes sind so genannte „Key Performance Parameter“ (KPI). Diese definieren quantifizierbare Maße für verschiedene Parameter der Leistungserbringung im Mobilfunknetz und werden während der Laufzeit ermittelt. Genutzt werden diese unter anderem im Rahmen der Abnahme und der Überprüfung der Vertragserfüllung, aber auch zur Verbesserung der Konfiguration des Netzwerkes während der Betriebszeit.

Die Parameter sind zum einen konkrete technische oder physikalische Eigenschaften, die leicht bestimmt werden können - bspw. Latenzen, Fehlerraten oder Datendurchsätze (Vgl. [65], S. 30ff; [17]). Gebräuchlich sind auch dienstabhängige Größen, wie die Anzahl der Nutzer in einer Zelle, oder protokollspezifische Fehler (Vgl. [65], S. 30ff).

Die KPI sind Bestandteil von Forschungs- und Entwicklungsarbeiten. Auf Herstellerseite ist das Interesse groß, durch entsprechende Entwicklung möglichst gute Werte für die KPI zu erzielen, da diese vertragsrelevant sind. Der Netzbetreiber ist wiederum interessiert, durch günstige Konfiguration der Netze, die ebenfalls Einfluß auf die KPI und damit implizit auch auf die Kundenzufriedenheit haben, stabile KPI zu erzeugen.

Aufgrund der Komplexität der zellulären Netze ist es jedoch schwierig, KPI zuverlässig zu ermitteln und Veränderungen auf bestimme Systemparameter zurückzuführen. In [12] wird deshalb ein Ansatz vorgestellt, wie Diagnose auf Basis der KPI automatisiert werden kann. Dazu wird das Problem der Diagnose als ein Klassifizierungsproblem aufgefasst. Entsprechend werden Ursachen sowie Symptome modelliert und mittels eines Naiven Bayes-Klassifikators (Vgl. [54], S. 349ff) zugeordnet. Die Verteilung der Zufallsvariablen in dieser Arbeit wird aus früheren Erkenntnissen des Mobilfunknetzes bestimmt. Entsprechend ist die Diagnose stark vom jeweiligen System abhängig.

Ein Ansatz jenseits der automatischen Diagnose, aber mit ähnlichen algorithmischen Lösungen, wird in [55] vorgestellt. Hier wird mittels Selbstorganisierender Karten[9] (Vgl. [54], S. 101ff) versucht, Netzwerkzellen bezüglich einiger hundert KPI in Gruppen einzuteilen und

---

[8] Very High Speed Integrated Circuit Hardware Description Language

[9] Englisch: Self Organising Maps (SOM)

Administratoren einen Überblick über die einzelnen Zustände und Bedingungen der einzelnen Zellen zu verschaffen. Die Visualisierung hilft bei der intuitiveren Arbeit und dem Schlußfolgern aus einer Vielzahl von Parametern.

Für derartige Ansätze ist offensichtlich ein operativer Zugang zum Netz nötig, da die KPI anders schwer zentral ermittelt werden können. Besteht dieser Zugang nicht, müssen alternative Möglichkeiten der dezentralen Ermittlung geschaffen werden. Die deutsche Computerfachzeitschrift CHIP, herausgegeben von der CHIP Communications GmbH München, führt seit dem Jahr 2010 einen „Großen Netztest“[10] durch. Mit diesem soll für Mobilfunkkunden ermittelt werden, welches Netz für welchen Dienst die besten Voraussetzungen bietet.

Getestet wird mit präparierten Fahrzeugen bzw. Testkoffern, wie sie in Abbildung 3.3 dargestellt sind. Beide werden von Testern auf Straßen, in Zügen und zu Fuß auf fest definierten Routen durch Deutschland bewegt. Die Testsysteme führen dabei regelmäßig verschiedene Tests mit handelsüblichen Mobiltelefonen, Tablets und Laptops durch. Getestet werden bspw. der Aufbau von Sprachanrufen, der Aufruf von Internetseiten und auch die Qualität von Videostreams. Diese werden unabhängig für die Netze der vier bedeutendsten deutschen Mobilfunkanbieter durchgeführt und Ergebnisse neben einigen verfügbaren technischen Parametern, wie der Feldstärke des Netzes sowie verfügbare Datenraten, gespeichert. Nach Aus- und Bewertung werden für verschiedene Nutzergruppen Netzbetreiber mit den besten Ergebnissen vorgeschlagen. Beispielhaft sind die Frequenzmessungen des derzeit im Aufbau befindlichen Mobilfunknetzes der 4. Generation (Long Term Evolution (LTE), Vgl. [72], S. 279ff) in Abbildung 3.4 dargestellt.

(a) Testkoffer für die Nutzung in Zügen und öffentlichen Plätzen bzw. Straßen[12]

(b) Ausrüstung eines Testfahrzeugs[13]

Abbildung 3.3.: Testausrüstung für den „Großen Netztest“ der Fachzeitschrift CHIP

---

[10] Informationen zum Test im Jahr 2012: http://chip.de/netztest2012

[12] Quelle: http://www.chip.de/ii/436241230_e52c8bc668.jpg, abgerufen am 03.09.2012

[13] Quelle: http://www.chip.de/ii/436241089_d041bb52a6.jpg, abgerufen am 03.09.2012

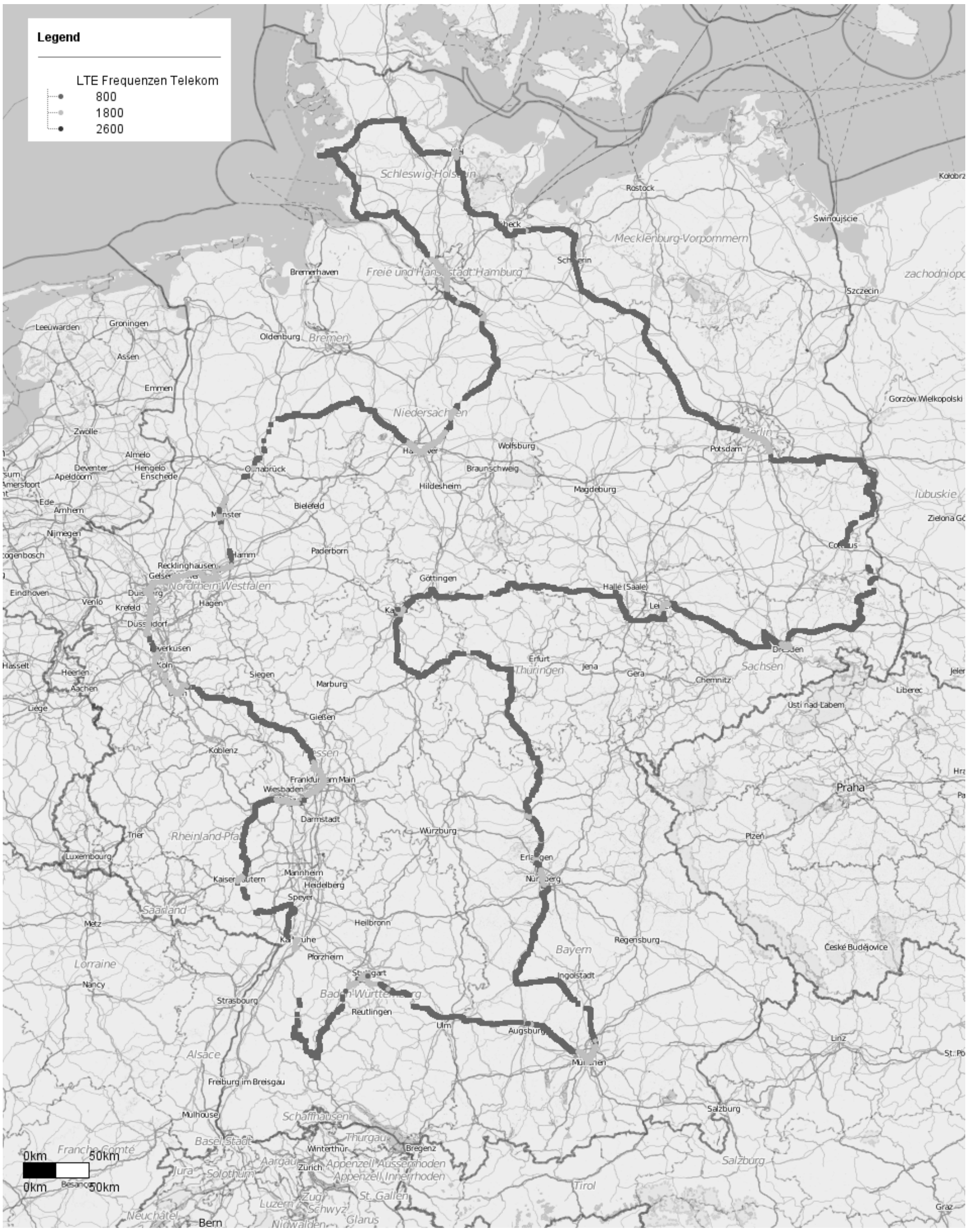


Abbildung 3.4.: Ermittelte Netzabdeckung und Frequenznutzung des LTE-Netzes der Deutschen Telekom (Ermittelt mit Testfahrzeug)[14]

Dieses Verfahren kommt vollständig ohne Kenntnisse über die Architektur oder Zugriff auf interne Zustände des zu testenden Systems aus. Ebenso ist die Definition der zu testenden Parameter nicht an für den Nutzer irrelevante physikalische oder technische Größen gebunden, sondern orientiert sich ausschließlich am Erfolg oder Mißerfolg der Nutzung eines Endkundendienstes, bspw. das Durchführen eines Anrufes oder dem Aufruf einer Internetseite.

---

[14] Quelle: http://www.chip.de/ii/529721611_f7171d4770.png, abgerufen am 03.09.2012

Allerdings ist der Aufwand für den Test immens und die räumliche und auch zeitliche Testabdeckung eher gering. Die feste Route für das Fahrzeug wurde einmal abgefahren, ebenso wie die Route für die Messungen in Zügen. Diese decken natürlich nur einen Bruchteil der Fläche Deutschlands ab, wie in Abbildung 3.4 deutlich zu erkennen ist, und ignorieren vor allem ländliche Gebiete, in denen die Dienstqualität erfahrungsgemäß geringer ist. Da jeder Punkt räumlich nur einmal passiert wurde, können außerdem zufällige, zeitliche Effekte die Ergebnisse beeinflussen.

Da die Ergebnisse dieses Netztests großes Medienecho fanden, haben auch die Netzbetreiber ein Interesse, möglichst gute Ergebnisse - natürlich unter minimalem technischen, organisatorischen und finanziellen Aufwand - zu liefern. Eine Möglichkeit, diesen Test über einen längeren Zeitraum und auf wesentlich größerer Fläche, jedoch unter ähnlichen Testbedingungen, durchzuführen, würde die Aussagekraft verbessern.

Das in dieser Arbeit vorgestellte Validierungskonzept bietet einen Ansatz, den Test entsprechend auszubauen.

3.5. Algorithmische Probleme

In der Zielsetzung dieser Arbeit wird das algorithmische Problem der Testpartitionierung genannt. Dieses hat Ähnlichkeiten zu einigen wohlbekannten und in der Literatur definierten Problemen. Eine Auswahl sowie einige Lösungsansätze werden im folgenden vorgestellt.

3.5.1. Kombinatorische Optimierungsprobleme

Zahlreiche realweltliche Probleme können in Form kombinatorischer Optimierungsprobleme formuliert und gelöst werden - häufig mittels Graphenalgorithmen oder als ganzzahliges lineares Optimierungproblem (Vgl. [52], S. XIII ff). Viele die ser Probleme sind als NP-schwer bekannt, so daß entsprechend zahlreiche Approximationsalgorithmen zur effizienten Bestimmung von Lösungen mit einer bestimmten Güte existieren.

Ein bekanntes Kernproblem ist RUCKSACK[15]. Dies ist in seiner allgemeinen Form wie folgt definiert (Vgl. [52], S. 473):

gegeben: $c_1, ..., c_n \in \mathbb{N}, w_1, ..., w_n \in \mathbb{N}, W \in \mathbb{N}$
gesucht: Teilmenge $S \subseteq 1, ..., n : \sum\limits_{j \in S} \leq W \wedge \sum\limits_{j \in S} c_j \overset{!}{=} max$

Gesucht ist also eine Teilmenge von Objekten, die ein bestimmtes Gesamtgewicht W nicht überschreitet aber die Summe des Nutzens der Objekte maximiert.

Das Optimierungsproblem der Teilmengenbestimmung ist NP-schwer (Vgl. [52], S. 477). Das abgeleitete Entscheidungsproblem, ob eine Teilmenge S existiert, die das Maximalgewicht W

---

[15] Englisch: KNAPSACK

nicht überschreitet aber in der Summe einen Nutzwert K erreicht oder überschreitet, ist NP-vollständig (Vgl. [52], S. 477; [50], S. 491).

Greedy-Algorithmen sind Heuristische Lösungsansätze für dieses Problem. Diese ordnen die Objekte zunächst nach einer Effizienz, die durch das Verhältnis Gewicht zu Nutzen definiert wird. Beginnend mit den höchsteffizienten Objekten werden diese dann sequentiell zur Teilmenge S hinzugefügt, bis das Gesamtgewicht überschritten wird. Es wird also sinnbildlich der Rucksack gepackt (Vgl. [50], S. 15ff). Der polynomiellen Laufzeit dieses Algorithmus steht eine geringe Ergebnisgüte gegenüber, was leicht durch entsprechende Beispiele gezeigt werden kann (Vgl. [50], S. 33f).

Auch sind einfache Verfahren mittels dynamischer Programmierung gebräuchlich, die auf der Menge der Objekte und dem Maximalgewicht W arbeiten (Vgl. [50], S. 20ff). Diese sind in der Lage, die optimale Lösung zu finden. Durch die Abhängigkeit vom Wert des Maximalgewichtes sind diese Verfahren pseudopolynomiell und $O(n \cdot W)$ beschränkt (Vgl. [52], S. 477ff).

Ein dem RUCKSACK ähnliches Problem ist BEHÄLTER[16]. Hierbei stellt sich die Frage, wie n Objekte verschiedener Größe auf eine minimale Anzahl von Behältern gleicher Größe verteilt werden können, ohne daß diese Behälter überlaufen.

Wird die Größe der Behälter auf 1 normiert, ergibt sich formal folgendes Problem (Vgl. [52], S. 485; [47], S. 46f):

gegeben: $a_1, ..., a_n, \forall j \in \{1, ..., n\} : 0 < a_j$
gesucht: $k \in \mathbb{N}$ sowie die Zuordnung $f : \{1, ..., n\} \rightarrow \{1, ..., k\}$, so daß
$\forall j \in \{1, ..., k\} : \sum_{\{} i : f(i) = j\} a_i \leq 1$

Viele praktisch relevante Fragestellungen aus der Logistik und der Produktion lassen sich über das Behälterproblem modellieren. Folglich sind auch zahlreiche Lösungsansätze analysiert worden. Prinzipiell gilt aber auch für BEHÄLTER, daß das Optimierungsproblem, also das Finden der Aufteilung auf eine minimale Anzahl von Behältern, NP-schwer (Vgl. [47], S. 46f) und das Entscheidungsproblem, ob eine Lösung für eine definierte Anzahl von Behältern existiert, NP-vollständig ist (Vgl. [52], S. 486ff).

Auch hier bieten verschiedene Greedy-Heuristiken eine Möglichkeit zur polynomiellen Lösung, die aber nicht garantiert optimal ist. Die meisten beruhen darauf, die zu verteilenden Objekte sequentiell nach bestimmten Taktiken auf die Behälter zu verteilen. Ggf. werden die Objekte vorab der Gewichte nach sortiert. Je nach Taktik resultieren Schranken von $O(n)$ bis $O(n \cdot \log(n))$.

Ein ähnliches Problem wie BEHÄLTER ist das Problem PARTITION, das in verschiedenen Ausprägungen existiert. Eine weitverbreitete Definition lautet (nach [59], S. 14):

gegeben: $A = \{a_1, ..., a_n\}, s : A \rightarrow \mathbb{N}$
gesucht: Menge $A' \subseteq A$, so daß $\sum\limits_{a \in A'} s(a) = \sum\limits_{a \in A \setminus A'} s(a)$

---

[16] Englisch: BIN PACKING

Gesucht ist also für eine Menge von Objekten, die ein Gewicht s haben, die Aufteilung in zwei Partitionen gleichen Gewichtes. Auch hier gilt die gleiche Aussage zu NP-Schwere und NP-Vollständigkeit des Optimierungsproblems und des Entscheidungsproblems (Vgl. [38], S. 60ff, 39ff).

Etwas abweichend wird in [44] (S. 5) das „Partitionierungsproblem“ wie folgt definiert:

gegeben: $O = \{o_1, ..., o_n\}, c : P \rightarrow \mathbb{R}$
gesucht: Partitionierung $P = \{p_1, ..., p_m\}$, so daß

- $\forall i, j, i \neq j : p_i \cap p_j = \varnothing \qquad \wedge$
- $\bigcup_{i=1}^{m} p_i = 0 \qquad \wedge$
- $c(P) = min!$

Gesucht wird also eine Zuordnung der Objekte o zu artitionen, wobei jedes Objekt genau einer Partition zugeordnet sein muß. Die Kosten, die aus dieser Partitionierung entstehen, sollen minimiert werden. Im Gegensatz zu den anderen, o.g. Problemen werden die Kosten bzw. Gewichte nicht als (Linear-) Kombination der Elemente oder Objekte ermittelt, sondern einer Partitionierung zugeordnet.

Die Definition dieses Problems resultiert in erster Linie aus dem Anwendungsfeld der Systempartitionierung beim Entwurf. Prinzipiell läßt sich das Partitionierungsproblem auf ein ganzzahliges, lineares Optimierungsproblem abbilden und somit vollständig optimal lösen. Neben diesem Ansatz mit exponentieller Laufzeit existieren auch Heuristiken mit polynomieller Laufzeit. Diese werde in konstruktive und iterative Verfahren unterschieden (Vgl. [44], S. 5ff; [86], S. 390ff).

Die konstruktiven Verfahren gruppieren in mehreren Schritten Objekte und Partitionen zu größeren Partitionen, bis ein bestimmtes Abbruchkriterium erreicht ist. Die Entscheidungen, welche Objekte gruppiert werden, basieren auf Closeness-Funktionen, die die realweltliche Relationen zwischen den einzelnen Objekten abbilden sollen. Bekanntester Vertreter ist das „Hierarchical Clustering“ in zahlreichen Abwandlungen.

Iterative Verfahren hingegen bewerten bestehende Partitionierungen bezüglich einer Kostenfunktion. Durch Manipulation der Partitionierung entstehen neue Varianten, die wiederum bewertet und mit vorherigen Lösungen verglichen werden können. Die Ausführungszeit eines Algorithmus wird durch die Komplexität der Kostenfunktion und der Menge der iterierten Partitionierungen bestimmt. Bekannter Vertreter ist der „Kernighan-Lin“-Algorithmus, der hauptsächlich für die Partitionierung großer digitaler Systeme eingesetzt wird und in zahlreichen Adaptionen existiert (Vgl. [86], S. 395ff).

3.5.2. Paralleles Rechnen

Ein praktisches Anwendungsfeld der im vorangegangenen Abschnitt genannten Algorithmen sind Fragestellungen rund um paralleles und verteiltes Rechnen. Hier liegt der Fokus vor allem auf der Aufteilung von Applikationen in verteilbare bzw. parallelisierbare Tasks sowie die Zuordnung dieser zu den Rechenknoten.

Die Aufteilung der Applikation wird häufig Partitionierung genannt (Vgl. [59], S. 5) und ist in der Problemstellung ähnlich definiert wie PARTITION im vorangegangen Abschnitt. Bei der Zuteilung wird von „Scheduling“ gesprochen (Vgl. [79], S. 5f). Wird das Scheduling zur Entwurfszeit, also vor der Ausführung des parallelen Programms, durchgeführt, wird dies statisches Scheduling genannt. Werden hingegen die Zuordnungen der Tasks während der Laufzeit im parallelen System geändert, ist von dynamischem Scheduling oder „Load Balancing“ die Rede.

Wie ähnlich die Fragestellungen sind, zeigt sich an der formalen Definition des Problems MULTIPROCESSOR SCHEDULING (nach [38], S. 65):

gegeben: Eine endliche Menge A von Tasks, eine Dauer $l(a) \in \mathbb{N}$ für jedes $a \in A$, eine Anzahl $m \in \mathbb{N}$ von Prozessoren und eine Zeitgrenze $D \in \mathbb{N}$.

gesucht: Existiert eine Partitionierung $A = A_1 \cup A_2 \cup ... \cup A_m$ von A in m disjunkte Teilmengen, so daß $max\{\sum_{a \epsilon A_i} l(a) : 1 \leq i \leq m\} \leq D$.

In diesem Falle ist das Problem selbst als Entscheidungsproblem formuliert, das ebenso NP-vollständig ist (Vgl. [38], S. 65f).

Prinzipiell ist die Dauer der Berechnung im Falle des statischen Scheduling nicht so entscheidend, wie im Falle des dynamischen. Wie in Abbildung 3.5 ersichtlich, wird die Auslastung des Rechnersystems beständig überwacht und bei Abweichung von Parametern entsprechend nachgeregelt. Das Verfahren ist vergleichbar zu Regelkreisen in der Steuerungstechnik. Alle Prozesse in diesem Regelkreis führen jedoch zu Verzögerungen, so daß sich im ungünstigen Falle zum Zeitpunkt des Einstellens neuer Werte der Zustand des Systems bereits wieder relevant geändert hat. Entsprechend schnell sollte insbesondere der Regler sein, der die Neuberechnung der Lastverteilung übernimmt.

Als Last in einem Wartesystem wird dabei „die Summe der zum Zeitpunkt t von der Bedieneinheit noch abzuarbeitenden Bedienzeit“ verstanden ([59], S. 27). Ein Wartesystem ist ein „System, in dem momentan nicht ausführbare Aufträge warten, ohne abgebrochen zu werden.“ ([59], S. 20).

Verfahren und Algorithmen zur Ausführung des statischen und dynamischen Schedulings hängen stark von den Rahmenbedingungen der Applikation und des Parallelrechensystems ab. Vor allem bei dynamischen Verfahren gibt es zahlreiche Möglichkeiten, wie Lasten ermittelt werden, wo Entscheidungen über Lastverteilungen getroffen werden und wie diese umgesetzt werden. Eine entsprechende Taxonomie ist in [66] zu finden. Wichtige Fragen sind, in welchen Phasen die Verfahren zentral oder dezentral arbeiten und ob das Rechnersystem homogen oder heterogen ist.

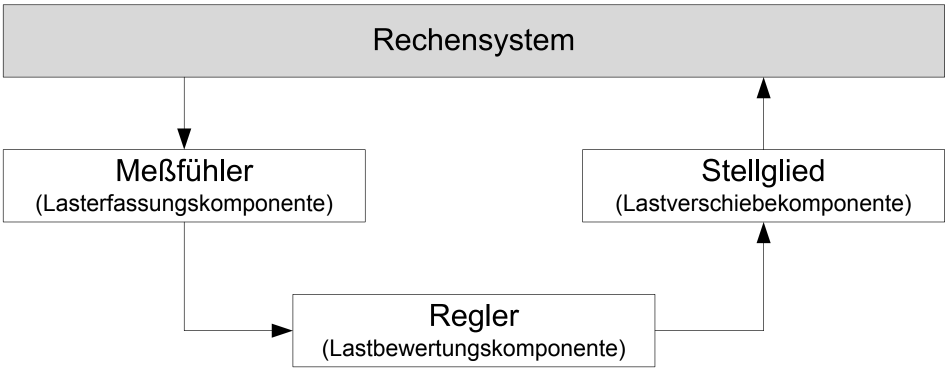


Abbildung 3.5.: Lastverwaltung in Parallelrechnern, Vgl. [59], S. 7

Die meisten zentralen Verfahren basieren auf dem Systemkonzept aus Abbildung 3.5 und verfügen über Warteschlangen, deren Elemente durch heuristische Lösungen auf Basis von Kostenmodellen auf die vorhandenen Prozessoren verteilt werden (Vgl. [43]; [85]; [42]).

3.6. Zusammenfassung

Im zurückliegenden Kapitel wurden bestehende Ansätze und Lösungen zu Themen, die diese Arbeit tangiert, vorgestellt. Dies betrifft in erster Linie Verfahren zum automatisierten Testen unter Leistungsgesichtspunkten. Neben Ansätzen zum Test von Software umfaßt dies auch andere Kategorien, wie Hardware oder auch Lastverteilungsverfahren für Parallelrechensysteme.

Ein Großteil der Verfahren sind im jeweiligen Modell und der Implementierung spezialisiert auf ein kleines Anwendungsfeld. Es fehlt ein abstraktes, disziplinenübergreifendes Konzept, das zudem heterogene Strukturen auf Seiten des zu testenden Systems wie auch auf Seiten der Testbench berücksichtigt. Das Konzept dieser Arbeit, das genau an dieser Stelle ansetzt, wird im folgenden Kapitel detailliert eingeführt. Einige der vorgestellten Ansätze finden sich darin wieder.

4. Validierungskonzept und Testbench

Es ist Ziel dieser Arbeit, ein Konzept und ein Framework zur Verfügung zu stellen, welches die Validierung eines Dienstbereitstellenden Systems ermöglicht.

In diesem Kapitel wird das Konzept erläutert. Der Ansatz zur Ausführung der Validierung wird vorgestellt und die Architektur der Testumgebung, die sich aus dem Ansatz und den zu benennenden Anforderungen ergibt, spezifiziert.

4.1. Einordnung

Bereits im Abschnitt 2.1 wurden die Begriffe Test, Verifikation und Validierung eingeführt bzw. definiert. Demzufolge ist Validierung die Überprüfung, ob das „richtige Produkt“ geschaffen wurde. Es soll also überprüft werden, ob der Kunde bekommt, was er haben will - nicht, was in eine Spezifikation geschrieben wurde.

Möglich und sinnvoll ist diese Betrachtung nur auf Systemebene, da sich hier erst das Gesamtverhalten ergibt, was durch den Kunden in Verhältnis zu seinen Wünschen bzw. Forderungen gesetzt werden kann. Folglich muß auch die Definition und Ausführung der Validierung des Systems als ein zusätzlicher Schritt an der Spitze des V-Modells stattfinden.

Da auch vollständig heterogene SPS validiert werden sollen, kann es vorkommen, daß verschiedene Teile des SPS verschiedene Entwurfsprozesse durchlaufen haben. Die Systemvalidierung findet dann an der Spitze all dieser, unter Umständen parallel ausgeführten, Entwurfsprozesse statt.

Entscheidend für den Erfolg einer Validierung im Vergleich zur Verifikation ist, daß zwischen Auftraggeber und Entwicklern ein klares, bestenfalls natürliches Verständnis der zu überprüfenden Parameter existiert. Es ist nicht Ziel der Validierung, einzelne Punkte der Spezifikation „abzuhaken“. Vielmehr steht im Mittelpunkt des Auftraggebers, die Einsatzfähigkeit des entstandenen Systems für den entsprechenden Einsatzzweck zu überprüfen.

Wird die Validierung auf die Entwicklungsschritte des V-Modells aufgesetzt, entstehen zwei Phasen für die Validierung, die umgesetzt werden müssen. Die erste Phase besteht in der Definition der Testfälle. Diese sollte noch vor der Erstellung der Spezifikation stattfinden, da mit dieser bereits technische Spezifika Einzug halten, die zu Mißverständnissen führen können. Ziel muß es sein, testbare Parameter zu identifizieren, die auf Seiten von Auftraggebern und Entwicklern absolut verständlich sind.

Insbesondere für heterogene Systeme sollten die testbaren Parameter nicht technologie- oder architekturgebunden sein. Verschiedene Entwicklergruppen mit verschiedenen Ausbildungsschwerpunkten haben hier nicht zwangsläufig das gleiche Verständnis, so daß sowohl bei der Definition der Testfälle als auch bei der Interpretation der Ergebnisse Mißverständnisse oder Fehler entstehen können. Hinzu kommt, daß der Auftraggeber unter Umständen von keinem der Entwicklungsschwerpunkte tiefgreifende Kenntnisse hat.

Für Dienstbereitstellende Systeme, wie sie im Abschnitt 2.3.2 definiert wurden, bietet es sich deshalb an, die Struktur des Systems auszublenden und sich auf den funktionalen „Zweck“ der Systeme zu konzentrieren. Dieser besteht darin, den Klienten die Dienste bereitzustellen. Eine sinnvolle Validierung ist es demzufolge, die korrekte Funktion dieser Dienste unter verschiedenen funktionalen und nichtfunktionalen Rahmenbedingungen zu überprüfen.

Hierzu ist es Teil des Konzeptes dieser Arbeit, die Nutzung und Bereitstellung der Dienste quantifizierbar (per Definition SPS) und damit nachvollziehbar und vergleichbar zu machen. Die Validierung wird infolge zu einem Soll-/ Ist-Vergleich. Dazu müssen die relevanten Parameter auf abstrakte mathematische Modelle, die für alle Beteiligten verständlich und eindeutig sind, abgebildet werden. Mit dieser Definition der Validierungsparameter und -anforderungen ist ein natürlich verständliches Lastenheft erstellt. Durch Ableitung der Spezifikation und unter Anwendung der üblichen Entwicklungsprozesse ist dieses System umzusetzen. Am Ende des gesamten Entwicklungsprozesses, der im V-Modell auch die Verifikation auf den verschiedenen Ebenen enthält, steht die Ausführung der Validierung.

In Anbetracht der Menge und Komplexität der Dienste sowie der Menge der zu verwaltenden Klienten ist eine automatisierte Ausführung des Tests fast zwingend erforderlich. Dafür wird ein Validierungssystem benötigt, welches die eingangs definierten Modelle bzw. Instanzen dieser als Eingabe nimmt und die Validierung entsprechend ausführt sowie die Ergebnisse speichert und aufbereitet.

Diese Form der Ausführung kann im Rahmen der Produktentwicklung zweifach genutzt werden. Zum einen bietet es den Entwicklern die Möglichkeit, das gesamte System zu testen. Auch wenn die Ergebnisse aufgrund ihrer Abstraktion nur bedingt Rückschlüsse auf mögliche Ursachen von auftretenden Mängeln erlaubt, so kann zumindest die „Anwesenheit“ von Schwächen ermittelt werden. Mit entsprechender Erfahrung und dem Zugriff auf Systemausgaben ist über eine zeitliche Korrelation auch eine grobe Fehlerursachenforschung möglich. Die automatische Ausführung der Validierung, die durch die mathematischen Modelle möglich sind, erlaubt langläufige Testreihen mit geringen Aufwänden.

Auf der anderen Seite erlaubt die Validierung dem Auftraggeber des Systems einen Abnahmetest, der exakt das überprüft, was im realen Betrieb auch vom entwickelten System gefordert wird. Mißverständnisse und Interpretationen technischer Details der Spezifikation treten durch die Abstraktion der Validierungsparameter nicht auf. Auch wiederholende Abnahmen veränderter Systemarchitekturen mit gleichen Funktionalitäten stellen kein Problem dar.

Kernpunkt für eine erfolgreiche Umsetzung beider Varianten - auch für verschiedene Arten von SPS - ist die Frage, wie die Dienste und deren Verfügbarkeit definiert sowie quantifiziert werden können und eine Ausführung der Validierung auf Basis dieser Quantifizierung ermöglicht werden kann. Grundlage bildet die Kapselung der Funktionen des zu testenden Systems, also des

SPS, in einzelne Dienste. Die Nutzung dieser Dienste ist, per Definition, applikationsspezifisch quantifzierbar. D.h., es kann für einen bestimmen Zugriff eines Klienten auf einen Dienst eine natürliche Zahl angegeben werden, die die Last dieses Zugriffs beschreibt (Vgl. Abschnitt 2.3.2). Die Summe der einzelnen Lasten aller Klienten ergibt die gesamte Last für einen Dienst.

Basierend auf diesem Modell wird für diese Arbeit folgende Hypothese aufgestellt (Vgl. [20]).

Hypothese 1 (Validierung Dienstbereitstellender Systeme)**.** *Die kontrollierte Erzeugung einer veränderlichen Menge von Anfragen (Last) durch eine veränderliche Menge von Klienten sowie die Aus- und Bewertung der Durchführung ist ein wirksames Mittel zur Validierung Dienstbereitstellender Systeme.*

Erfahrungen, die zu dieser Aussage und damit zum Ansatz dieser Arbeit führten, stammen aus konkreten Industrieprojekten im Bereich der Mobilfunkinfrastruktur. Hier wurde in der letzten Phase des Tests vor der Einführung neuer Produktgenerationen ein aufwändiger Feldtest gestartet. Eine Instanz des zu testenden Mobilfunknetzes wurde in einem Teil der Firmenzentrale aufgebaut und durch Mitarbeiter genutzt. Hierzu standen Mobiltelefone mit entsprechenden SIM-Karten zur Verfügung. Als Ergänzung wurden weitere Telefone per Datenkabel an PCs angeschlossen und von einem zentralen Skriptsystem ferngesteuert.

Die Testingenieure sahen seinerzeit zwei wesentliche Optimierungspotentiale. Zum einen sollte die Zahl der gezielt fernsteuerbaren Mobiltelefone erhöht und zum anderen die Kosten für den Test gesenkt werden. Insbesondere die Lösung, Mobiltelefone per Computer fernzusteuern und somit Aktivitäten im Mobilfunknetz durchführen zu lassen, erwies sich als teuer und bezüglich einiger Testeigenschaften, wie der Mobilität der Netznutzer, als zu unflexibel.

Bezogen auf das Modell der SPS haben Mitarbeiter bzw. zentrale Testskripte unter Nutzung der Telefone Last in das zu testende Infrastruktursystem gebracht. Da diese Methodik für zahlreiche Produktzyklen Anwendung fand, fußt die Hypothese - zumindest in der Anwendung als ein Systemtestwerkzeug - auf praktischen Erfahrungen der Industrie. Die Abstrahierung vom Mobilfunknetz zum Dienstbereitstellenden System ist naheliegend.

Der Fokus dieser Arbeit liegt auf der Definition der benötigten Modelle sowie der Bereitstellung des Frameworks. Mit den Modellen wird es möglich sein, den ursprünglichen Systemtestansatz der Mobilfunknetzwerke auf die Validierung Dienstbereitstellender Systeme anzuwenden. Das Framework wird die automatische Ausführung der Validierung ermöglichen.

Inwieweit die Hypothese praktikabel ist und eine veritable Validierung realer Systeme erlaubt, liegt nicht im akademischen Ermessen sondern muß sich durch praktische Anwendung zeigen.

Die weitere Betrachtung des Validierungssystems erfolgt aus Sicht der Entwickler, nicht der Auftraggeber. In diesem Umfeld ist die Begrifflichkeit der Validierung eher unüblich. Vielmehr wird von Test gesprochen - also dem Werkzeug zur Durchführung von Verifikation und Validierung (siehe Abschnitt 2.1). Demzufolge wird in den folgenden Ausführungen häufiger der Begriff Test verwendet, da sich hieraus auch zahlreiche ableitende Begriffe ergeben, die intuitiv verständlich und gebräuchlich sind.

4.2. Ansatz

Ausgangspunkt für das SPS-Validierungssystem bildet eine lauffähige Instanz des SPS. Dieses wird in Anlehnung an existierende Literatur auch als System Under Test[1] (SUT) bezeichnet. Wie in Abbildung 4.1 erkennbar, wird die Struktur des Systems vollkommen ausgeblendet und lediglich die Menge von Diensten, die durch das SPS zur Verfügung gestellt werden, betrachtet. Desweiteren ist eine Menge von Klienten notwendig, die technologisch in der Lage sind, auf die Dienste des SPS zuzugreifen. Dies geschieht über Request/Response-Mechanismen. Die Klienten basieren auf verschiedenen Plattformen und verfügen über verschiedene Ressourcen, so daß nicht jeder auf alle bzw. die gleichen Dienste zugreifen kann.

Es ist nun Ziel, diese Klienten mittels geeigneter Kommunikationsmechanismen „fernzusteuern“ und dadurch von einem zentralen *Automatisierungssystem* aus Lasten für das SUT zu erzeugen. Im Gegensatz zu statischen Skriptsystemen soll die erzeugte Last im Gesamtsystem definierten *Vorgaben* der Testingenieure folgen und damit der Hypothesenaussage „Die kontrollierte Erzeugung einer veränderlichen Menge von Anfragen“ (siehe Hypothese 1, Abschnitt 4.1) gerecht werden. Auf diese Weise wird es für den Test ermöglicht, schwankendes Nutzerverhalten abzubilden. Am Beispiel des Mobilfunknetzwerkes ist es so möglich, das über die Tageszeit variierende Verhalten von Firmen- und Privatnutzern als Muster zu reproduzieren.

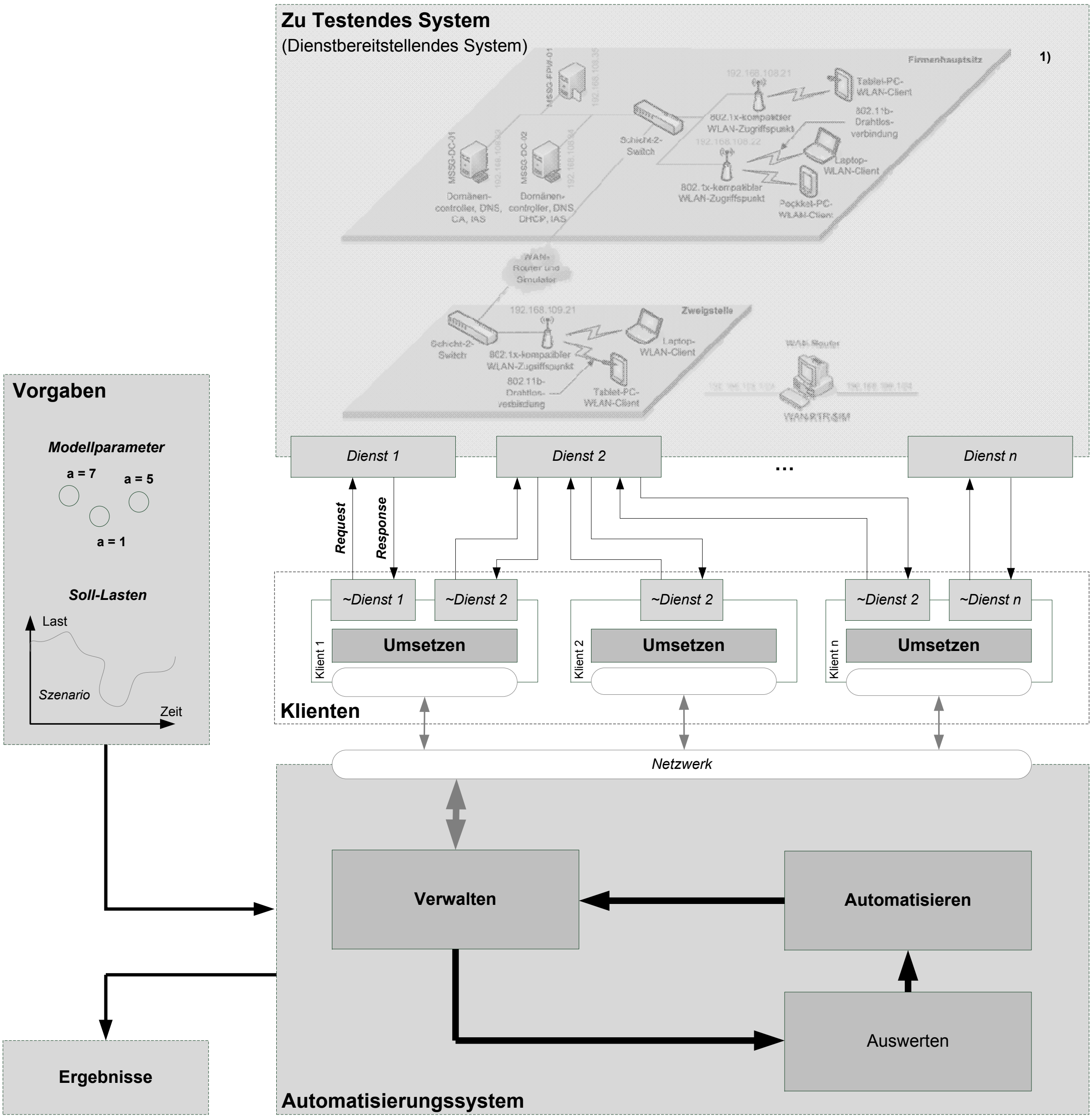
Jeder Klient ist in der Lage, durch Nutzung der Dienste eine bestimmte, quantifizierbare Last für diesen Dienst zu erzeugen. Hierzu müssen auf dem Klienten entsprechende Implementierungen zur Nutzung der Dienste umgesetzt sein (in Abbildung 4.1 „*∼Dienst X*“ genannt) was fast zwangsläufig gegeben ist. Bspw. ist ein Mobiltelefon per se in der Lage, den Sprach- oder meist auch den Datendienst des Mobilfunknetzes zu nutzen. Wie genau die Lastquantität, die der jeweilige Klient generiert, zu bestimmten Zeitpunkten kontrollierbar ist, hängt vom jeweiligen Dienst und der Abbildung auf Lasten ab.

Zur Fernsteuerung durch ein zentrales Automatisierungssystem müssen die Klienten eine Möglichkeit zur Kontrolle der Dienstzugriffe bieten. Außerdem wird ein *Netzwerk* zur Kommunikation zwischen Automatisierungssystem und Klienten benötigt. Ein *Umsetzer* sorgt für die logische Verbindung zwischen Netzwerk und Dienstzugriffsmechanismen.

Ein Test wird in diesem Konzept generiert, indem der Ingenieur für jeden Zeitpunkt eines auszuführenden Tests eine *Soll-Last* für jeden der Dienste des SUT angibt. Während der Ausführung wird zu diskreten Zeitpunkten die geforderte Last für jeden der Dienste von einem *Automatisierungsalgorithmus* automatisch in Testaufträge dekompositioniert. Die Aufträge werden an die entsprechenden Klienten übertragen. Hierzu ist eine *Verwaltung* der verfügbaren Klienten und deren Zustände sowie der Verbindungen notwendig. Abhängig von der jeweiligen Technologie des Klienten bzw. der Art des zu nutzenden Dienstes wird der Testauftrag vom Klienten umgesetzt bzw. ausgeführt. Parameter und Ergebnisse des Testauftrages werden an das Automatisierungssystem zurückgemeldet, wo eine entsprechende *Auswertung* erfolgt und ggf. Adaptionen durch neu generierte Testaufträge vorgenommen werden. Ziel der Automatisierung

---

[1] Deutsch: Zu testendes System



1) *Quelle: http://www.microsoft.com/germany/technet/datenbank/articles/900017.mspx abgerufen, 10.10.2012*

Abbildung 4.1.: Ansatz zur lastbasierten Validierung von SPS sowie grundlegende Struktur der Systeme ist es, die Summe der aktuell generierten Lasten aller Klienten dienstabhängig an die Soll-Last anzunähern.

Mit der *Auswertung* der Rückmeldungen erfolgt auch eine Ergebnisaufbereitung, die vom Testingenieur zur Bewertung der durchgeführten Validierung genutzt werden kann.

Das Konzept, bestehend aus den verschiedenen Teilaufgaben, muß in eine entsprechende Struktur überführt werden, die verschiedene Anforderungen zu erfüllen hat. Diese lehnt sich an die Konzepte zur Testautomatisierung und zur Lastverteilung an, wie sie u.a. in den Abbildungen 3.1 und 3.5 zu finden sind.

4.3. Anforderungen

Bereits aus dem relativ einfachen Beispiel des Mobilfunknetzwerktests ergeben sich einige entscheidende Anforderungen an das Validierungssystem. Durch die Verallgemeinerung auf lastbasierte Validierung von SPS ergeben sich weitere Rahmenbedingungen, die bei der Konzeptionierung und Umsetzung eines entsprechenden Testframeworks zu berücksichtigen sind. Dieses Testframework, das aus verschiedenen Komponenten zur Definition und Ausführung der Systemvalidierung dient, wird im folgenden *Testbench* genannt.

Voraussetzung für den Einsatz des Validierungskonzeptes ist die Möglichkeit, von zentraler Stelle aus mit den Klienten zu kommunizieren und auf diesem Weg fernzusteuern sowie Statusinformationen zu beziehen. Da es sich bei der Testbench und den Klienten üblicherweise um softwaregetriebene Digitalsysteme handelt, muß folglich auch ein digitaler Kommunikationsmechanismus dazwischen existieren. Im praktischen Alltag wird es sich in aller Regel um einen Bus oder ein standardisiertes Netzwerk handeln.

Ist die Kommunikationsmöglichkeit gegeben, liegt die größte Herausforderung in der Heterogenität und der Dynamik der Klienten. Diese entsteht durch Klienten, die auf verschiedenen Technologien basieren und vor allem über verschiedene Ressourcen verfügen. In Folge dessen ist nicht jeder Klient technologisch in der Lage, jeden Dienst nach belieben zu nutzen. Unter Umständen gibt es für bestimmte Dienste des SPS keine Unterstützung auf bestimmten Klienten. Auch betriebsmittelbedingte Ausschlüsse der Parallelnutzung bestimmter Dienste sind denkbar.

Dies führt zum einen zu differierenden Implementierungen der *Umsetzer* auf den Klienten. Zum zweiten muß die Testbench - im Gegensatz zu einigen anderen Ansätzen, bspw. im Bereich von Cluster- oder Grid-Tests - die Einschränkungen bei der Generierung von Testaufträgen berücksichtigen und das Gesamtsystem entsprechend optimieren.

Desweiteren führen die beschränkten Ressourcen und ggf. auch das Kommunikationsnetz zu einer Dynamik in der Menge der Klienten, die während der Laufzeit des Tests behandelt werden muß. Die Nichterreichbarkeit von Klienten bzw. auch der Ausfall von Klienten ist kein Fehler, der zum Abbruch der Validierung führt, sondern allenfalls zur Reorganisation innerhalb der Testverteilung auf die Klienten.

Am Mobilfunkbeispiel wird die Problematik schnell klar. Das zu testende SPS ist das Mobilfunknetz, als Klienten dienen mobile, batteriebetriebene Mobiltelefone und Laptops. Die Bereitstellung der Kommunikation zwischen Klienten und Testbench kann in diesem Falle über einen Dienst des SPS, also das Mobilfunknetz selbst, erfolgen. Per Datenlink, bspw. nach GSM- oder UMTS-Standard, können Informationen zwischen Testbench und Klient ausgetauscht werden. Dies setzt jedoch voraus, daß alle am Test beteiligten Klienten den Dienst der Datenübertragung unterstützen. Auf den Klienten selbst muß ein Programm in Form einer Software oder eines Skriptes hinterlegt werden, das die Kommunikation mit der Testbench steuert und entsprechende Kommandos in Dienstaufrufe des Klienten zum SPS auslöst. Auf Grund schlechter Netzabdeckung im Mobilfunknetz oder einer leeren Batterie können Mobiltelefone während der Ausführung von Tests ausfallen. Auch eine Wiederinbetriebnahme des Klienten ist denkbar. Diese Dynamik in der Verfügbarkeit der Klienten muß ebenso vom System unterstützt werden.

Sowohl im Mobilfunkbeispiel wie auch in vielen anderen denkbaren SPS-Testszenarien kann die Zahl zu verwaltender Klienten hoch sein - denkbar ist eine Anzahl von einigen hundert. Auch die Anzahl verschiedener Dienste, die parallel getestet werden sollen, kann mehrere Dutzend betragen. Sämtliche Strukturen der Testbench müssen folglich auf diese Größenordnungen hin konzipiert werden.

Unter Berücksichtigung der Verzögerungszeiten, die für die Kommunikation zwischen Testbench und Klienten - und zurück - notwendig sind, ergibt sich zwangsläufig, daß ein Systemtest mit geringen Zyklen- bzw. Reaktionszeiten nur in sehr eingeschränkten Fällen denkbar ist. Das vorgestellte Konzept ist vielmehr vorgesehen, langlaufende Validierungsphasen automatisiert ausführen zu können. Kurzfristige Abweichungen zwischen Soll- und Ist-Last, die bspw. durch Störungen in der Klientenarchitektur hervorgerufen werden, müssen tolerierbar sein.

4.4. Modell

Zur Umsetzung des präsentierten Ansatzes unter den ebenso genannten Anforderungen ist ein formales Modell notwendig, das alle quantifizierbaren Parameter enthält. Es dient zum einen der Modellierung des Nutzerverhaltens sowie der Lastvorgaben, wie auch der Beschreibung der technologischen Rahmenparameter, bspw. maximaler Lasten. Eine Instanz dieses Modells wird als Eingabe für einen automatisch ausführbaren Test genutzt.

Zur Veranschaulichung der einzelnen Größen sind diese in Abbildung 4.2 an den einzelnen Komponenten des Validierungskonzeptes notiert.

4.4.1. SPS und Klienten

Zunächst muß das realweltliche System aus SPS und Klienten modelliert werden. Die bereitgestellten Dienste D des SPS und die verfügbaren Klienten S werden als Mengen definiert.

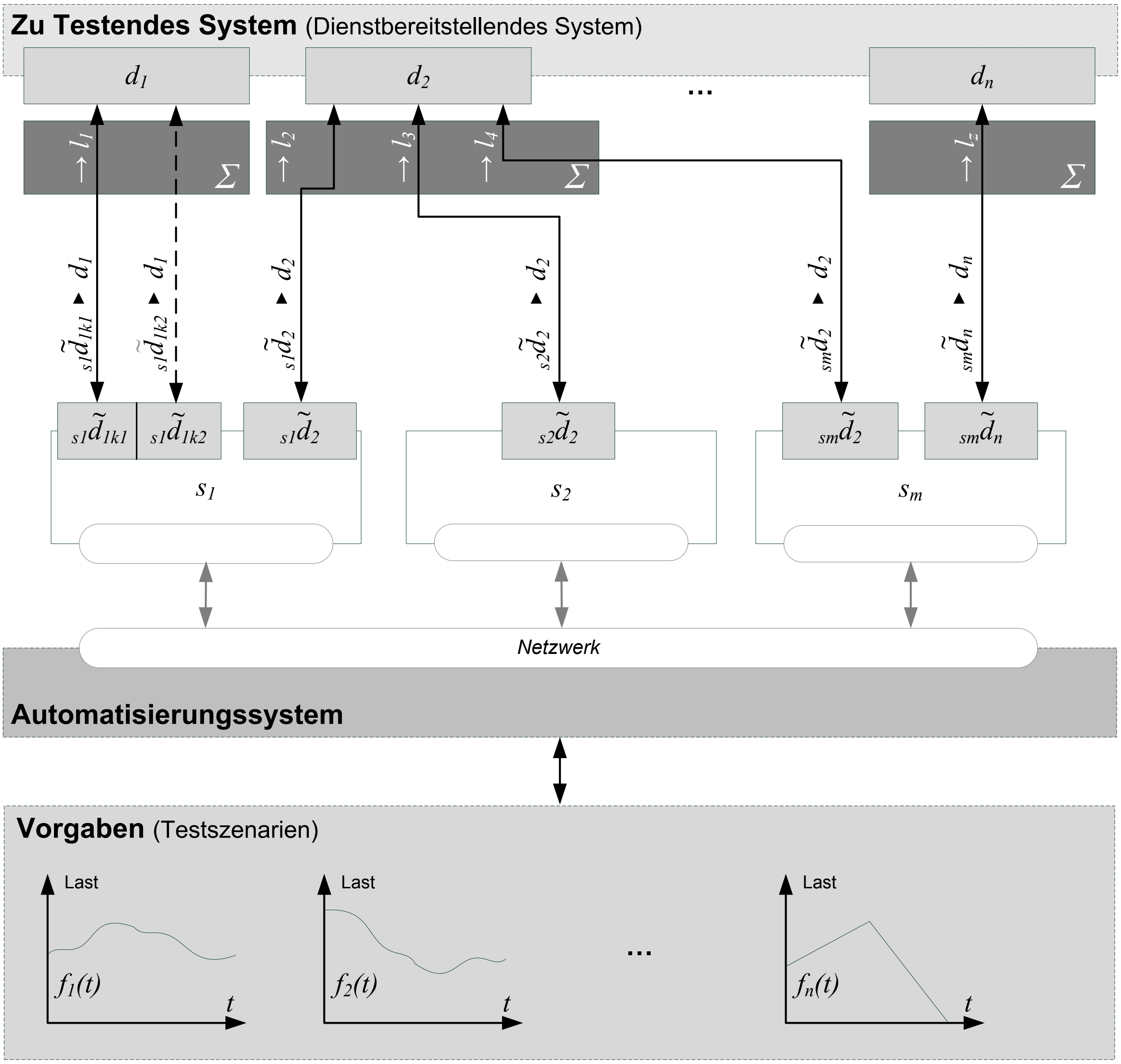


Abbildung 4.2.: Darstellung der Modellgrößen - die Zustände K der verschiedenen Dienstimplementierungen sind nur für Dienst 1 von Klient 1 angegeben.

$$D = \{d_1, d_2, ..., d_n\} \tag{4.1}$$
$$S = \{s_1, s_2, ..., s_m\} \tag{4.2}$$

In vielen praktischen Fällen, wie dem Mobilfunknetzbeispiel, wird die Anzahl n der Dienste sehr viel kleiner sein als die Anzahl m der Klienten, die diese Dienste nutzen.

Auf jedem Klienten s_i muß nun für die zu nutzenden Dienste d_j eine Zugriffsimplementierung ${}_{s_i}\tilde{d}_j$ realisiert sein. Wie diese im Detail umzusetzen ist, hängt vom jeweiligen Dienst und der Beschaffenheit des Klienten ab. In jedem Falle können die jeweiligen Realisierungen technologiebedingt und plattformabhängig variieren. Demzufolge gilt für die Menge aller Dienstimplemtierungen $\tilde{D}$ auf den Klienten folgendes:

$$\tilde{D} \subseteq (D \times S) \tag{4.3}$$

Desweiteren kann sich jede Instanz einer Zugriffsimplementierung $\tilde{d} \in \tilde{D}$ in verschiedenen Zuständen $k_l \epsilon K$ befinden. Diese Zustände sind ebenso technologieabhängig und beschreiben bspw. verschiedene Betriebs- oder Protokollmodi für den Zugriff auf den jeweiligen Dienst und führen unter Umständen zu differierenden Lasten.

Basierend auf diesen Mengen wird die Relation „*Nutzen*“ $\rhd$ definiert.

$$\rhd \subseteq ((\tilde{D} \times K) \times D) \tag{4.4}$$

Diese Relation gibt also an, daß ein Klient mit den entsprechenden Protokollimplementierungen, die sich in einem bestimmten Zustand befinden, auf die Dienste des SPS zugreift. „*Nutzt*“ (von der Relation „Nutzen“) nun ein Klient s_i den SPS-Dienst d_j mit der zugehörigen Implementierung ${}_{s_i}\tilde{d}_j$, die sich im Zustand k_l befindet, wird dieser Sachverhalt entsprechend notiert:

$${}_{s_i}\tilde{d}_{j k_l} \blacktriangleright d_j \tag{4.5}$$

In dieser Betrachtung der „Nutzung“ eines Dienstes fehlt in erster Überlegung noch eine Behandlung der Zeit. Intuitiv ist es eine semantisch vollständigere Aussage, wenn ein Klient einen Dienst zum Zeitpunkt t nutzt. Bei genauerer Überlegung kann das synchrone Konzept jedoch sinnvoller durch die asynchronen Zustände K der Zugriffsimplementierung $\tilde{d}$ umgesetzt werden. Immer dann, wenn sich $\tilde{d}$ im Zustand k_l befindet, nutzt sie den SPS-Dienst entsprechend der hinterlegten Logik. Dabei ist es unerheblich, ob dieser Fall zum Zeitpunkt t_1 oder t_2 - oder sogar zu beiden Zeitpunkten - auftritt. Entsprechend kann die Zeit als weitere Kardinalität in dem

Modell (noch) außen vorgelassen werden. Der Wechsel zwischen den einzelnen Zuständen kann vom Klienten selbst oder durch entsprechende Testaufträge durch die Testbench ausgelöst werden.

Zur Abdeckung der Anforderungen aus dem Konzept fehlt noch die Modellierung der Last - also der quantitativen Bewertung der Relation „Nutzen“. Dazu genügt die Definition einer zugehörigen Funktion, die jeder Nutzungsvariation, bestehend aus dem Klienten s_i, dessen Implementierung $\tilde{d}_j$ zur Nutzung des Dienstes d_j sowie dem zugehörigen Zustand k_l, eine natürliche Zahl zuordnet.

$$last : \triangleright \rightarrow \mathbb{N} \quad , \quad ({}_{s_i}\tilde{d}_{j k_l} \blacktriangleright d_j) \mapsto l \tag{4.6}$$

Die Definition der Lastfunktion erfolgt durch einen Testingenieur. Da die Repräsentation des Wertes den gesamten Testablauf beeinflußt, muß die Beziehung zwischen der realweltlichen „Belastung“ eines Dienstes sinnvoll auf die Last abgebildet werden.

All die benannten Mengen und Konstrukte beschreiben logische oder physische Eigenschaften des SPS und der Klienten. Zur Ausführung der Validierung mittels eines automatisierten Tests auf Basis des vorgestellten Konzeptes sind weitere Beschreibungen notwendig.

4.4.2. Testszenario

Dem oben vorgestellten Ansatz folgend wird ein Test für einen Dienst durch die Vorgabe einer über die Zeit veränderlichen Last definiert. Demzufolge müssen Testingenieure diese *Soll-Last* als eine Funktion der Zeit vorgeben. Während der Ausführung der Validierung wird dann versucht, mit den verfügbaren Klienten eben diese vorgegebene Soll-Last zu erreichen.

Soll-Lasten müssen für alle Dienste vorgegeben werden, so daß sich ein kompletter Validierungsdurchlauf für ein SPS mit n Diensten d_i aus einer Menge von n Funktionen, im folgenden Testszenario genannt, wie folgt definiert:

$$\{f_1(t), f_2(t), ..., f_n(t)\} \quad , \quad f_i : T \rightarrow \mathbb{R} \tag{4.7}$$

T repräsentiert in diesem Falle die Zeit. Zum Zwecke der einfacheren Definition der Soll-Last Funktionen f_i ist die Zeit im Modell als kontinuierlich anzusehen. Für den Durchgang einer Validierung ist die Betrachtung negativer Zeit nicht sinnvoll, da dieser zum Zeitpunkt $t = 0$ gestartet wird. Es ergibt sich für f_i folglich der Definitionsbereich $\mathbb{R}^+$.

Die reellen Zahlen bilden den Wertebereich der Funktionen. Dies steht im Widerspruch zur Modelldefinition in Gleichung 4.6, in der die Last prinzipiell als positive, ganze Zahl definiert ist. Die Erweiterung ist dem Komfort zur Definition des Testszenarios geschuldet, da kontinuierliche, reellwertige Funktionen leichter zu definieren sind. Zwangsläufig werden jedoch die reellen

Funktionswerte aus den Testszenarien im Rahmen der rechnerbasierten Behandlung auf diskrete Funktionswerte projiziert.

Als Voraussetzung für die automatisierte Auswertbarkeit und Interpretierbarkeit dürfen die Funktionen f_i keine Polstellen haben, also an keinem Punkt der Definitionsmenge unendliche oder negativ unendliche Werte annehmen. Im übrigen exisitieren keine Anforderungen an die Testszenarien. Folglich sind auch Unstetigkeitsstellen, die keine Polstellen sind, erlaubt und zum Teil für gewisse Testfälle auch durchaus sinnvoll.

Während der Ausführung eines Testszenarios, die zum Zeitpunkt $t = 0$ beginnt, muß das Automatisierungsmodul der Testbench auf Basis geeigneter Metriken und mit Hilfe geeigneter Algorithmen den jeweiligen Soll-Lastwert für jeden Dienst in Testaufträge zerlegen und den verfügbaren Klienten zuordnen. Ziel dieser *Testpartitionierung*, wie der entsprechende Vorgang im folgenden genannt wird[2], ist, daß die Summe der Lasten aller Testaufträge sowie der aktuell von den Klienten ausgeführten Aufträge genau der im Szenario geforderten Last entspricht.

Da die Modellierung dieser Sachverhalte eng an den Lösungsalgorithmus gebunden ist, werden entsprechende formale Modelle im folgenden Kapitel 5 vorgestellt.

4.5. Architektur der Testbench

Neben den Modellen zur Beschreibung von Tests im Sinne des vorgestellten Konzeptes ist auch eine Testbench notwendig, die Instanzen der Modelle als Eingabe zur Ausführung einer Validierung benutzt und die im Abschnitt 4.3 definierten Anforderungen erfüllt bzw. berücksichtigt (Vgl. [19]).

Zur Realisierung der Testbench müssen die einzelnen Aufgaben, die in Abbilung 4.1 dargestellt sind, auf entsprechende, interagierende Komponenten abgebildet werden. Da die einzelnen Funktionen nicht zwangsläufig vom gleichen Nutzer verwendet werden bzw. auch wegen entsprechender Server- und Netzwerkstrukturen auch auf verschiedene Rechner verteilt werden müssen, werden die Funktionen auch in verschiedene Module aufgeteilt, die untereinander über Netzwerkprotokolle kommunizieren.

---

[2] Das algorithmische Problem der PARTITION ist in der Literatur bereits definiert und deckt sich mit dem hier beschriebenen Problem nicht vollständig. Da Kernaussagen aber übereinstimmen, wird im folgenden die Zerlegung einer Last in Teillasten und das Zuordnen zu verfügbaren Klienten als *Testpartitionierung* bezeichnet.

Ein Überblick über die resultierende Architektur ist in Abbildung 4.3 gegeben. Die Testbench wird durch vier Module realisiert, die zum Teil wiederum aus weiteren Teilkomponenten mit speziellen Aufgaben bestehen:

1. Testklient
2. Testservermodul
3. Automatisierungsmodul
4. Auswertungsmodul

Verantwortlich für die Lastgenerierung an einem Dienst sind die Klienten. Auf diesen muß eine Applikation „*Testklient*“ ausgeführt werden, die mit der übrigen Testbench kommuniziert sowie die Lastaufträge empfängt und durch Nutzung lokaler Ressourcen ausführt.

Da gemäß der Anforderungen die Klienten nicht zwangsläufig während der gesamten Testausführung verfügbar bzw. erreichbar sind, ist eine zentrale Komponente notwendig, die die logischen Kommunikationsverbindungen zu allen Klienten verwaltet. Sämtliche Zustandsinformationen über die einzelnen Klienten und die aktuell zugeordneten Lasten werden in der „*Testservermodul*“ genannten Komponente gesammelt und gespeichert.

Die eigentliche Ausführung des Tests, wie er durch eine Instanz des im Abschnitt 4.4 beschriebenen Modells definiert ist, erfolgt durch das „*Automatisierungsmodul*“. Hier werden die Testszenarien unter Berücksichtigung verfügbarer Klienten und deren jeweiliger Zustände bzw. Parameter in Lastaufträge für die einzelnen Klienten partitioniert und an das Testservermodul weitergeleitet. Dieses propagiert dann die Aufträge an die Klienten.

Die Testberichte der Klienten werden im Testservermodul gesammelt und in geeigneter Weise, bspw. in einer Datenbank, gespeichert. Von da aus kann mit dem „*Auswertungsmodul*“ während bzw. nach der Testausführung eine Analyse von Soll- und Istverhalten des SUT durchgeführt werden.

Die Grundfunktionalität der Module ist SUT-unabhängig und kann für verschiedene Systeme genutzt werden. Es gibt jedoch einige Module, die an spezielle Plattformen, Kommunikationstechnologien oder zum Teil auch an effiziente Teststrategien adaptiert werden müssen.

Der Datenfluß zwischen den Modulen und Teilkomponenten ist durch die schwarzen Pfeile in Abbildung 4.3 dargestellt. Zum Start einer Validierung werden Modellparameter und Testszenarien mittels einer Graphischen Oberfläche (GUI) eingegeben und an das Testautomatisierungsmodul übertragen. Dort findet eine initiale Testpartitionierung statt, deren Ergebnis eine Menge von Testaufträgen für die einzelnen Klienten ist. Die Testaufträge werden vom „*Testgenerator*“ parametrisiert, also bspw. mit Zieltelefonnummern oder URLs versehen. Die komplettierten Aufträge werden an das Testservermodul übertragen, wo diese über die gespeicherten Verbindungskanäle an die jeweiligen Testklienten übertragen werden. Zeitgleich werden Informationen über die laufenden Tests der Klienten in Datenstrukturen und einer Datenbank des Testservermoduls abgelegt.

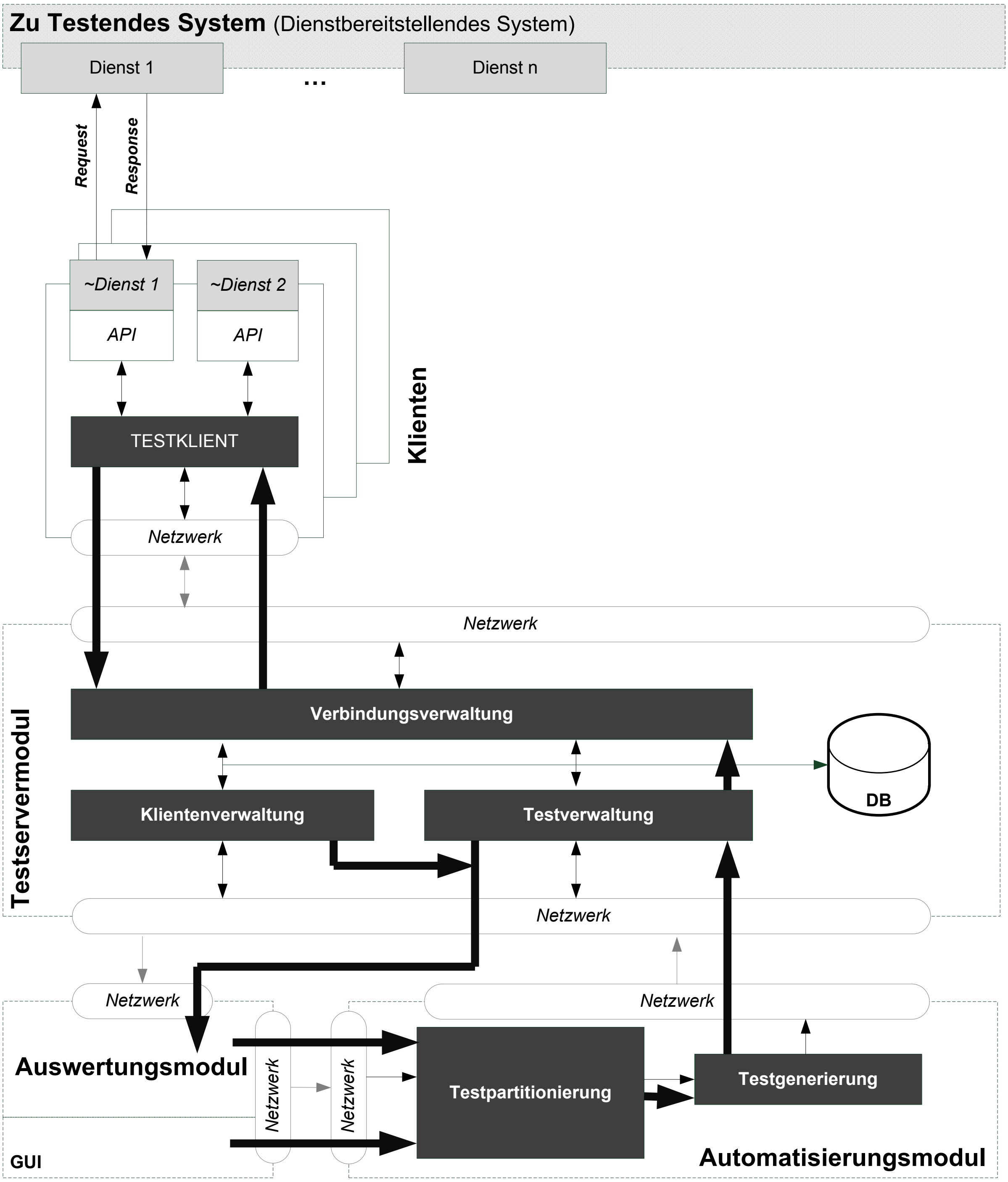


Abbildung 4.3.: Architektur der Testbench und Darstellung der Module

Der Testklient führt empfangene Testaufträge aus, indem er mit den verfügbaren Technologien der Klienten auf die Dienste des SPS zugreift. Während der Ausführung bzw. nach Abarbeitung des Auftrags werden Parameter oder Ergebnisse an das Testservermodul gesendet. Dort werden diese mit den zugehörigen Test-Datenstrukturen verbunden und an das Auswertungsmodul gesendet. Die Relevanz der Ergebnisse wird dort bewertet und entsprechende Aktualisierungen, vor allem über die tatsächlich erzeugte Dienstlast der Klienten, an das Automatisierungsmodul gesendet. Das kann entsprechend auf Abweichungen reagierten und der Kreislauf beginnt infolge von vorn.

Im folgenden werden die allgemeinen Funktionen und Besonderheiten der einzelnen Module vorgestellt. In Kapitel 6 wird eine konkrete, technologiegebundene Implementierung des Systems zum Test zellulärer Netze erläutert.

4.5.1. Testklient

Die Klienten dienen als Dienstkonsumenten für das SPS. Um sie innerhalb der Testbench als Lasterzeuger zu verwenden, müssen zwei Voraussetzungen erfüllt werden.

Zum einen muß es gelingen, Systemfunktionen des Klienten, die den Zugriff auf die Dienste des SPS bewerkstelligen, aus einer eigenentwickelten Anwendung bzw. aus einem Skript heraus zu nutzen und bestenfalls zu parametrisieren. Üblicherweise bedingen verschiedene Dienste auch systemseitig verschiedene Zugriffstrategien, so daß für die durch den Klienten unterstützten Dienste auch unterschiedliche Zugriffsmethoden realisiert werden müssen.

Zum zweiten muß diese Anwendung auch in der Lage sein, über ein Medium mit dem Testservermodul zu kommunizieren, da von dort die Informationen über die zu erzeugenden Lasten eines jeden Klienten übermittelt werden. Es ist hierbei zunächst unerheblich, ob dabei ein dediziertes Netz oder einen Dienst des SPS selbst als Kommunikationsmedium verwendet wird.

Im Ganzen wird folglich ein Umsetzer[3] - Testklient genannt - gesucht, der die Aufträge vom Testservermodul empfängt und auf dem Klienten umsetzt. Zusätzlich müssen Testergebnisse sowie Zustandsinformationen, die für die Ausführung und die Auswertung der Tests zentral notwendig sind, ebenso über ein Kommunikationsmedium an das Testservermodul gesendet werden.

Prinzipiell ist es auch möglich, daß der Klient als Gerät von Nutzern verwendet wird. Ein einfaches Beispiel hierfür ist wieder das Telefon im Mobilfunknetz. Wird dieses im Rahmen von Feldtests an Nutzer gegeben, so verwenden diese das Gerät in nicht plan- oder kontrollierbarer Art zur Nutzung der Dienste. Wird dieses Gerät durch Installation eines Testklienten auch in die Testbench integriert, kann es in einer hybriden Nutzung zur Validierung des SPS beitragen. In Abhängigkeit von der jeweiligen Technologie ist anzustreben, daß die Last, die durch den Nutzer verursacht wird, ebenso ermittelt und damit im laufenden Test der Testbench berücksichtigt wird.

Es ist eine zentrale Annahme für das vorgestellte Validierungskonzept, daß die genutzten Klienten nicht identisch sein müssen. Verschiedene Plattformen sowie Ressourcen müssen ebenso

---

[3] Englisch: Wrapper

berücksichtigt werden wie differierende und schwankende Qualität im Zugriff auf die Dienste sowie auf die Kommunikationsmedien. Demzufolge muß der Testklient ggf. auch für verschiedene Plattformen umgesetzt werden, bzw. auf den gleichen Plattformen konfigurierbar gestaltet werden. Durch entsprechende Zustandsmeldungen muß dem Testservermodul mitgeteilt werden, welche Dienste zum aktuellen Zeitpunkt vom Klienten überhaupt genutzt werden können, so daß die Testpartitionierung dies berücksichtigen kann.

Demzufolge ist der Testklient systemabhängig. Werden das SUT und damit auch die Klienten gewechselt, so ist auch die Implementierung der Testklienten anzupassen. Abhängig von den konkreten Plattformen und Technologien können plattformübergreifende Paradigmen, wie bspw. Java, genutzt werden, um den Aufwand zur Anpassung gering zu halten und auf die hardwareabhängigen Zugriffe zu beschränken.

Folglich ist es auch sinnvoll - insofern technisch möglich - die Kommunikation zwischen Testservermodul und Testklient auf verbreitete Technologien, wie bspw. ein IP-Netz, abzubilden. Damit wird die Wahrscheinlichkeit erhöht, die Implementierung auch für weitere Testszenarien zu verwenden.

4.5.2. Testservermodul

Während der Testklient als Umsetzer zwischen der übrigen Testbench und den Klientensystemen zur Nutzung der zu testenden Dienste wirkt, fungiert das Testservermodul zum einen als Relais zwischen dem Automatisierungsmodul und dem Klientennetz sowie als Zwischenspeicher für die Zustände der einzelnen Klienten.

Zur Erfüllung dieser Aufgaben benötigt das Testservermodul zwangsläufig Zugriff auf das Kommunikationsmedium zum Informationsaustausch mit den Klienten. Zur Laufzeit kontaktiert entweder das Testservermodul die bekannten Klienten oder die Klienten kontaktieren das ihnen bekannte Testservermodul. Die Wahl des Paradigmas hängt stark von technischen Parametern, wie die Netzwerkstruktur oder Sicherheitsmaßnahmen (Firewalls), ab. An dieser Stelle können auch testbench-interne Sicherheitsmaßnahmen integriert werden, die bspw. nur bekannten Klienten die Teilnahme am Test erlaubt. Auch Verfahren zur Sicherung der Kommunikation, bspw. Signaturen oder Verschlüsselungen, können je nach verfügbaren Ressourcen umgesetzt werden. Es müssen hierbei jedoch vor allem die beschränkten Ressourcen der Klienten berücksichtigt werden.

Besteht Verbindung zu einem Klienten, wird der jeweilige Zustand in einer zentralen Datenstruktur des Testservermoduls verwaltet. Hierzu zählen in erster Linie Informationen wie die Erreichbarkeit (Netzadressen), verfügbare Ressourcen, implementierte Dienste, maximal generierbare Lasten und aktuell generierte Lasten für die einzelnen Dienste. Je nach Anwendungsfall können weitere Informationen, bspw. die aktuelle Position, hinzukommen. Informationen über Zustandsänderungen werden durch den Klienten an das Testservermodul gemeldet, so daß die Informationen der Datenstrukturen angepaßt werden können. Je nach verwendeter Kommunikationstechnologie kann das Testservermodul auf verschiedene Arten (Timeout, ...) auch den Verlust der Verbindung erkennen und demzufolge den Zustand des Klienten auf inaktiv aktualisieren.

Zustandsänderungen einzelner Klienten werden an das Auswertungsmodul weitergeleitet, so daß dieses entscheiden kann, ob eine Neuberechnung der Testpartitionierung durchgeführt werden muß. Wird eine solche Berechnung durchgeführt, empfängt das Testservermodul entsprechend parametrisierte Laständerungen für einzelne Klienten und leitet diese an die entsprechenden Empfänger weiter.

Die Durchführung der Lastgenerierung seitens der Klienten wird in Form von Berichten an das Testservermodul gemeldet. Diese Informationen sind für eine direkte oder spätere Auswertung des durchgeführten Tests notwendig. Da diese Berichte nicht nur zur Laufzeit des Testservermoduls relevant sind, werden Sie in einem geeigneten persistenten Speicher außerhalb der Laufzeitumgebung gespeichert.

Da das Testservermodul in der Lage sein muß, zahlreiche Klienten zu verwalten und Verbindung zu diesen zu gewährleisten, empfiehlt sich die Umsetzung auf einer leistungsfähigen Serverplattform. Es liegt nahe, den persistenten Speicher in dem Falle über ein Datenbanksystem zu realisieren, wobei dieses nicht notwendigerweise auf der gleichen Plattform installiert sein muß. Es ist außerdem möglich, auch die Zustandsinformationen der Klienten nicht nur in internen Datenstrukturen zu verwalten, sondern diese auch in die Datenbank zu retten. Auf diese Weise gelingt es, die Auswertung und Visualisierung eines laufenden Tests weitestgehend von der Laufzeit des Testservermoduls zu trennen und entsprechende Kapazitäten für den Testbetrieb zu nutzen. Insbesondere zur Visualisierung eines laufenden Tests wäre ein aktives Datenbanksystem sinnvoll, welches bei bestimmten Änderungen der Datensätze automatische Aktionen in einer graphischen Oberfläche oder dem Auswertungsmodul triggern kann (Vgl. [82], S. 18).

Die Implementierung des Moduls ist durch den zwingend erforderlichen Zugriff auf das Kommunikationsmedium zum Klientennetz plattformgebunden. Auch durch die Wahl von Betriebssystem und Datenbanksystem wird logischerweise eine Plattformabhängigkeit geschaffen. Aus diesem Grund ist es sinnvoll, das Testservermodul auf standardisierten und verbreiteten Systemen umzusetzen und damit den Aufwand für die Anpassung an andere Plattformen gering zu halten. Kernpunkt bildet auch hier das Medium zur Kommunikation mit den Klienten. Gelingt es hier, verbreitete Netztechnologien, wie bspw. IP-Netze, zu nutzen, kann das Testservermodul im günstigsten Fall sogar SUT-unabhängig realisiert werden. Ändert sich das Anwendungsfeld des Validierungskonzeptes, muß das Modul nicht angepaßt werden, sondern allenfalls der Testklient.

4.5.3. Automatisierungsmodul

Mit dem Testklienten und dem Testservermodul sind organisatorische und technische Grundvoraussetzungen geschaffen, um überhaupt zentral einen Test mit heterogenen und verteilten Klienten durchführen zu können. Zur Umsetzung eines automatisierten Tests nach dem vorgestellten Konzept und den genannten Modellen wird ein weiteres Modul benötigt, welches die abstrakten, dienstbezogenen Testszenarien auf die verfügbaren Klienten abbildet. Das zugehörige Modul wird Automatisierungsmodul genannt und ist der Kern dieser Arbeit.

Die Testpartitionierung stellt dabei den aufwändigsten Teil der Automatisierung dar. Eingabe bilden die Testszenarien. Diese beschreiben für jeden Dienst des SPS die Last, die in Summe von allen Klienten zu bestimmten Zeitpunkten erzeugt werden sollen. Durch einen geeigneten Algorithmus muß daraus dann die Last errechnet werden, die jeder der verfügbaren Klienten generieren soll.

Gesteuert wird diese Aufteilung durch ein internes Modell, das über Funktionen und Relationen in der Lage ist, technologische Parameter und ein bestimmtes Verhalten der Klienten zu beschreiben. Erst damit wird es möglich, bspw. das Verhalten einer Menge von Nutzern eines Mobilfunknetzwerkes automatisiert nachzubilden und aus den Ergebnissen des gesamten Tests Rückschlüsse auf das Verhalten des SUT unter realweltlichen Bedingungen zu ziehen. Es ist offensichtlich, daß dieses interne Modell entscheidenden Einfluß auf die Verwendbarkeit der Validierungsergebnisse hat.

Bei der Berechnung der Testpartitionierung sind außerdem die aktuellen Lasten, die zum Zeitpunkt der Berechnung durch die Klienten für die einzelnen Dienste bereits generiert werden, zu beachten. Das Ergebnis der Berechnung ist dann eine positive oder negative Laständerung für jeden verfügbaren Klienten. Ist die Änderung ungleich null, wird ein entsprechender Lastauftrag generiert. Dieser beinhaltet Informationen über den betroffenen Klienten und den Dienst sowie die Angabe über die zu generierende Last. Ob die absolute Last oder die relative Änderung angegeben wird, hängt vom jeweiligen Test ab.

Was an dieser Partitionierung zunächst weitestgehend trivial klingt, ist bei näherer Betrachtung ein komplexes algorithmisches Problem. Eine Lösung dessen ist demzufolge zentraler Bestandteil dieser Arbeit und wird im folgenden Kapitel 5 detailiert beschrieben.

Diese Testaufträge werden nach der Partitionierung im Testgenerator parametrisiert. Verschiedene Diensttechnologien und -prozesse bedingen bestimmte Parameter, die zur Ausführung notwendig sind. Meist handelt es sich hierbei um Endpunkte für Kommunikationsrufe. Ein offensichtliches Beispiel liefert das Mobilfunk-SPS. Soll ein Klient Last erzeugen, indem er einen Sprachruf startet, so wird für diesen Vorgang eine Telefonnummer benötigt, die angerufen werden kann.

Entscheidend für die Menge verfügbarer Parameter und deren Zuordnung zu den Klienten ist die Anforderung, daß der Parameter keinen Einfluß auf die erzeugte Last im Sinne der in Gleichung 4.6 eingeführten Semantik haben darf. Dies würde zur Verfälschung der von der Testpartitionierung berechneten und von den Klienten dann tatsächlich generierten Lasten führen. Auch müssen die bereitgestellten Ressourcen der Parameterendpunkte zu jeder Zeit genügend Kapazitäten für die Bearbeitung der Lasten aller Klienten zur Verfügung stellen. Kann bspw. ein Anruf nicht gestartet werden, weil die Nummer außerhalb des SUT wegen zu vieler Anrufe nicht erreichbar ist, würde dies die Testergebnisse beeinflussen. Es empfiehlt sich daher, die Parameter so zu wählen, daß die Endpunkte außerhalb des SUT liegen. Auf diese Weise können Kapazitäten einfacher bereit gestellt und mögliche Abhängigkeiten bzw. Konflikte zwischen einzelnen Tests vermieden werden. Desweiteren kann durch die Zuweisung der Parameter seitens des Testgenerators auch sichergestellt werden, daß die Auslastung auf verschiedene Endpunkte gleichmäßig verteilt wird.

Sind die Lastaufträge parametrisiert, werden diese an das Testservermodul weitergeleitet. Da Automatisierungs- und Servermodul auf verschiedenen Systemen ausgeführt werden können, ist ein geeignetes Kommunikationsmedium zwischen beiden notwendig. Da die Verwendung von Standardrechentechnik erstrebenswert ist, können IP-Netze mit verschiedenen Transportprotokollen ohne große Aufwände verwendet werden. Auf gleichem Weg sendet das Testservermodul Zustandsänderungen der Klienten an das Automatisierungsmodul - wenn auch über den formalen Zwischenschritt des Auswertungsmoduls. Ein Komparator muß vergleichen, ob auch nach der Zustandsänderung die tatsächlich erzeugte Last noch immer innerhalb der Toleranzen im Vergleich zur Soll-Last liegt. Ist das nicht der Fall, muß ein neuer Testpartitionierungsvorgang gestartet werden.

Parallel dazu werden Statusinformationen des Testautomatisierungsmoduls im Datenbanksystem des Testservermoduls gespeichert, so daß spätere Auswertungen möglich sind.

Die Implementierung der Testpartitionierungskomponente kann unabhängig vom jeweiligen zu testenden System erfolgen - vorausgesetzt, es sollen keine SUT-spezifischen Parameter für eine Heuristik verwendet werden, die eine effizientere Lösung des algorithmischen Problems „Testpartitionierung“ erlaubt. Eine derartige Unabhängigkeit kann für den Testgenerator nicht sichergestellt werden, da ein impliziter Zusammenhang zwischen den Parametersätzen, den Strategien zur Parametrisierung und den zugehörigen Diensten existiert. Demzufolge ist unter Umständen eine Adaption des Testgenerators an das jeweilige SUT notwendig.

4.5.4. Auswertungsmodul

Um die Aufwände für die Validierung zu rechtfertigen, ist natürlich auch eine Auswertung der ausgeführten Tests notwendig. Hierfür steht ein Auswertungsmodul zur Verfügung. Dieses dient im Automatisierungskreislauf außerdem der Aufbereitung von Statusinformationen des Testservermoduls, die dann dem Automatisierungsmodul zur Verfügung gestellt werden.

Auswertung der Validierung ist in zwei Betriebsarten möglich. Zum einen können laufendee Tests überwacht werden. Parameter, wie die verfügbaren Klienten, aktuelle Lasten sowie die Testszenarien können für das testüberwachende Personal dargestellt werden. Prinzipiell sind auch Eingriffe technisch möglich - bspw. durch Sperren von Klienten oder eine Änderung der Testszenarien. Es hängt jedoch stark vom jeweiligen Test ab, ob ein solcher gezielter Eingriff überhaupt noch möglich ist. Insbesondere bei einer hohen Anzahl an Klienten und Diensten wird ein Überblick über das Geschehen schwierig.

Wichtiger ist die Auswertung abgeschlossener Tests. Hierzu werden die Berichte der Klienten vom Testservermodul abgerufen sowie geeignet ausgewertet und dargestellt. Diese Berichte enthalten Informationen über Erfolg oder Fehlschlag bestimmter Klientenaktionen zu den Zeitpunkten während des Tests. Falls technisch möglich, werden auch Zusatzinformationen gespeichert. Je nach Dienst und Technologie des Klienten können das Fehlermeldungen, Ressourcenbedarf oder ähnliches sein.

Eine Bewertung der Testergebnisse ist allgemein nur auf Basis der Klientenmeldungen möglich. Das setzt voraus, daß zumindest der Erfolg der beauftragten Lasterzeugung durch den Klienten festgestellt und entsprechend gemeldet werden kann. Rückschlüsse, welche Probleme bzw. Fehler innerhalb des SUT zu Mißerfolgen führen, sind nur dann möglich, wenn entsprechende systeminterne Zustände oder Berichte aus dem SUT zur Verfügung stehen. Diese können dann zeitlich in Relation gesetzt werden.

Da das Auswertungsmodul nicht direkt zur Vorbereitung und der Durchführung des Tests benötigt wird, ist die Konzeptionierung und Implementierung nicht Bestandteil dieser Arbeit. Alle Statusänderungen vom Testserver werden an das Automatisierungsmodul weitergeleitet.

4.6. Zusammenfassung

Das Validierungskonzept dieser Arbeit stand im Mittelpunkt dieses Kapitels. Auf Basis praktischer Erfahrung zum Systemtest von Mobilfunknetzwerken wird die Hypothese aufgestellt, daß die Erzeugung von Lasten an den Diensten eines SPS eine effiziente Validierungsstrategie ist.

Vom Beispiel des Mobilfunksystems wurde auf Dienstbereitstellende Systeme abstrahiert und ein mathematisches Modell zur Beschreibung aller relevanten Mengen und Relationen aufgestellt. Die Grundidee besteht darin, die Nutzung eines SPS-Dienstes durch einen Klienten als Relation zu abstrahieren und auf eine quantifizierte „Last“ abzubilden. Vorgabe für jeden Validierung sind Soll-Lasten für alle Dienste des SPS.

Instanzen dieses Modells dienen als Eingabe für eine Testbench, die die Ausführung der Validierung dann automatisiert. Die Testbench wird in verschiedene Module strukturiert, so daß eine verteilte Ausführung möglich ist und eine größtmögliche Unabhängigkeit von den jeweiligen Spezifika der zu validierenden SPS erreicht wird.

Während einige Komponenten der Module als Verwaltungsinstrumentarien einfach implementiert werden können, muß für die Automatisierung ein Algorithmus zur Testpartitionierung gefunden werden, der eine große Anzahl von Klienten und Diensten verwalten kann. Dies ist Bestandteil des nächsten Kapitels

5. Dynamische Testpartitionierung

Die Kernkomponente zur Umsetzung des Konzeptes ist das Testautomatisierungsmodul. Hier werden die abstrakten Eingaben der Testszenarien in einzelne Lastaufträge für die verfügbaren Klienten zerlegt. Da hierzu zahlreiche Nebenbedingungen beachtet werden müssen, ist das resultierende algorithmische Problem komplex und nicht ohne weiteres durch existierende Verfahren effizient lösbar.

In diesem Kapitel wird deshalb das Problem formal beschrieben. Zur Lösung werden zwei Algorithmen sowie deren Umsetzung erläutert. Der erste Ansatz basiert auf einem existierenden Optimierungsprogramm, der zweite ist eine selbst entwickelte Heuristik. Die Verfahren werden mittels theoretischer Betrachtungen und dem Vergleich verschiedener Tests evaluiert.

5.1. Problemdefinition

Aufgabe des Testautomatisierungsmoduls ist es, eine Verteilung der in den Testszenarien geforderten Lasten zu einem diskreten Zeitpunkt auf die verfügbaren Klienten zu berechnen. Es wird im folgenden davon ausgegangen, daß ein Testszenario $\{f_1(t), f_2(t), ..., f_n(t)\}$, wie es im Abschnitt 4.4.2 eingeführt wurde, existiert. Zur Ausführung der Tests steht eine Menge an Klienten $S = \{s_1, s_2, ..., s_m\}$ zur Verfügung. Für einen Algorithmus, der die entsprechende Verteilung berechnen soll, müssen nun Rahmenparameter definiert werden, innerhalb derer der Lösungsraum aufgespannt werden kann.

Diese Parameter sollten das realweltliche System derart modellieren, daß aus den resultierenden Lösungen für die Testpartitionierung auch Rückschlüsse auf den übergeordneten Validierungsprozeß gezogen werden können.

Die ersten Randbedingungen ergeben sich aus der Plattform und den Ressourcen der Klienten. Wie in den Anforderungen an das Konzept (Vgl. Abschnitt 4.3) beschrieben, handelt es sich bei den eingesetzten Klienten nicht um eine homogene Menge leistungsstarker Systeme, sondern vielmehr um Klienten, die auf verschiedenen technologischen Plattformen beruhen und über verschiedene, meist jedoch stark eingeschränkte Ressourcen verfügen. Entsprechend unterliegen die einzelnen Klienten Einschränkungen in der Ausführung von Tests. Dies beeinflußt die maximale, absolute Last, die ein Klient für einen jeweiligen Dienst des SPS erzeugen kann. Im ungünstigsten Falle ist es sogar möglich, daß ein Klient für einen speziellen Dienst keine Last erzeugen kann, da keine entsprechende Implementierung verfügbar ist bzw. keine Ressourcen zur Verfügung stehen.

Aufbauend auf der Funktion $last$, die in Gleichung 4.6 definiert wurde, wird eine Variable ${}^{t}l_{s,i} \in \mathbb{N}$ eingeführt, die die aktuelle, absolute Last angibt, die ein Klient s für den Dienst i zum Zeitpunkt t erzeugt. Aus der Elementmenge der natürlichen Zahlen muß ${}^{t}l_{s,i} \geq 0$ gelten.

Um den klientenspezifischen Einschränkungen gerecht zu werden, muß außerdem eine weitere Konstante $a_{s,i} \in \mathbb{N}$ eingeführt werden. Diese beschreibt die maximale Last, die der Klient s für den Dienst i zuverlässig erbringen kann. Demzufolge muß sich die Last eines Klienten immer zwischen 0 und a bewegen:

$$\forall t \in T, s \in S, i \in [1, n] \quad : \quad 0 \leqq {}^{t}l_{s,i} \leqq a_{s,i} \tag{5.1}$$

Die Zeit T folgt der Definition aus Abschnitt 4.4.2.

Mit diesen Anforderungen werden zunächst grundlegende technische Grenzen in einem Modell beschrieben. Einem Algorithmus zur Verteilung müssen jedoch weitere Metriken zur Verfügung gestellt werden, um die einzelnen Lösungen des immens umfänglichen Lösungsraums bewerten zu können.

5.1.1. Kosten und Einschränkungen

Hierzu wird eine Kostenfunktion für jeden Klienten s und jeden Dienst i definiert.

$$c_{s,i} : T \times \mathbb{N}^n \to \mathbb{R}_0^+ \quad , \quad (t, (l_{s,1}, ..., l_{s,n})) \mapsto c_{s,i} \tag{5.2}$$

Die Kosten eines Dienstes i errechnen sich in Abhängigkeit der jeweiligen Lasten $l_{s,j}$, die der Klient s auf den Diensten $1 \leq j \leq n$ generiert. Auf diese Weise können Kosten klientenseitig auch dienstübergreifend modelliert werden. Auch technologische Abhängigkeiten oder Ressourceneinschränkungen können über dieses Kostenmodell beschrieben werden.

Ein anschauliches Beispiel liefert wieder das Mobilfunknetzwerk. Aufgrund der Beschaffenheit der Mobiltelefone ist es unwahrscheinlich, daß Nutzer zum gleichen Zeitpunkt telefonieren und Kurznachrichten versenden. Dieses Nutzerverhalten muß durch entsprechende dienstübergreifende Kostenfunktionen im Modell abgebildet werden.

Die Zeit wurde als weitere Dimension der Kostenfunktion hinzugefügt. Somit ist es möglich, für verschiedene Zeitpunkte des Tests verschiedene Kosten zu definieren. Sollte diese Dimension nicht benötigt werden, kann sie durch einen eindimensionalen Definitionsbereich oder, der Vereinfachung wegen, als Dimension ganz entfernt werden.

Zur Modellierung eines kompletten Testsystems ist es notwendig, für jeden Dienst, den ein Klient nutzen kann bzw. soll, eine Kostenfunktion aufzustellen. Da dies für jeden Klienten gilt, entsteht eine $n \times m$ Kostenfunktionsmatrix $\mathcal{C}$:

$$\mathcal{C} \quad = \quad \begin{pmatrix} c_{1,1}(t,(l_{1,1},...,l_{1,n})) & \dots & c_{1,n}(t,(l_{1,1},...,l_{1,n})) \\ \vdots & \ddots & \vdots \\ c_{m,1}(t,(l_{1,1},...,l_{1,n})) & \dots & c_{m,n}(t,(l_{1,1},...,l_{1,n})) \end{pmatrix} \tag{5.3}$$

Die Zeilen repräsentieren die Klienten $1...m$, die Spalten die Dienste $1...n$. Prinzipiell ist eine Angabe einer Kostenfunktion für einen nicht unterstützten Dienst eines Klienten nicht notwendig. Der mathematischen Vollständigkeit wegen, müssen diese Kostenfunktionen als konstante Nullfunktionen in der Matrix notiert werden.

Damit kann allerdings nur die Kostenneutralität nicht implementierter Dienste modelliert werden. Für die Korrektheit des Modells, insbesondere das Erreichen der Soll-Last, ist es zwingend nötig, die Lastneutralität einer Nichtimplementierung zu beschreiben. Auch temporäre Nicht-Verfügbarkeit oder der Ausschluß spezieller Klienten während der Validierungsausführung sind wünschenswert.

Hierzu wird nahezu identisch zur Kostenmatrix eine Einschränkungsmatrix $\mathcal{G}$ wie folgt definiert:

$$\mathcal{G} \quad = \quad \begin{pmatrix} g_{1,1}(t,(l_{1,1},...,l_{1,n})) & \dots & g_{1,n}(t,(l_{1,1},...,l_{1,n})) \\ \vdots & \ddots & \vdots \\ g_{m,1}(t,(l_{1,1},...,l_{1,n})) & \dots & g_{m,n}(t,(l_{1,1},...,l_{1,n})) \end{pmatrix} \tag{5.4}$$

Die einzelnen Funktionen $g_{s,i}$ unterscheiden sich jedoch durch ihren Wertebereich von den Kostenfunktionen:

$$g_{s,i} : T \times \mathbb{N}^n \to \{0,1\} \quad , \quad (t,(l_{s,1},...,l_{s,n})) \mapsto g_{s,i} \tag{5.5}$$

Die Einschränkung $g_{s,i}$ eines Klienten s für einen Dienst i bildet nur auf die Werte 0 oder 1 ab und beschreibt, ob der Klient in der Lage ist, Last für diesen Dienst zu generieren oder nicht. Die entsprechende Abbildung erfolgt, wie bei den Kostenfunktionen, in Abhängigkeit der Lasten aller Dienste und der Zeit. Damit sind neben statischen Nicht-Implementierungen auch Abhängigkeiten zwischen den Diensten möglich. So ist es beispielsweise beim Mobilfunkstandard GPRS nicht möglich, Sprach- und Datendienste zur gleichen Zeit zu nutzen. Wird ein Anruf gestartet oder angenommen, müssen Datendienste angehalten werden. Die Sprachdienste werden dabei priorisiert. Eine Modellierung dieser Abhängigkeit ist mit $\mathcal{G}$ möglich, die Priorisierung kann jedoch nicht beschrieben werden. Sie ergibt sich zwangsläufig durch die Sequentialisierung bei der algorithmischen Lösung, die später noch detailiert beschrieben wird.

Aus den Definitionen der Testszenarien und den Modellen der Last und Einschränkungen der Klienten kann nun eine Bedingung abgeleitet werden, die erfüllt werden muß. Ziel ist es schließlich, daß für jeden Dienst die Summe der tatsächlich generierten Lasten der im Testszenario zur Zeit t geforderten Last entspricht. Aus den Definitionen 4.7 und 5.4 ergibt sich also folgende Bedingung:

$$\forall i \in [1,n] \quad : \quad \left| \sum_{s=1}^{m} (l_{s,i} \cdot g_{s,i}(t,(l_{s,1},...l_{s,n}))) - f_i(t) \right| < \varepsilon_i \quad , \quad \varepsilon_i > 0 \tag{5.6}$$

Die Summe aller Lasten eines Dienstes i darf nur innerhalb einer definierten Schranke ε_i von der geforderten Szenarienlast abweichen. Eine Differenz von 0 ist zwar theoretisch denkbar, praktisch jedoch schwer umsetzbar. Dies ist schon allein durch die Modellierungsvereinfachung bedingt, daß in Definition 4.7 eine Soll-Last den Wertebereich der reelen Zahlen hat, während die Einzellasten $l_{s,i}$ nach Gleichung 4.6 Element der natürlichen Zahlen sind.

Die Lastgrenzen für einzelne Klienten, die in Gleichung 5.1 beschrieben sind, gelten auch in diesem Fall. Gemeinsam mit Gleichung 5.6 wird aber nur das notwendige Kriterium beschrieben, um korrekte Tests in Bezug auf die Testszenarien zu erzeugen und dabei die Rahmenbedingungen der Klienten zu berücksichtigen.

Insbesondere bei vielen verfügbaren Klienten existieren in der Regel zahlreiche, mögliche Lösungen, die dieses Kriterium erfüllen. Basierend auf der Kostenmatrix 5.3 kann ein Bewertungskriterium definiert werden, daß es erlaubt, die Güte der Lösung in Bezug auf die definierten Kostenfunktionen zu bewerten. Die Gesamtkosten K einer Verteilung auf die Klienten ergibt sich aus den Kosten der Lasten, die jeder Klient durch Zuweisung umsetzt:

$$K \quad = \quad \sum_{s=1}^{m} \sum_{i=1}^{n} \left(c_{s,i}\left(t,(l_{s,1},...l_{s,n})\right) \cdot g_{s,i}\left(t,(l_{s,1},...l_{s,n})\right) \right) \tag{5.7}$$

5.1.2. Optimierungsproblem

Basierend auf diesem Modell ist es nun Aufgabe eines geeigneten Algorithmus im Testautomatisierungsmodul, für ein gegebenes Testszenario sowie eine Kosten- und Einschränkungsmatrix eine Aufteilung der Lasten $l_{s,i}$ für alle Klienten s und Dienste i zu finden, so daß die Bedingungen 5.1 und 5.6 erfüllt sind, sowie sinnvollerweise die Kosten K minimiert werden:

$$K \overset{!}{=} min \tag{5.8}$$

Diese Definition entspricht einem Optimierungsproblem. Um dieses weiter zu spezifizieren, sind Überlegungen zur Charakteristik der einzelnen Funktionen notwendig.

Kern des Optimierungsproblems ist die Zielfunktion, gegeben durch 5.8 und 5.7. Die Aufsummierung der Einzelkosten ist linear. Jedoch werden an die Kostenfunktionen c, die in Gleichung 5.2 definiert sind, keinerlei Anforderungen gestellt. Sie müssen weder linear noch stetig sein, sondern werden in vielen praktischen Anwendungen Unstetigkeiten und Nichtlinearitäten aufweisen. In Folge dessen können die Gesamtkosten K nicht als linear in den zu berechnenden Einzellasten l angesehen werden und das Optimierungsproblem ist eines aus der Klasse der *nichtlinearen Programmierung* (Vgl. [30], S. 9 f, [84], S. 12).

Erschwerend kommt hinzu, daß gemäß Gleichung 4.6 die Last nur Werte aus der Menge der natürlichen Zahlen annehmen darf. Dies ist der Interpretierbarkeit von Dienstnutzungsmechanismen geschuldet, erschwert aber das Optmierungsproblem ungemein, da der Lösungsraum für l diskret ist.

Da mittels der Gleichung 5.6 und 5.1 auch Nebenbedingungen gesetzt sind, die den zu optimierenden Lösungsraum einschränken, handelt es sich schlußendlich beim beschriebenen Problem um ein nichtlineares, diskretes Optimierungsproblem unter Nebenbedingungen (Vgl. [39]).

5.2. Lösungsansätze

Das Testautomatisierungsmodul hat die Aufgabe, eben jenes im vorangegangen Abschnitt definierte Problem zur Testlaufzeit zyklisch zu lösen. Hierfür wird ein Algorithmus benötigt, der auf Basis der Eingaben und unter Berücksichtigung der definierten Nebenbedingungen die Lasten der einzelnen, verfügbaren Klienten berechnet.

Über die Beschreibung des eigentlichen Optimierungsproblems müssen hierzu weitere Konventionen über mögliche Ein- und Ausgaben dieses Algorithmus sowie Anforderungen an Genauigkeit und Berechnungszeiten gestellt werden.

Eingaben für den Algorithmus bilden alle modellrelevanten Parameter der zurückliegenden Abschnitte. Sie können unterteilt werden in statische Parameter, die bereits vor dem Test ermittelt bzw. definiert werden können und dynamischen Parametern, die sich während der Laufzeit ergeben.

Statische Eingabeparameter sind die folgenden:

- Anzahl der Klienten und deren Lastgrenzen (Gleichung 5.1)
- Kostenfunktionsmatrix $\mathcal{C}$ (Gleichung 5.3)
- Einschränkungsmatrix $\mathcal{G}$ (Gleichung 5.4)
- Testszenario (Gleichung 4.7)

Einige Rahmenbedingungen können sich im Laufe der Testausführung ändern, so daß die Eingaben an den Algorithmus während der Laufzeit angepaßt werden müssen.

Entsprechende dynamische Eingabeparameter sind:

- Erweiterte Einschränkungsmatrix $\mathcal{G}'$
- Aktuell generierte Lasten $l'_{s,i}$ aller Klinenten s

Die erweiterte Einschränkungsmatrix $\mathcal{G}'$ ergibt sich aus der statischen Einschränkungsmatrix $\mathcal{G}$ und den Laufzeitinformationen über aktuell verfügbare Klienten. Damit ist es möglich, dem Algorithmus Informationen über temporär nicht erreichbare oder nicht nutzbare Klienten zur Verfügung zu stellen.

Etwas aufwändiger sind die Betrachtungen zu den aktuell generierten Lasten. Hierzu ist ein Verständnis des Prozesses notwendig. Zum Start des Tests generiert keiner der Klienten Lasten. Erst durch entsprechende Anforderungen des Testszenarios und erstmalige Ausführung des Testautomatisierungsalgorithmus werden Lasten für einzelne Klienten errechnet und die jeweiligen Klienten in entsprechende Zustände gebracht. Nach Ablauf einer bestimmten Zeit kann es passieren, daß ein Klient den ihm zugewiesenen Zustand verläßt und deshalb die ihm zugeteilte Last nicht mehr generiert. In Folge dessen ist eine Neuberechnung notwendig. Allerdings haben sich die Vorbedingungen geändert, da nun nicht alle Klienten im Ruhezustand sind. Sie produzieren unter Umständen noch Lasten für verschiedene Dienste. Demzufolge sucht der Algorithmus keine absolute Last mehr sondern muß eine Änderung der aktuellen Last in Relation zum Testszenario berechnen. Dieser Umstand muß dem Lösungsalgorithmus in Form der aktuell generierten Lasten mitgeteilt werden.

Hierfür wird die absolute Last $l_{s,i}$, wie sie für verschiedene Definitionen bisher verwendet wurde, aufgesplittet und mit einer Zeitmarkierung versehen:

$$\begin{aligned} {}^{t+1}l_{s,i} = {}^{t}l'_{s,i} + {}^{t}l^{+}_{s,i}, \qquad {}^{t}l'_{s,i} &\in \mathbb{N} \\ {}^{t}l^{+}_{s,i} &\in \mathbb{Z} \\ {}^{t+1}l_{s,i} &\geq 0 \end{aligned} \tag{5.9}$$

Während ${}^{t}l'_{s,i}$ die aktuelle Last repräsentiert, die ein Klient s am Dienst i zur Zeit t erzeugt, gibt ${}^{t}l^{+}_{s,i}$ eine relative Änderung an, die im Zeitpunkt t zur Erfüllung der Testszenarien errechnet werden muß. Da diese Änderung auch negativ sein kann, bspw. wenn das Szenario sinkende Lasten fordert, hat es den diskreten Wertebereich der ganzen Zahlen $\mathbb{Z}$. Summiert ergibt die aktuelle Last und die im Zeitpunkt t errechnete, relative Änderung die bisher verwendete Notation der absoluten Last $l_{s,i}$, allerdings nun zum Zeitpunkt $t + 1$.

Diesem Zeitmodell kommt das iterative Berechnungsmodell der Testbench entgegen. Der Algorithmus zur Testparitionierung errechnet im Zeitpunkt t auf Basis der aktuellen Last zum Zeitpunkt t eine Änderung der Last, die nach Übermittlung an die Klienten zum Zeitpunkt $t + 1$

zu einer absoluten Last am SPS führt. Es ist jedoch zu beachten, daß dem Gesamtsystem keine synchrone, gleichverteilte Zeit zu Grunde liegt. Die Iteration $t + 1$ ist nur der Vereinfachung halber gewählt.

Entsprechend müssen auch die Gleichungen zur Problemdefinition angepaßt werden. Gesucht wird nun nicht mehr die absolute Last l sondern die relative Last l^+ in Abhängigkeit der aktuellen Last l'.

$$\begin{aligned} {}^tK = \sum_{s=1}^{m} \sum_{i=1}^{n} \Bigg(\quad & c_{s,i}\Big(t, (\,{}^tl'_{s,1} + \,{}^tl^+_{s,1}, \dots \,{}^tl'_{s,n} + \,{}^tl^+_{s,n})\Big) \cdot \\ & g_{s,i}\Big(t, (\,{}^tl'_{s,1} + \,{}^tl^+_{s,1}, \dots \,{}^tl'_{s,n} + \,{}^tl^+_{s,n})\Big)\Bigg) \qquad (5.10) \\ {}^tK \overset{!}{=} min \end{aligned}$$

Auch die Bedingungen 5.1 und 5.6 ergeben sich entsprechend der Substitution.

$$\forall s \in S, i \in [1, n] \quad : \quad 0 \leqq \; ({}^tl'_{s,n} + \,{}^tl^+_{s,n}) \leqq a_{s,i} \qquad (5.11)$$

$$\begin{aligned} \forall i \in [1, n] \quad : \quad & \left| \sum_{s=1}^{m} \Big({}^tl'_{s,i} + \Big({}^tl^+_{s,i} \cdot g_{s,i}(t, (\,{}^tl'_{s,1} + \,{}^tl^+_{s,1}, \dots \,{}^tl'_{s,n} + \,{}^tl^+_{s,n}))\Big)\Big) - f_i(t) \right| \\ & < \varepsilon_i \quad , \; \varepsilon_i > 0 \qquad (5.12) \end{aligned}$$

Basierend auf dieser Erweiterung wird also ein Algorithmus benötigt, der das in Gleichung 5.10 beschriebene Optimierungsproblem löst. Gesucht wird dabei die Belegung der relativen Lasten ${}^tl^+_{s,i}$ für alle Klienten i und alle Dienste s zum Algorithmenausführungszeitpunkt t, die die genannten Bedingungen erfüllen und minimale oder zumindest minimierte Kosten verursachen. Die Zeit t gilt während der Berechnung der Lösung als konstant.

Neben den formalen Parametern gibt es weitere informelle Anforderungen, insbesondere die Berechnungszeit in Abhängigkeit des Aufrufparadigmas und der Anzahl der Klienten sowie der Anzahl der Dienste.

Der Algorithmus muß immer dann zur Neuberechnung der relativen Lasten aufgerufen werden, wenn sich die aktuelle Last oder die geforderten Lasten im Testszenario ändern. Da laut Problembeschreibung die Kosten für jeden Dienst auch in Abhängigkeit der absoluten Last anderer Dienste gebildet werden können, ist prinzipiell eine komplette Neuberechnung des Problems notwendig.

Ob die Neuberechnung in Folge der entstandenen Differenz zwischen Soll- und Ist-Last sofort, also asynchron mit dem Eintreten des Ereignisses, oder synchron zu einer Aktualisierungsperiode gestartet wird, hängt dabei von den Rahmenbedingungen des Tests und auch der Komplexität des Lösungsalgorithmus ab.

Intuitiv ergibt sich die Komplexität des beschriebenen Problems aus der Anzahl der Klienten m und der Anzahl der Dienste n. Ist m groß - im praktischen Fall kann von bis zu einigen tausend Klienten ausgegangen werden - ergeben sich demzufolge verhältnismäßig lange Rechenzeiten für den Algorithmus. Diese führen mit weiteren Parametern zu einer entsprechenden Verzögerung in der Lastanpassung. Wie groß diese Verzögerung sein sollte oder im ungünstigsten Fall sein darf, hängt stark vom zu testenden System und den daraus abgeleiteten Rahmenparametern ab. Eine absolute Betrachtung ist folglich nicht sinnvoll.

Allerdings ist ebenfalls intuitiv anzunehmen, daß das beschriebene Optimierungsproblem schwierig zu lösen ist. Es ist zu befürchten, daß insbesondere die erlaubte Nichtlinearität und Unstetigkeit der Zielfunktion, die sich aus den Eigenschaften der Kostenfunktionen in der Kostenfunktionsmatrix $\mathcal{C}$ ergibt, eine globaloptimale Lösung ohne vollständige Suche im kompletten Lösungsraum unmöglich macht.

In den folgenden Abschnitten werden verschiedene Ansätze zur Berechnung der relativen Lasten diskutiert.

5.2.1. Bekannte Probleme und Algorithmen

Der obige Ansatz, das zu lösende Problem als eine Optimierungsaufgabe zu beschreiben, ergab sich aus den zugehörigen Formalismen und deren Abhängigkeiten. Wünschenswert wäre es jedoch, die Aufgabe auf bekannte Probleme der Informatik, wie sie im Abschnitt 3.5 vorgestellt wurden, abzubilden und somit auch bewährte Lösungsansätze nutzen zu können.

Hierzu ist eine Betrachtung der Natur des zu lösenden Problems hilfreich. Der Vereinfachung wegen wird dabei nur von einem zu testenden Dienst ausgegangen. Gegeben ist eine Sollauslastung, die in Form eines Testszenarios zeitlich abhängig quantifiziert wird. Zur Aufteilung dieser Last auf die vorhandenen Klienten sind nun zwei Aufgaben zu lösen. Zum einen muß das Gesamtproblem „Sollauslastung“ in Teilprobleme „Klientenlasten“ zerlegt und zum anderen diese Teilprobleme den Klienten zugeordnet werden. Für beide Probleme müssen die Randbedingungen, insbesondere die maximale Last, berücksichtigt und die Kosten der resultierenden Lösung möglichst minimiert werden.

Werden diese beiden Aufgaben getrennt voneinander betrachtet, fallen die Ähnlichkeiten zu Standardproblemen der Informatik auf, wie sie im Abschnitt 3.5 erläutert wurden. Die Zerlegung des Gesamtproblems erinnert stark an das Rucksackproblem - oder vielmehr ein inverses Rucksackproblem, wo der Rucksack möglichst günstig ausgepackt werden soll. Dabei ist die „Größe“ der zu entnehmenden Teile nicht fixiert sondern kann frei definiert werden. Allerdings müssen die Mengen gemäß Problemdefinition ganzzahlig sein, so daß das Problem weiterhin diskret ist. Bezüglich der definierten Kosten ist im Gegensatz zum Rucksackproblem, wo eine Maximierung angestrebt wird, eine Minimierung gefordert. Die Überführung eines Maximierungs- in ein Minimierungsproblem wäre jedoch durch einen Vorzeichenwechsel möglich.

Die Zuordnung zu den Klienten wiederum ist dem Partitionierungsproblem oder dem Problem *MULTIPROCESSOR SCHEDULING* ähnlich, wie sie im Kapitel über den Stand der Technik vorgestellt wurden (siehe Abschnitte 3.5.1 und 3.5.1). Es gilt, die Teilprobleme aus „dem Rucksack“ an geeignete Klienten zu verteilen, so daß diese die Lasten generieren können und die Gesamtlösung nach der Kostenfunktionsmatrix möglichst günstig ist.

Allerdings sind beide Aufgaben im Falle der Testpartitionierung nicht unabhängig voneinander. Die Kosten sind nicht allein von der Größe des Teilproblems - also der absoluten Last l - abhängig, sondern auch vom Klienten, denen sie zugeordnet wird. Demzufolge besteht eine Wechselwirkung zwischen den Ergebnissen der Zerlegung und der Zuordnung. Um die beiden Aufgaben optimal oder zumindest gut lösen zu können, wird also ein Algorithmus benötigt, der den durch beide Probleme gemeinsam aufgespannten Lösungsraum betrachtet und bewertet.

5.2.2. Optimierungsproblem

Da das Problem bereits als Optimierungsaufgabe beschrieben ist, liegt es nahe, es auch mittels verfügbarer Solver zu lösen. Wegen der großen Zahl verfügbarer Solver bietet sich hierfür das Programm Matlab der Firma Mathworks an ([8], [78]).

Die Optimization Toolbox von Matlab bietet mit fmincon einen Solver zur Lösung nichtlinearer Optimierungsprobleme unter Nebenbedingungen. Ein solches Problem wird dabei wie folgt beschrieben (Vgl. [8] S. 247f):

$$\begin{aligned}
\underset{x}{min} f(x) & \quad unter\ den\ Nebenbedingungen && (5.13)\\
c(x) &\leq 0 && (5.14)\\
c_{eq}(x) &= 0 && (5.15)\\
A \cdot x &\leq b && (5.16)\\
A_{eq} \cdot x &= b_{eq} && (5.17)\\
lb \leq \ x &\leq ub && (5.18)
\end{aligned}$$

Ziel ist es, eine Belegung für den Vektor x zu finden, für den die Funktion $f(x)$ minimal ist und der die Nebenbedingungen erfüllt. c und c_{eq} sind nichtlineare Funktionen. A und A_{eq} sind wiederum konstante Matrizen und b, b_{eq} sowie lb und ub konstante Vektoren.

Zur Nutzung von fmincon müssen folglich die Gleichungen 5.10 bis 5.12 auf die genannten Formalismen abgebildet werden. Da es sich bei x um einen zu minimierenden Vektor handelt, muß die bisher übliche zweidimensionale Betrachtung einer Last $l_{s,i}$ für den Klienten s und den Dienst i sowie die Betrachtungen zur Kosten- und Einschränkungsmatrix auf Vektoren umgebrochen werden.

Dies ist ohne Einschränkung der Anforderungen möglich, da die $n \times m$ Matrizen auf Vektoren der Länge $n \cdot m$ abgebildet werden können. Die gesuchten Laständerungen ${}^{t}l^{+}_{s,i}$ ergeben sich zum Matlabvektor x wie folgt:

$$\begin{pmatrix} {}^{t}l^{+}_{1,1} & \cdots & {}^{t}l^{+}_{1,n} \\ \vdots & \ddots & \vdots \\ {}^{t}l^{+}_{m,1} & \cdots & {}^{t}l^{+}_{m,n} \end{pmatrix} \rightarrow \begin{pmatrix} {}^{t}l^{+}_{1,1} \\ \vdots \\ {}^{t}l^{+}_{1,n} \\ {}^{t}l^{+}_{2,1} \\ \vdots \\ {}^{t}l^{+}_{m,n} \end{pmatrix} \tag{5.19}$$

Die Abbildung der Kostenmatrix erfolgt identisch.

Zur Formulierung der Gültigkeitsgrenzen der Lasten stehen zwei Möglichkeiten zur Verfügung. Zum einen können die Unter- und Obergrenzen[1] (siehe Gleichung 5.18) für den zu berechnenden Vektor x angegeben werden. Die Maximallast $a_{s,i}$, die ein Klient s für den Dienst i erzeugen kann, muß demzufolge an die entsprechende Position des Vektors ub eingetragen werden, wie es in Gleichung 5.19 für die Lasten definiert wurde.

Gleiches gilt für die untere Grenze. Da immer eine relative Laständerung ${}^{t}l^{+}_{s,i}$ gesucht ist, kann diese als untere Grenze immer den negierten Wert der aktuellen Last ${}^{t}l'_{s,i}$ annehmen. Gemäß Gleichung 5.9 ergäbe damit eine minimale Laständerung eine neue absolute Last von null.

Alternativ kann zumindest die maximale Last auch über die Ungleichung 5.16 beschrieben werden. Die maximale Last wird im Vektor b definiert, während A als Einheitsmatrix erstellt werden muß.

Der Solver fmincom schließt Lösungen außerhalb dieser Grenzen lb, ub oder b aus. Erste Tests zeigten variierende Ergebnisse zwischen den beiden genannten Modellierungen für die maximalen Lasten. Aufgrund deutlicher Unterschiede der Berechnungszeiten wird die Modellierung über die erstgenannte Variante der Vektoren lb und ub gewählt.

Wesentlich schwieriger gestaltet sich jedoch die Formulierung der Gleichung 5.12, die die Annäherung von Testsoll an das Ist beschreibt. Aufgrund der semantischen Unabhängigkeiten der beiden Dimensionen Klienteninstanz und Dienstinstanz kann keine algebraische Basisoperation gefunden werden, die bspw. unter Nutzung der Gleichungen 5.16 oder 5.17 die Gleichung 5.12 beschreiben. Es kann jedoch die Möglichkeit zur Ablaufsteuerung in Matlab genutzt werden (Vgl. [8], S. 17ff). Diese gestattet die Nutzung der aus üblichen Programmiersprachen bekannten Kontrollstrukturen und damit auch die Generierung von Schleifen zur Berechnung der Summen.

Durch die Einbindung der Einschränkungsfunktionen g läßt sich diese Nebenbedingung nicht mehr unbedingt als lineare Funktion darstellen. Für eben diese Sachverhalte bietet fmincon die vektorwertigen Funktionen c und ceq. Diese müssen syntaktisch in eine Funktion gekapselt

---

[1] Englisch: lower bound (lb), upper bound (ub)

werden, in der entsprechende Werte der Rückgabevektoren errechnet werden. Für die Berechnung der Soll-Ist-Annäherung empfiehlt sich die Verwendung der Funktion c_{eq}.

Eine Implementierung dieser Funktion in Matlab ist im Listing 5.1 angegeben. Die Funktion zur Ungleichheitsprüfung c wird nicht benötigt und folglich der Vektor der Rückgabewerte leer generiert. Die Differenz zwischen der Summe aller Laständerungen auf allen Klienten (angegeben im Vektor x) und der zu berechnenden Laständerung (angegeben im Vektor lt) wird für jeden Dienst berechnet und im Ergebnisvektor c_{eq} gespeichert. Wie in Gleichung 5.12 definiert, erfolgt dabei eine Normierung, ob die Differenz innerhalb eines definierten Bereichs, hinterlegt im Vektor $EPSILON$, liegt und damit die Last als erreicht gilt.

Die gegebene Funktion $cond$ ist durch die Berücksichtigung der Eingabevektorgrößen für alle im Abschnitt 5.1 beschriebenen Probleme nutzbar. Zur Vereinfachung werden keine Einschränkungsfunktionen g verwendet. Vielmehr existiert ein Vektor g in der gleichen Dimension wie x, der die Werte $\{0, 1\}$ annehmen kann und damit eine statische Modellerierung der Verfügbarkeit erlaubt. Die Integration von Einschränkungsfunktionen ist jedoch analog zu den Kostenfunktionen in Listing 5.2 problemlos möglich.

```
function [c, ceq] = cond(x, lt, g)
global EPSILON

c    = [];

s =      zeros(1, numel(lt));
ceq =    zeros(1, numel(lt));

a =      numel(x) / numel(lt);

for i=1:a
    for j=1:numel(lt)
       s(j) = s(j) + ( (x(((i - 1) * numel(lt)) + j)) * (g(((i - 1) * numel(lt)) + j)) );
    end
end

for j=1:numel(lt)
    if abs(s(j) - lt(j)) < EPSILON(j),
        ceq(j) = 0;
    else
        ceq(j) = s(j) - lt(j);
    end
end
```

Listing 5.1: Definition der Nebenbedingungsfunktionen c und ceq

Die Kostenfunktionen $c_{s,i}$ der einzelnen Klienten s für die Dienste i werden ebenso in eigenen Funktionen $c\_s\_i(l1, ..., ln)$ definiert. Diese wird immer in Abhängigkeit der absoluten Lasten l aller Dienste berechnet. Ein Beispiel für eine solche Kostenfunktion ist in Listing 5.2 zu finden. Mit dem zugehörigen Aufruf durch entsprechende Parameter entspricht die beschriebene Funktion folgendem Sachverhalt:

$$c_{4,0} \quad = \begin{cases} 0 & {}^{t}l_{4,0} = 0 \\ 400 & {}^{t}l_{4,0} < 20 \\ 1000 & {}^{t}l_{4,0} < 40 \\ 500 & tl_{4,0} \geq 40 \end{cases} \tag{5.20}$$

Die Berechnung der Kosten für den Dienst 0 erfolgt in diesem Falle also ohne Berücksichtigung der Lasten anderer Dienste, die auf dem Klienten 4 umgesetzt werden.

```
function y = c_4_0(l1, l2, l3)

if l1 == 0,
    y = 0;
else
    if l1 < 20,
        y = 400;
    else
        if l1 < 40,
            y = 1000;
        else
            y = 500;
        end
    end
end
```

Listing 5.2: Definition einer Kostenfunktion für ein Problem mit drei Diensten

Die zu optimierende Zielfunktion, wie sie in Gleichung 5.13 beschrieben ist, muß ebenfalls in einer eigenen Funktion modelliert werden und wird in Listing 5.3 angegeben. Hier ist die Summe entsprechend aufzulösen. Wie im Listing 5.1 wird die Menge der Einschränkungsfunktionen zu einem Vektor vereinfacht.

```
function y = objfunction(x, l0, lt, g)
y =    c0_0( ( x(1) + l0(1) ), ( x(2) + l0(2) ) ) * g(1)
    + c0_1( ( x(1) + l0(1) ), ( x(2) + l0(2) ) ) * g(2)
    + c1_0( ( x(3) + l0(3) ), ( x(4) + l0(4) ) ) * g(3)
    + c1_1( ( x(3) + l0(3) ), ( x(4) + l0(4) ) ) * g(4);
```

Listing 5.3: Definition der Zielfunktion

Mit den aufgeführten Listings ist das Optimierungsproblem für Matlab beschrieben. Der Solver fmincon bietet zahlreiche Varianten zum parametrisierten Start. Die umfänglichste, unter Verwendung aller definierten Bedingungen, ist im folgenden Listing gegeben.

```
[x,fval,exitflag] =
    fmincon ( fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,
                (f1,f2,...) )
```

Listing 5.4: Übergabe- und Rückgabeparameter des fmincon-Solvers

Die einzelnen Parameter haben folgende Bedeutung:

- fun - die zu minimierende Funktion, die einen Vektor x als Eingabe akzeptiert und den Funktionswert f an der Stelle x zurückgibt. Üblicherweise wird fun als Handle auf eine in eine Datei ausgelagerte Funktion angegeben.
- x0 - gibt den Startwert für den zu optimierenden Vektor x an
- A, b, Aeq, beq - die linearen Nebenbedingungen gemäß Gleichungen 5.16 und 5.17
- nonlcon - die Funktion zur Berechnung der nichtlinearen Nebenbedingungen, wie in Gleichung 5.1 angegeben. Üblicherweise wird auch $nonlcon$ als Handle auf eine in eine Datei ausgelagerte Funktion angegeben.
- options - eine Datenstruktur zur Definition verschiedener Kontrollparameter des Solvers
- fx - Menge weiterer Vektoren, die an alle Funktionen als Parameter weitergegeben werden
- x - Ergebnisvektor der Minimierung
- fval - Wert der Zielfunktion an der berechneten Stelle x
- exitflag - Rückgabewert des Algorithmus' zur Definition der Terminierungsursache

Für die im Vorlauf genannten Umsetzungen des Optimierungsproblems ergibt sich ein Aufruf des Solvers wie in Listing 5.5 angegeben. Zur Definition der Algorithmenparameter wird eine Struktur vom Typ optimset angelegt und entsprechend modifiziert. Neben der Angabe von Iterationsbeschränkungen ist insbesondere die Auswahl des Lösungsalgorithmus' entscheidend.

Der Solver fmincom unterstützt vier Lösungsalgorithmen (Vgl. [4], S. 2-7f.):

- trust-region-reflective
- interior-point
- active-set
- sqp

Den Vorschlägen der Toolbox-Dokumentation ([4]) zur Auswahl des geeigneten Lösungsalgorithmus folgend wurde nach einigen Tests die „Interior-Point“-Implementierung gewählt (Vgl. [18]). Alle anderen Algorithmen fanden selbst in relativ einfachen Probleminstanzen keine Lösungen bzw. terminierten nicht in vertretbarer Zeit.

```
options=optimset;
options=optimset(options,'Display', 'off');
options=optimset(options,'Algorithm','interior-point');
options=optimset(options,'MaxIter', 10000);
options=optimset(options,'MaxFunEvals', 1000000000);
options=optimset(options,'TolX', 1E-50);
options=optimset(options,'TolCon', 1E+00);

[x,fval,exitflag,output,lambda,grad,hessian] = ...
fmincon(@objfunction,x0,[],[],[],[],lb,lu,@neben,options,
    10,lt,g);
```

Listing 5.5: Aufruf des fmincon-Solvers

Der Aufruf der Optimierung wurde wiederum in eine Funktion gekapselt, die mit den entsprechenden aktuellen Parametern gerufen werden kann.

Bei dieser Matlab-basierten Lösung handelt es sich um einen problemunabhängigen Ansatz, der erwartungsgemäß mit verhältnismäßig hohen Rechenzeiten aufwarten wird. Zur Lösung des Optimierungsproblems ist deshalb ein weiterer, problemspezifischer Ansatz erarbeitet worden, der im folgenden Abschnitt vorgestellt wird.

Qualitative und quantitative Bewertungen und Ergebnisse von Matlab und dem problemspezifischen Ansatz werden im Anschluß in Abschnitt 5.3 diskutiert.

5.2.3. Heuristik

Die Verwendung des Matlab-Solvers ist mit einigen Nachteilen verbunden. Neben Lizenzierungs- und Kostenfragen ist vor allem die Integration in die Testbench schwierig.

Ein alternativer Ansatz zur Lösung des in Abschnitt 5.1 definierten Problems ist eine Heuristik. Hierunter wird ein Verfahren verstanden, das neben wissenschaftlich gesicherten Erkenntnissen auch auf „Hypothesen, Analogien und Erfahrungen“ aufbaut (Vgl. [23], S. 296). Ziel ist es, in kurzer Zeit relativ gute Lösungen für das Problem zu finden.

Im Falle heuristischer Algorithmen werden häufig problemspezifische Besonderheiten genutzt, um schnelle und brauchbare Lösungsverfahren zu definieren. Demzufolge ist es zunächst notwendig, vereinfachende Annahmen vom allgemeinen, im Abschnitt 5.2 formulierten Optimierungsproblem zu finden.

5.2.3.1. Problemspezifische Annahmen

Ziel ist es, im beschriebenen Problem Einschränkungen oder Vereinfachungen zu finden, die eine schnellere Lösung ermöglichen. Da der Testpartitionierungsalgorithmus immanenter Bestandteil des Automatisierungsmoduls ist, sollte er für jedes denkbare Systemtestszenario verwendbar sein. Eine Anpassung dessen an ein konkretes SUT ist somit nicht sinnvoll. Vielmehr sollen Eigenarten der dynamischen Testparitionierung herausgestellt werden, die für alle denkbaren Testszenarien gelten oder deren Einführung nur zu sehr geringen praktischen Einschränkungen führen.

Ein erster Ansatzpunkt liegt in der Auflösung der Zweidimensionalität des Problems. Im beschriebenen Optimierungsproblem ist es das Ziel, eine optimale Aufteilung der Gesamtlast auf die Klienten und deren jeweilige Dienstimplementierungen zu finden. Entsprechend ergibt sich die Zahl der zu optimierenden Variablen aus dem Produkt von Klientenanzahl und Anzahl der Dienste. Zum Finden einer optimalen Lösung ist dieser Ansatz zwingend notwendig, da nach Definition auch dienstübergreifende Last- und damit Kostenabhängigkeiten bestehen können (gegeben in Gleichung 5.2).

In den meisten praktischen Fällen kann jedoch davon ausgegangen werden, daß der Einfluß zwischen den einzelnen Diensten eher gering ausfällt. Häufig resultiert er vor allem aus Betriebsmittelkonkurrenzen der einzelnen Dienstimplementierungen auf den Klienten. Ein Großteil der Klienten wird die Dienstnutzung implizit oder explizit priorisieren. Dies kann durch technologische Parameter oder auch durch Bedienkonzepte o. ä. realisiert sein.

In Folge dieser Eigenschaft ergibt sich die Möglichkeit, das Optimierungsproblem in eine Sequenz von Optimierungsproblemen für die einzelnen Dienste aufzulösen. Der Dienst i wird dabei festgeschrieben und die zu berechnende Laständerung auf die verfügbaren Klienten möglichst optimal verteilt. Nach der Terminierung dieses Schrittes wird die Laständerung für den Dienst i auf allen Klienten festgeschrieben und der nächste Dienst $i + 1$ kann berechnet werden. Auf diese Weise wird der Suchraum deutlich eingeschränkt.

Eine zweite Vereinfachung ergibt sich aus der Ganzzahligkeit der zu berechnenden Last bzw. Laständerungen, wie sie in den Gleichungen 4.6 und 5.9 definiert wurden. Dadurch kann davon ausgegangen werden, daß jede zu partitionierende Laständerung als Summe ganzzahliger Teillasten betrachtet werden kann, die möglichst optimal auf die verfügbaren Klienten verteilt werden müssen. Damit wird die Lösung zu einem kombinatorischen Problem, das bspw. über einen iterativen Ansatz gelöst werden kann.

5.2.3.2. Algorithmus

Auf Basis der beschriebenen Einschränkungen wird ein heuristischer Algorithmus abgeleitet, der das Optimierungsproblem möglichst schnell und mit vertretbaren Ergebnissen löst.

Eingaben für diesen Algorithmus sind die zu berechnenden Laständerungen der einzelnen Dienste. Diese ergeben sich zum einen aus Änderungen der Funktionswerte $f_i(t)$ in den Testszenarien

(Vgl. Gleichung 4.7) und bzw. oder Änderungen in den absoluten Lasten ${}^{t}l_{s,i}$ einzelner Klienten. Ausgabe des Algorithmus' sind die relativen Laständerungen ${}^{t}l^{+}_{s,i}$. Zur Berechnung benötigt der Algorithmus weiterhin Zugriff auf die Kostenmatrix $\mathcal{C}$ sowie die aktuellen Lasten ${}^{t}l'_{s,i}$ zum Zeitpunkt t. Es wird davon ausgegangen, daß diese Informationen global verfügbar sind und nicht als Parameter übergeben werden.

Da der Algorithmus sehr umfänglich ist, wird er im folgenden in Form verschiedener Funktionen, die jeweils Teilberechnungen durchführen, vorgestellt. Zur weiteren Vereinfachung wird die Behandlung von Einschränkungen aus der Matrix $\mathcal{G}$ nicht mehr wie im mathematischen Modell als gesonderter Faktor in den Berechnungen aufgeführt. Eine Integration wie im mathematischen Modell ist jedoch auch für den sequentiellen Algorithmus trivial und bedarf deshalb an dieser Stelle keiner weiteren Betrachtung.

Unter Verwendung der genannten problemspezifischen Vereinfachungen ist der Algorithmus $TestPartitioning$ entstanden, der verschiedene Konzepte von Algorithmen aus dem Abschnitt 3.5 kombiniert. Die Grundstruktur des Partitionierungsalgorithmus ist in Algorithmus 1 dargestellt.

Algorithmus 1 Partitioniere Laständerungen

```
procedure TESTPARTITIONING($offset[], recursionlevel$)
    $recursion \leftarrow false$
    for all Dienste $i$ do
        $(correction[i], average[i]) \leftarrow$ INITIALALLOCATION($offset[i]$)
        if $correction[i] \neq 0$ $AND$ $correction[i] < offset[i]$ then
            $recursion \leftarrow true$
        end if
    end for
    for all Dienste $i$ do
        OPTIMISESERVICE($average[i]$)
    end for
    if $recursion = true$ $AND$ $recursionlevel < MAX\_RECURSIONS$ then
        TESTPARTITIONING($correction, (recursionlevel + 1)$)
    end if
     return
end procedure
```

Nach dem Start wird durch die Funktion $InitialAllocation$ zunächst für jeden Dienst eine initiale Verteilung der zu generierenden Last auf alle Klienten errechnet. Im Anschluß verbessert $OptimiseService$ die Verteilung iterativ für jeden Dienst. Unter bestimmten Umständen kann es vorkommen, daß die Verteilung nicht vollständig gelungen ist, so daß eine Differenz zwischen Soll- und berechneter Ist-Last besteht. In diesem Falle wird der Algorithmus durch Anpassung der entsprechenden Laständerungen rekursiv erneut gestartet.

Die Funktion $InitialAllocation$ ist in Algorithmus 2 aufgeführt. Aufgabe dieser Funktion ist es, für einen Dienst die benötigte Laständerung, die als Eingabeparameter übergeben wird, auf alle Klienten zu verteilen. Hierzu wird mittels einer weiteren Funktion $CalcAverage$ ein einheitlicher Wert $average$ errechnet. Dieser entspricht nicht unbedingt dem arithmetischen Mittel aus Laständerung und Anzahl der Klienten, also $offset/m$. Vielmehr kann als kleinster Wert auch 1 ($offset > 0$) bzw. als größter Wert -1 ($offset < 0$) verwendet werden. Auch Werte größer ($offset > 0$) bzw. kleiner ($offset < 0$) als das arithmetische Mittel sind möglich.

In Abhängigkeit davon, ob $average$ positiv oder negativ ist - dementsprechend also die Gesamtlast des Dienstes zu- oder abnehmen soll - wird er dann auf alle verfügbaren Klienten als initiale Laständerung ${}^t l^+_{s,i}$ gespeichert. Bei der Verteilung von $average$ muß jedoch geprüft werden, ob der Klient eine entsprechende Laständerung überhaupt ausführen kann. Bei positiven Werten darf die maximale Last $a_{s,i}$ nicht überschritten werden, und bei negativen Werten darf die resultierende Gesamtlast des Klienten nicht unter 0 fallen.

Entsprechende Abweichungen werden in einer Variable $compare$ aufaddiert. Übersteigt der Wert von $compare$ die geforderte Lasterhöhung $offset$, wird die Schleife abgebrochen. Die etwaige Differenz zwischen geforderter Lasterhöhung $offset$ und tatsächlich verteilter Last $compare$ wird als Rückgabewert der rufenden Funktion zur Verfügung gestellt.

Die initiale Last wird durch entsprechenden Aufruf in $TestPartitioning$ zunächst für jeden Dienst generiert und den Klienten zugewiesen. Im folgenden Schritt muß die pauschale Zuweisung optimiert werden. Auch dies geschieht einzeln für jeden Dienst durch den Aufruf der Funktion $OptimiseService$, die einen iterativen Optimierungsansatz ähnlich zu Kerninghan-Lin verfolgt und als Algorithmus 3 angeben ist.

Auch $OptimiseService$ unterscheidet die Fälle, in denen die zu berechnende Laständerung positiv oder negativ ist, da sich hieraus unterschiedliche Berechnungen für Maxima und Minima ergeben. Im folgenden wird nur auf positive Laständerungen und damit auch ein positives $average$ aus dem Algorithmus 2 eingegangen.

Der Optimierungsansatz besteht darin, die initial errechneten Laständerungen jeweils anderen Klienten zuzuordnen und zu prüfen, ob dadurch geringere Kosten für die Gesamtlösung entstehen. Da sich die Gesamtkosten nach Gleichung 5.7 aus der Summe aller Kliententeilkosten ergibt, ist es bei einem Klientenpaar (i, j) also das Ziel, folgendes zu prüfen:

$$\begin{aligned} & c_{i,h}(t, (..., {}^t l'_{i,h} + {}^t l^+_{i,h}, ...)) + c_{j,h}(t, (..., {}^t l'_{j,h} + {}^t l^+_{j,h}, ...)) > \\ & c_{i,h}(t, (..., {}^t l'_{i,h}, ...)) + c_{j,h}(t, (..., {}^t l'_{j,h} + {}^t l^+_{j,h} + {}^t l^+_{i,h}, ...)) \end{aligned} \tag{5.21}$$

Es gilt jedoch auch in diesem Fall, daß durch das Verschieben der Last die jeweilige Maximallast des Klienten j (entspricht $a_{j,h}$) und das Minimum der Laständerung am Klienten i (für den Fall ${}^t l^+_{i,h} = 0$) berücksichtigt werden muß und dadurch die zu verschiebende Laständerung kleiner sein kann als das ursprünglich mit $average$ berechnete ${}^t l^+_{i,h}$. Die entsprechende Formulierung fehlt der Übersichtlichkeit wegen in Gleichung 5.21, kann jedoch durch entsprechende Differenzen und Maxima leicht ergänzt werden.

Kann durch die Verlagerung eine Kostenverbesserung erreicht werden, wird das entsprechende Paar (i, j) und die Kostendifferenz zunächst vorgemerkt. Wird beim Prüfen weiterer Klientenpaare

Algorithmus 2 Verteile initiale Laständerungen für Dienst i

```
procedure INITIALALLOCATION(offset)
    $compare \leftarrow 0$
    if $offset > 0$ then
        $average \leftarrow \textsc{CalcAverage}(offset)$
        for all Klienten $s$ do
            $tmp \leftarrow min(average, (a_{s,i} - {}^{t}l_{s,i}'))$
            ${}^{t}l^{+}_{s,i} \leftarrow tmp$
            $compare \leftarrow compare + tmp$
            if $compare \geq offset$ then
                ${}^{t}l^{+}_{s,i} \leftarrow (tmp - (compare - offset))$
                $compare \leftarrow offset$
                Break
            end if
        end for
        $correction \leftarrow offset - compare$
    else if $offset < 0$ then
        $average \leftarrow \textsc{CalcAverage}(offset)$
        for all Klienten $s$ do
            $tmp \leftarrow max(average, -{}^{t}l_{s,i}')$
            ${}^{t}l^{+}_{s,i} \leftarrow tmp$
            $compare \leftarrow compare + tmp$
            if $compare \leq offset$ then
                ${}^{t}l^{+}_{s,i} \leftarrow (tmp - (compare - offset))$
                $compare \leftarrow offset$
                Break
            end if
        end for
        $correction \leftarrow offset - compare$
    end if
     return $correction, average$
end procedure
```

Algorithmus 3 Optimiere Dienst h

```
procedure OPTIMISESERVICE($average$)
    $k = 0, max_i = -1, max_j = -1, cost_{max} = 0, diff_{max} = 0$
    while $k < MAX\_ITERATIONS$ do
        $k = k + 1$
        if $average > 0$ then
            for all pairs $(i, j), i \neq j, i \neq max_j, j \neq max_i$ do
                $diff = min( {}^tl^+_{i,h}, max(average, ((a_{j,h} - ( {}^tl'_{j,h} + {}^tl^+_{j,h})))$
                if $diff > 0$ then
                    Berechne Kostenersparnis $cost_{save}$, wenn $diff$ vom
                     Klienten $i$ zum Klienten $j$ verschoben wird
                    if $cost_{save} > cost_{max}$ then
                        $max_i = i, max_j = j$
                        $cost_{max} = cost_{save}, diff_{max} = diff$
                    end if
                end if
            end for
            if $cost_{max} > 0$ then
                ${}^tl^+_{j,h} = {}^tl^+_{j,h} + diff$
                ${}^tl^+_{i,h} = {}^tl^+_{i,h} - diff$
            else
                $break$
            end if
        else if $average < 0$ then
            for all pairs $(i, j), i \neq j, i \neq max_j, j \neq max_i$ do
                $diff = max( {}^tl^+_{i,h}, max(average, {}^tl^+_{j,h})))$
                if $diff < 0$ then
                    Berechne Kostenersparnis $cost_{save}$, wenn $diff$ vom
                     Klienten $i$ zum Klienten $j$ verschoben wird
                    if $cost_{save} > cost_{max}$ then
                        $max_i = i, max_j = j$
                        $cost_{max} = cost_{save}, diff_{max} = diff$
                    end if
                end if
            end for
            if $cost_{max} > 0$ then
                ${}^tl^+_{j,h} = {}^tl^+_{j,h} + diff$
                ${}^tl^+_{i,h} = {}^tl^+_{i,h} - diff$
            else
                $break$
            end if
        end if
    end while
end procedure
```

eine größere Kosteneinsparung gefunden, wird diese in der Folge verwendet. Innerhalb einer Iteration ist Rückübertragung der Lasten vom Paar (i, j) auf das kombinatorisch mögliche Paar (j, i) ausgeschlossen.

Wurde nach der Prüfung aller Paare (i, j) eine Kostenverbesserung größer als 0 gefunden, wird diese durch entsprechende Zuweisung ausgeführt und ein erneuter, iterativer Durchlauf in gleicher Weise gestartet. Im nächsten Iterationsschritt ist eine Rückübertragung der Laständerung von j nach i implizit ausgeschlossen, da dies zu einer Erhöhung der Kosten führen würde und damit ignoriert wird. Das „Schwingen" einer Laständerung zwischen zwei Klienten in jeder Iteration ist damit ausgeschlossen.

Kann in einem der Durchläufe keine Verbesserung mehr gefunden werden, wird die Optimierung abgebrochen. Gleiches gilt beim Erreichen einer maximalen Anzahl von Iterationen, die in der Konstanten $MAX\_ITERATIONS$ hinterlegt wird. Damit ist das Durchsetzen einer Schranke für die Optimierungsdauer möglich.

Die Optimierung wird für jeden Dienst einzeln gerufen. Bei der Optimierung eines Dienstes h sind die Laständerungen für alle zuvor optimierten Dienste $< h$ bereits fixiert, so daß selbst bei ungünstigen Auswirkungen auf die Gesamtkosten - bedingt durch dienstübergreifende Abhängigkeiten der Kostenfunktionen - eine Änderung der berechneten Dienste nicht möglich ist.

Nach Abschluß der Optimierung muß der Algorithmus $TestPartitioning$ mögliche Abweichungen zwischen dem geforderten $offset$ und der tatsächlich generierten Laständerung behandeln. Diese können bei der Ausführung von $InitialAllocation$ entstehen. Zur Erkennung wird bereits beim Aufruf von $InitialAllocation$ für jeden Dienst geprüft, ob der entsprechende Rückgabewert $correction$ ungleich 0 ist und eine entsprechende Nachbehandlung notwendig ist. Dies ist jedoch nur der Fall, wenn $correction$ kleiner als $offset$ des entsprechenden Dienstes ist. Bei Gleichheit muß davon ausgegangen werden, daß sämtliche Klienten für den Dienst bereits maximal ausgelastet sind und somit das Erreichen von $offset$ für diesen Dienst nicht möglich ist.

Sind Abweichungen gefunden, wird die Funktion $TestPartitioning$ rekursiv erneut aufgerufen. Als zu berechnendes $offset$ dient nun die Abweichung zwischen bisherigen $offset$ und der tatsächlich generierten Laständerung. Die entsprechenden Werte sind im Vektor $correction$ gespeichert, so daß dieser als entsprechender Übergabeparameter verwendet werden kann. Als zusätzlicher Parameter wird die Rekursionstiefe inkrementiert. Eine weitere Rekursion wird nur ausgeführt, wenn die Tiefe kleiner als eine vordefinierte Maximalzahl an Rekursionen $MAX\_RECURSIONS$ ist.

Nach dem Rücksprung aus den Rekursionen sind die Laständerungen berechnet und können vom Testframework entsprechend weiterverarbeitet werden.

5.3. Evaluierung

In den zurückliegenden Abschnitten wurden zwei Ansätze zur Lösung des Testpartionierungsproblems vorgestellt. Sowohl die Lösung des Optimierungsproblems mittels Matlab als auch die Lösung mittels der Heuristik bieten die Möglichkeit, korrekte und brauchbare Lösungen in vertretbarer Zeit zu finden. Deshalb werden beide Verfahren im folgenden evaluiert und damit eine Entscheidung für die letztlich einzusetzende Variante zu ermöglichen.

Im folgenden werden Grenzen und Möglichkeiten der Verfahren zunächst theoretisch analysiert und diskutiert. Es schließt sich eine experimentelle Untersuchung an, bei der verschiedene Testpartitionierungen mittels eines speziell dafür konzipierten Evaluierungssystems ermittelt wurden.

5.3.1. Theoretische Analyse

Der Matlab-Solver $fmincon$ ist ein allgemeiner Lösungsansatz für nichtlineare Optimierungsprobleme. Er zählt zur Gruppe der Barriereverfahren (Vgl. [30], S. 77ff) und basiert auf einer entsprechenden Weiterentwicklung der „Innere-Punkte-Methode“ zur Lösung linearer Optimierungsprobleme (vorgestellt in [18], Vgl. auch [13], S. 249ff). Da es sich bei Matlab um ein kommerzielles Softwarepaket handelt, sind Details zur Umsetzung des Verfahrens nicht veröffentlicht. Zwei prinzipielle Probleme ergeben sich jedoch aus den Anforderungen der Testpartitionierung und der mathematischen Definition des Interior-Point Verfahrens.

Zum einen handelt es sich hierbei nicht um ein Verfahren zur diskreten Optimierung. Folglich sind sowohl während der Berechnung als auch als Ergebnis reellwertige Elemente des Lösungsvektors x möglich (Vgl. Listing 5.5). Gemäß Anforderung und der Interpretationsfähigkeit soll die Laständerung l^+ (Vgl. Gleichung 5.9) für das Testpartitionierungsproblem jedoch ganzzahlig sein.

Zum zweiten erwartet das Interior-Point Verfahren zweifach ableitbare Zielfunktionen als Eingabe (Vgl. [15], S. 156, [90], S. 1). Es ist jedoch elementarer Bestandteil der Testmodellierung, daß die Kostenfunktionen über Unstetigkeitsstellen verfügen können (Vgl. Abschnitt 5.1.2). Da die Zielfunktion die Summe aller Kostenfunktionen ermittelt, würde auch diese über entsprechende nicht differenzierbare Stellen verfügen. Entsprechend kommt es vor, daß der Solver ungünstige Optimierungsräume analysiert.

Das erst genannte Problem kann durch entsprechendes Runden der berechneten Funktionswerte x in allen Berechnungen, insbesondere der Nebenbedingungen und der Zielfunktion, behoben werden. Allerdings führt das genau dazu, daß bspw. bei der Berechnung der Zielfunktion durch das Auf- bzw. Abrunden weitere nicht differenzierbare Stellen eingeführt werden. Damit wird durch die Beseitigung des erstgenannten Problems das zweite Problem verschärft. Tests im Laufe der Evaluierung verdeutlichen, daß eine Rundung der Funktionswerte nicht zu brauchbaren Ergebnissen führt. Erschwerend kommt hinzu, daß der Solver häufig aufgrund mangelnder Änderungen der Zielfunktion bei verhältnismäßig stark variierenden Laständerungen abbricht.

Da dennoch für die Testpartitionierung nur ganzzahlige Lasten gültig sind, wurde für das Evaluierungssystem ein Algorithmus zur Rundung und Anpassung der Optimierungsergebnisse aus Matlab implementiert. Das Verfahren ist im Anhang B kurz vorgestellt.

Die Heuristik hingegen ist in ihrer Konzeption von Beginn an auf die Generierung diskreter Werte ausgelegt. Da hier der vollständige Algorithmus bekannt ist, ist die Analyse der zu erwartenden Rechenkomplexität interessant. Aus der Analyse der gesamten Prozedur $TestPartitioning$, gegeben in Algorithmus 1, ergibt sich, daß die Laufzeit in erster Linie von der Anzahl der Rekursionen abhängt, da diese zu einer wiederholten, vollständigen Ausführung der Prozedur führt. In jedem Aufruf wird jeweils für alle Dienste (die Anzahl entspricht n) zunächst $InitialAllocation$ und im Anschluß $OptimiseService$ gerufen. Demzufolge gilt zunächst:

$$\begin{aligned} LZ_{TestPartitioning} &= \#Rekursionen \cdot \\ &\quad (n \cdot LZ_{InitialAllocation} + n \cdot LZ_{OptimiseService}) \end{aligned} \tag{5.22}$$

Die Ausführung von $InitialAllocation$ (Vgl. Algorithmus 2) wird dominiert von der einmaligen Ausführung von $CalcAverage$ und der Zuteilung zu allen Klienten (die Anzahl entspricht m). Entsprechend ergibt sich hierfür:

$$LZ_{InitialAllocation} = LZ_{CalcAverage} + m \tag{5.23}$$

Ausschlaggebend ist jedoch die Laufzeit der Optimierungsfunktion $OptimiseService$, die in Algorithmus 3 gegeben ist. Diese optimiert zwischen Klientenpaaren i, j, so daß die entsprechenden Schleifen eine Laufzeit von m^2 zur Folge haben. Je nach Optimierungsspielräumen wird diese paarweise Optimierung entsprechend iteriert. Folglich ergibt sich die Laufzeit von $OptimiseService$ zu:

$$LZ_{OptimiseService} = \#Iterationen \cdot m^2 \tag{5.24}$$

Weder in $OptimiseService$ noch in $InitialAllocation$ spielt die Alternative, ob $average$ bzw. $offset$ positiv oder negativ sind, eine entscheidende Rolle für die Laufzeit.

Da sowohl die Zahl der Iterationen in $OptimiseService$ wie auch die Zahl der Rekursionen von $TestPartitioning$ mittels Konstanten begrenzt sind, ergibt sich aus den Gleichungen 5.22 bis 5.24 eine Abschätzung der maximalen Laufzeit wie in folgender Gleichung angegeben:

$$\begin{aligned} LZ_{TestPartitioning} \leq\ & MAX\_RECURSIONS \cdot \\ & \Big(n \cdot \big((LZ_{CalcAverage} + m) + (MAX\_ITERATIONS \cdot m^2)\big)\Big) \end{aligned} \tag{5.25}$$

In allen im Laufe der Arbeit betrachteten Fälle ist die Berechnung von $CalcAverage$ in konstanter Laufzeit möglich. Da desweiteren $MAX\_RECURSION$ und $MAX\_ITERATION$ konstant sind, ergibt sich die Laufzeit des Algorithmus zur Testpartitionierung in $O - Notation$ zu:

$$LZ_{TestPartitioning} = O(m^2 \cdot n) \tag{5.26}$$

Folglich kann die Berechnung in polynomieller Laufzeit erfolgen. Die quadratische Abhängigkeit von der Anzahl der Klienten ist der paarweisen Optimierung geschuldet. Durch die konstanten Rekursions- und Iterationsgrenzen ist auch die maximale Laufzeit leicht justierbar, so daß die Terminierung der Berechnung und damit die Bereitstellung der Ergebnisse in bestimmten Zeitschranken möglich ist.

Unabhängig von der Betrachtung der maximalen Laufzeit wäre eine Abschätzung der zu erwartenden Iterationen und Rekursionen interessant. Die Zahl der Iterationen hängt in höchstem Maße von der Art der Kostenfunktionen und den zugewiesenen Lasten ab. Eine allgemeine Aussage über die zu erwartende Anzahl ist somit nicht möglich.

Anders ist das bei der Zahl der Rekursionen. Diese hängen in erster Linie von $average$, das in $InitialAllocation$ berechnet wird, ab. Wird dieser Wert seinem Namen als das arithmetische Mittel aus der zu verteilenden Laständerung und der Zahl der verfügbaren Klienten gerecht und ist desweiteren jeder der Klienten in der Lage wenigstens eine Laständerung von $average$ aufzunehmen, ist kein rekursiver Aufruf der $TestOptimisation$ notwendig. Ist $average$ kleiner als das arithmetische Mittel bzw. können Laständerungen wegen der Nebenbedingungen nicht zugewiesen werden, müssen entsprechende Rekursionen zur Korrektur ausgeführt werden. Gleiches trifft zwangsläufig zu, wenn das Verhältnis aus Laständerung und Klientenzahl nicht ganzzahlig ist.

Neben dieser quantitativen Betrachtung der Laufzeit spielen auch qualitative Betrachtungen der Ergebnisse eine entscheidende Rolle. Prinzipiell gilt, daß die Heuristik kein vollständiges Optimierungsverfahren bildet. Durch Einschränkungen der Größe des Lösungsraumes und vereinfachende Annahmen beim Finden lokaler Optima steht eine überschaubare und begrenzte Ausführungszeit im Vordergrund.

Die gröbste Vereinfachung findet sich in der Optimierung der einzelnen Dienste ($OptimiseService$). Eine Optimierung erfolgt durch iteriertes, paarweises Verschieben diskreter Laständerungen. Die Größe der Laständerungen variiert, besteht aber initial aus dem in

$InitialAllocation$ berechneten Wert $average$. Demzufolge hat dieser Wert nicht nur einen großen Einfluß auf die Anzahl der Rekursionen und damit die Laufzeit, sondern auch auf die Qualität der Ergebnisse. Ist $average$ sehr groß, werden u. U. kleine lokale Minima beim paarweisen verschieben der Last übersprungen und damit nicht gefunden. Der genaue Einfluß von $average$ variiert jedoch auch in Abhängigkeit von der Art der Kostenfunktionen. Einige exemplarische Messungen hierzu sind im nächsten Abschnitt zu finden.

Ein nicht so offensichtlicher aber dennoch wichtiger Qualitätsfaktor ist die Art der Paarbildung bei der Optimierung. Dies ist besonders dann relevant, wenn die Klienten in Gruppen mit gleichen Kostenfunktionen gefaßt werden. Dies ist durchaus praxisrelevant, da sich Klienten meist auch in Kategorien, wie bspw. verschiedene Mobiltelefonmodelle, einteilen lassen. Gleiche Geräte mit gleichem Nutzerverhalten sollten folglich auch durch die gleichen Kostenfunktionen modelliert werden.

Wird die Paarbildung klassisch über zwei Schleifen realisiert, die jeweils bei Klient 1 beginnen, werden die Laständerungen auf die jeweils ersten Klienten einer Gruppe verschoben, so daß eine sehr einseitige Nutzung der Klienten entsteht. In der Umsetzung des Prozedur $OptimiseService$ wird das erste zu prüfende Paar (i, j) zufällig aus der Menge der Klienten gewählt und im Anschluß sequentiell durch Inkrementierung der Klientenindizes fortgefahren. Auf diese Weise kann die Iterationssymmetrie mit geringem Aufwand vermieden werden.

Ein weiterer Ansatz zur Vereinfachung des Problems bildet die sequentialisierte Berechnung der einzelnen Dienste. Die Laständerung wird für jeden Dienst einzeln auf die Klienten verteilt und optimiert. Durch die getrennte Betrachtung wird die Anzahl der Kombinationen deutlich verringert. Da jedoch die Kostenfunktionen in Abhängigkeit der Last aller Dienste definiert sind, kann es vorkommen, daß sich die resultierenden Kosten eines bereits optimierten Dienstes durch Laständerungen eines später optimierten Dienstes verändern und sich somit die günstigste Lösung verschiebt.

Die vorgestellte Heuristik berücksichtigt diese Fälle nicht. Um die Auswirkungen gering zu halten, sollte bei der Definition der Dienste darauf geachtet werden, daß genau die, deren Kosten stark von der Last anderer Dienste abhängen, möglichst als letzte bearbeitet werden.

5.3.2. Evaluierungssystem

Um einige der soeben diskutierten Betrachtungen belegen oder illustrieren zu können sowie weitere Eigenschaften der Lösung mittels Matlab oder der Heuristik aufzuzeigen, ist die Ausführung geeigneter Testpartitionierungen sinnvoll. Für diese Ausführung wird verständlicherweise eine Implementierung der Optimierungsaufgabe in Matlab (Vgl. Abschnitt 5.2.2) sowie eine funktionsfähige Umsetzung des Algorithmus $TestPartitioning$ (Vgl. 5.2.3) benötigt.

Im Fokus der Tests steht meist ein Vergleich der Ergebnisse und Leistungsparameter der Matlab- und der Heuristiklösung. Hierfür ist es notwendig, sowohl die gleichen Eingabedaten für beide Ansätze zu generieren, als auch die Ergebnisse beider Lösungen in ein vergleichbares Schema zu überführen. Um Fehler bei der manuellen Bearbeitung zu vermeiden und um sowohl Eingaben als auch Ergebnisse mehrerer tausend Datensätze berücksichtigen zu können, ist ein programmgestützes Evalulierungssystem sinnvoll, das die Testpartitionierung mittels beider Lösungsansätze automatisiert.

Ziel ist es, ein Testsystem, bestehend aus Testszenarien, Kostenfunktionen sowie weiteren Parametern (Vgl. Abschnitt 5.1), in einem Formalismus zu definieren und daraus dann die Eingaben für beide Lösungsalgorithmen in entsprechenden Formaten zu erzeugen. Die Ergebnisse beider Algorithmen wiederum sind in einen Formalismus zu überführen und, wenn möglich, für die Auswertung aufzuarbeiten.

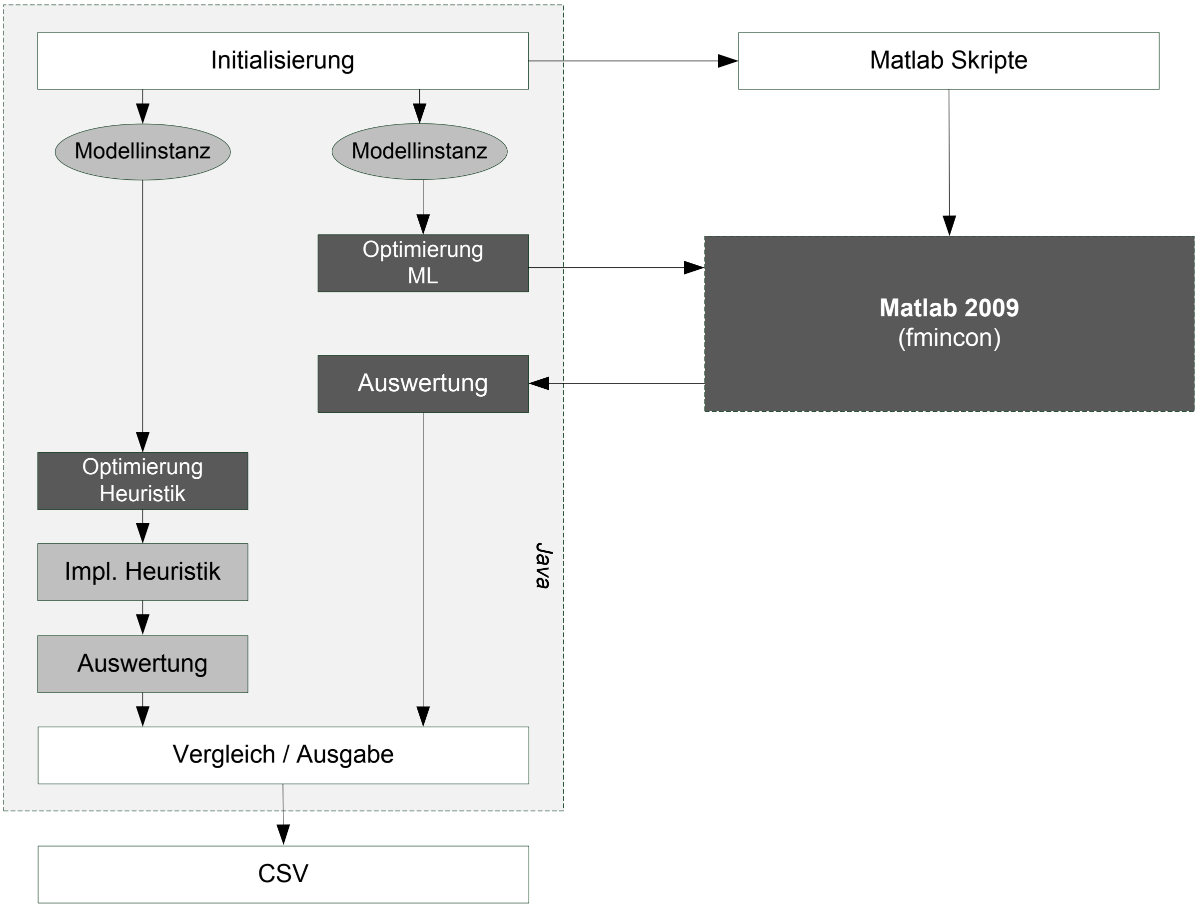


Abbildung 5.1.: Evaluierungssystem

Bestimmend für die Umsetzung dieses Systems ist die Möglichkeit, das externe Programm Matlab fernzusteuern. Der Hersteller stellt hierfür Programmierschnittstellen[2] für verschiedene Sprachen und Technologien bereit (Vgl. [78], S. 503ff). Eine relativ schlecht publizierte und dokumentierte Möglichkeit ist das Java Matlab Interface (JMI), welches es erlaubt, Matlab aus einem Java-Programm zu nutzen (Vgl. [51]). Da ein Großteil der Testbench auch in Java umgesetzt ist,

---

[2] Englisch: Application Programming Interface (API)

empfiehlt sich die Verwendung von Java ebenso für die Umsetzung des Evaluierungssystems. Entsprechend wurde auch $TestPartitioning$ in Java implementiert.

Die Struktur des Evaluierungssystems, die in Abbildung 5.1 dargestellt ist, ergibt sich aus den Anforderungen und technischen Rahmenbedingungen.

Klienten werden in einer konfigurierbaren Klasse beschrieben. Diese beinhaltet Methoden zur Lastberechnung auf Basis der hinterlegten Kostenfunktionen und speichert die Werte für die maximale Last a. Beim Start des Evaluierungssystems werden zwei getrennte Arrays angelegt, die jeweils identisch konfigurierte m Instanzen der Klientenklasse referenzieren. Ein Array wird durch Matlab bearbeitet, das zweite durch die Heuristik. Für Matlab werden desweiteren durch das Evaluierungssystem die Kostenfunktionen (Vgl. Listing 5.2 und die Zielfunktion (Vgl. Listing 5.3) als M-Dateien angelegt.

Im Anschluß wird Matlab per JMI gestartet. Da die Kostenfunktionen der Klienten in der Klientenklasse hinterlegt sind, wird vom Evaluierungssystem zunächst für jeden Dienst eines jeden Klienten die entsprechende Kostenfunktion als Matlab Skript in eine Skriptdatei geschrieben. Außerdem wird die Zielfunktion, die auch als Summe aller Kostenfunktionen berechnet wird, als Skript erzeugt. Diese Skripte sind für Matlab zugreifbar und können entsprechend als Eingaben für den Solver genutzt werden.

Diese Initialisierung ist nur einmal beim Start des Evaluierungssystems notwendig. Für die Evaluierung wird eine Menge von Laständerungen für jeden Dienst definiert. Die einzelnen Schritte der Laständerungen werden iterativ durch die folgenden Mechanismen berechnet.

Zunächst werden die Laständerungen mittels Matlab berechnet. Hierfür wird die Funktion $optimise$ verwendet, die in 5.5 vorgestellt wurde. Sämtliche Parameter, wie die aktuelle Last der einzelnen Klienten, die Maximallasten und die geforderte Laständerung, werden direkt beim Start von $optimise$ als Parameter übergeben. Der Aufruf der Funktion erfolgt mittels eines von JMI bereitgestellten Invoke-Kommandos.

Nach Terminierung der Optimierungsfunktion in Matlab gibt der blockierende JMI-Aufruf die Ergebnisse der Optimierung an das Java-basierte Evaluierungssystem zurück. Diese Ergebnisse umfassen in erster Linie den optimierten Vektor x (siehe Gleichung 5.19), der den Laständerungen l^+ entspricht, den Funktionswert der Zielfunktion an der Stelle x sowie einen Rückgabewert mit dem Grund der Terminierung. Korrekte, wenn auch suboptimale, Lösungen sind durch fmincon nur gefunden worden, wenn der Rückgabewert größer als 0 ist. Ein Wert von genau 0 entspricht der Terminierung wegen der Überschreitung von Iterationsbegrenzungen. Ein Wert kleiner als 0 beschreibt den Abbruch aufgrund unplausibler Lösungen von x (Vgl. [8], S. 248).

Vor dem Aufruf der $optimise$ Funktion wird die Systemzeit gespeichert und die Differenz zur Systemzeit nach der Rückkehr als Rechenzeit gespeichert.

Deutet der Rückgabewert auf gültige Ergebnisse hin, werden die berechneten Laständerungen in den Klientenobjekten gespeichert. Wie bereits in der theoretischen Betrachtung erwähnt, ist hierzu ein Runden der reellwertigen Vektorwerte notwendig. Ein Runden kann jedoch zu deutlichen Änderungen der entstehenden Kosten führen, so daß sowohl der von Matlab ermittelte Funktionswert der Zielfunktion, als auch der Wert nach dem Runden zum Vergleich gespeichert wird. Die Rechenzeit zum Runden bzw. zur Korrektur der Rundungsabweichungen wird nicht zur Rechenzeit der Funktion $optimise$ addiert.

Nach der Berechnung durch Matlab wird die Heuristik mit dem zweiten Klientenarray gestartet. Die übrigen Parameter, wie Kostenfunktionen, Lastgrenzen und -änderungen sind gleich zu denen in der Matlab-Ausführung. Da die Heuristik selbst in Java realisiert ist und nur mit ganzzahligen Werten arbeitet, kann diese direkt auf die Klientenobjekte zugreifen, so daß ein nachgelagertes Zuweisen und Runden der Ergebnisse entfällt. Auch für die Ausführung der Heuristik wird die Rechenzeit gemessen und gespeichert. Außerdem werden Informationen über die Anzahl an Rekursionen und Iterationen vermerkt.

Sind mehrere Laständerungsschritte vorgegeben, wird der gesamte Vorgang wiederholt. Die Klienten enthalten dabei jedoch bereits die aktualisierten Lasten aus der vorhergehenden Iteration. Um Messungen automatisiert wiederholen zu können, ist es desweiteren vorgesehen, den kompletten Vorgang beliebig häufig zu wiederholen, wobei bei jedem Durchlauf die Klienten neu initialisiert werden. Sämtliche Ergebnisse jedes Durchlaufes werden in CSV-Dateien gespeichert, so daß diese mit geeigneten Programmen ausgewertet werden können.

5.3.3. Ergebnisse

Um die Matlab-Lösung und die Heuristik vergleichen zu können, wurden mit dem beschriebenen Evaluierungssystem zahlreiche Kombinationen aus Klientenkonfigurationen und Testszenarien gelöst. Hierzu wurde das Evaluierungssystem auf einem durchschnittlichen Arbeitsplatz-PC (32-Bit Windows XP System, Dualcore Prozessor, 3 GB RAM) ausgeführt. Als virtuelle Maschine kommt das Java Development Kit von Sun/Oracle in der Version 1.6 zum Einsatz. Matlab wird in der Version 7.8.0.347 (R2009a) ebenfalls als 32-Bit Windows-Variante verwendet.

Die verschiedenen Tests werden durch variierende Konfigurationen definiert. Diese werden im folgenden jeweils durch eine Tabelle beschrieben.

Ein Beispiel für eine solche Konfiguration ist in Tabelle 5.1 zu finden. Zu erwähnen ist die dritte Zeile, die Informationen über die Anzahl der Kostenfunktionsgruppen enthält. Diese gibt an, wieviele verschiedene Kostenfunktionen verwendet werden. Diese dienen der Vereinfachung der Kostenmodelle, da es gerade bei großer Anzahl an Klienten praktisch kaum vorkommen wird, daß für jeden Klienten eine eigene, individuelle Kostenfunktion definiert werden muß. Vielmehr werden die Klienten bezüglich bestimmter Eigenschaften gruppiert und dann für die jeweiligen Gruppen eine Kostenfunktion definiert. Es ergeben sich also Äquivalenzklassen der Klienten, in denen die jeweiligen Elemente der Klasse auch die gleichen Kostenfunktionen zugeordnet bekommen. Die Anzahl der Kostenfunktionen in Tabelle 5.1 beschreibt folglich die Anzahl der Gruppen.

Parameter	Wert	Ergänzung
# Klienten	6	Anzahl der Klienten (m)
# Dienste	1	Anzahl der Dienste (n)
# Kostenfunktionsgruppen	3	Anzahl verschiedener Kostenfunktionen
$average$	$offset/m$	Wert von $average$, berechnet in jeder Rekursion
# Schritte	1	Anzahl der Schritte
$offset$	10	Laständerung in jedem der Schritte
# Messungswiederholungen	6	Anzahl der kompletten Wiederholungen

Tabelle 5.1.: Beispiel einer Konfigurationstabelle für einen Testlauf

Im Laufe dieser Arbeit erfolgt die Zuordnung der Klienten zu den Äquivalenzklassen über ein Modulo auf den Klienten-IDs. Am Beispiel der Konfiguration, wie sie in Tabelle 5.1 gegeben ist, werden den Klienten 1 und 4 die Kostenfunktion 1, den Klienten 2 und 5 die Kostenfunktion 2, usw. zugeordnet.

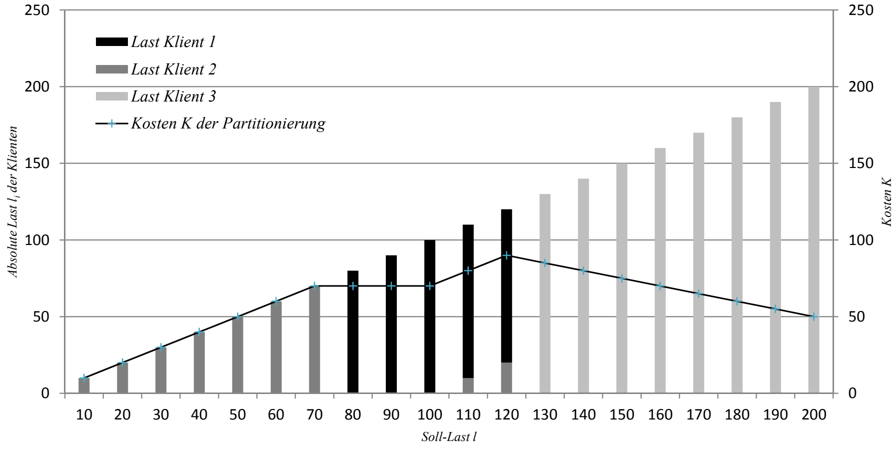
Aufgabe der Algorithmen (Matlab und Heuristik) wird es sein, die Laständerung $offset$ in jedem der Schritte möglichst günstig auf die Klienten, die zu jedem Zeitpunkt verfügbar sind, zu verteilen. Ggf. werden die Messungen unter identischen Startbedingungen wiederholt.

5.3.3.1. Lineare Kostenfunktionen

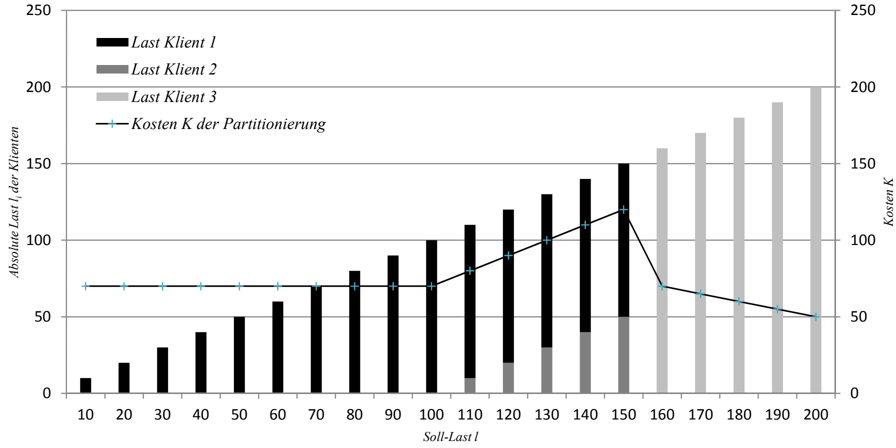
In einem ersten Schritt wird die Testpartitionierung mit einer kleinen Menge Klienten getestet, die wiederum durch einfache, lineare Kostenfunktionen beschrieben sind. Dies bietet die Möglichkeit, das globale Optimum analytisch zu bestimmen und mit den Ergebnissen von Matlab und der Heuristik zu vergleichen.

Die Konfiguration für den ersten Testlauf ist in Tabelle 5.2 gegeben. Als Kostenfunktionen kommen drei lineare Funktionen (Gleichung 5.27 bis 5.29) zum Einsatz. Die Kostenfunktionen $c_{1,1}$ und $c_{3,1}$ haben eine Unstetigkeit an der Stelle 0 um sicherzustellen, daß die Nichtbelegung des Klienten auch zu Kosten von 0 führen. Die maximale Last $a_{s,1}$ der einzelnen Klienten s für den (einzigen) Dienst 1 sind in den Gleichungen 5.30 bis 5.32 gegeben. Es werden 20 Messungen durchgeführt, bei der die initiale Last jeweils 0 ist und eine Laständerung von $10, 20, ...200$ zu generieren ist.

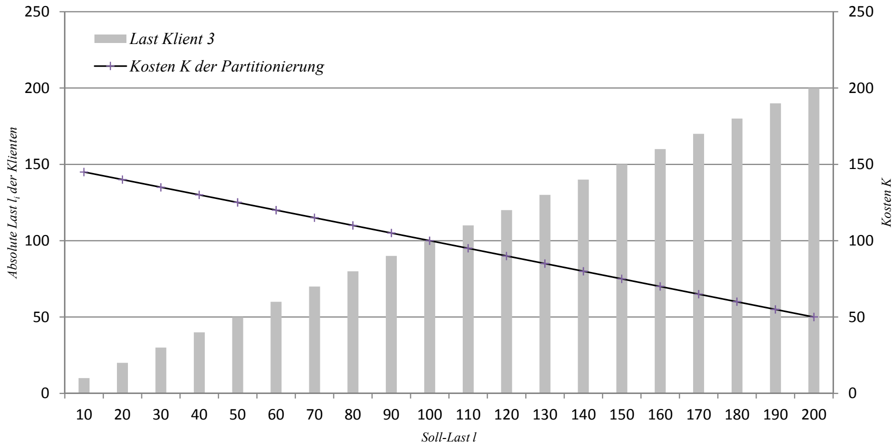
Aus dieser Konfiguration ergibt sich für die verschiedenen Laständerungen eine analytisch bestimmte, kostenoptimale Belegung der Klienten, wie sie in Abbildung 5.2(a) graphisch



(a) Analytisch bestimmt



(b) Heuristik



(c) Matlab

Abbildung 5.2.: Messung 1 - Verteilung der Lasten auf Klienten, resultierende Kosten

Parameter	Wert	Ergänzung
# Klienten	3	$= m$
# Dienste	1	
# Kostenfunktionsgruppen	3	
$average$	$offset/m$	
# Schritte	1	
$offset$	10, 20, ..., 200	20 Messungen mit versch. Laständerungen
# Messungswiederholungen	6	Wiederholung jeder Messung

Tabelle 5.2.: Konfiguration der Messung 1 - Lastverteilung

dargestellt ist. Jeder Balken repräsentiert eine absolute Last, die verschieden auf die drei Klienten verteilt wird, was entsprechend farblich verdeutlicht wird. Die daraus resultierenden Kosten sind als Kurve dargestellt. Wird die Testpartitionierung der gegebenen Konfiguration im Evaluierungssystem ausgeführt, führt dies zu den Ergebnissen, die in Abbildung 5.2(b) für die Nutzung der Heuristik und in Abbildung 5.2(c) für die Nutzung von Matlab gezeigt werden.

$$c_{1,1}(x_1) = \begin{cases} 0 & , x_1 = 0 \\ 70 & , x_1 > 0 \end{cases} \tag{5.27}$$

$$c_{2,1}(x_1) = x_1 \tag{5.28}$$

$$c_{3,1}(x_1) = \begin{cases} 0 & , x_1 = 0 \\ 150 - (0.5 \cdot x_1) & , x_1 > 0 \end{cases} \tag{5.29}$$

$$a_{1,1} = 100 \tag{5.30}$$

$$a_{2,1} = 150 \tag{5.31}$$

$$a_{3,1} = 300 \tag{5.32}$$

Beim Vergleich von Ergebnissen der Heuristik mit dem analytischen Optimum fallen zunächst die höheren Kosten im generierten Lastbereich 10 bis 60 auf. Diese resultieren aus der Nutzung von Klient 1 anstatt Klient 2. Die Ursache hierfür liegt in den iterativen Optimierungsschritten der Heuristik. Bei einer zu generierenden Last von bspw. 30 ergibt sich $average$ zu 10 (in Messung 1 gilt: $average = offset/\#Klienten$) und wird initial auf die drei Klienten verteilt. Daraus ergeben sich Kosten von $(70, 10, 145)$ für die Klienten $(1, 2, 3)$ und folglich Gesamtkosten von 225. Im anschließenden Optimierungsschritt wird versucht, durch Verschieben von Lasten $average$ auf andere Klienten eine Kostenoptimierung zu finden. Zielführend ist es, Klient 3 zu entlasten. Wird dessen Last auf Klient 1 verschoben, ergeben sich Kosten von $(70, 10, 0)$. Nach Klient 2 ergibt sich $(70, 20, 0)$. Demzufolge werden in der ersten Iteration die Lasten mit $(20, 10, 0)$ auf die

Klienten 1 bis 3 verteilt. In den folgenden Optimierungsiterationen wird höchstens ein Lastpaket von $average = 10$ verschoben. Eine Kostenverbesserung kann bei Klient 1 nur erreicht werden, indem alle 20 Lasteinheiten in einer Operation entfernt werden. Da dies nicht möglich ist, ist der nächst bessere Optimierungsschritt, sämtliche Lasten bis zum Erreichen von 70 Einheiten auf Klient 1 zu verschieben.

Für eine Laständerung von 70 existieren zwei optimale Lösungen. Sowohl die Verlagerung der kompletten Last auf Klient 1 als auch auf Klient 2 führen zu Gesamtkosten von 70. Entsprechend sind die analytische und die mittels Heuristik ermittelte Lösung korrekt und optimal.

Höhere Kosten im Vergleich zum Optimum entstehen auch im Lastbereich von 130 bis 150. Bei den jeweils resultierenden $average$-Werten führt die Optimierung mit der Heuristik zu den gleichen Abläufen wie am o.g. Beispiel mit einer Last von 30. Erst bei einem $average$ größer als 50 kann in der ersten Optimierungsiteration der Klient 3 nicht mehr auf eine Last von 0 gebracht werden, da die maximale Last für Klient 1 genau 100 ist. Demzufolge ergibt sich durch die Verschiebung von Klient 1 zu Klient 3 eine höhere Kosteneinsparung.

Die von Matlab errechnete Lösung, wie sie in Abbildung 5.2(c) dargestellt ist, weicht stark vom analytischen Optimum ab. Hier wird prinzipiell die komplette Last auf Klient 3 gepackt. Die führt vor allem im Lastbereich 10 bis 100 zu deutlich höheren Kosten.

Einfluß der Unstetigkeitsstellen Gründe für die deutliche Abweichung der Matlab-Lösung in Abbildung 5.2(c) liegen vermutlich darin begründet, daß Unstetigkeitsstellen, wie sie in der Konfiguration von Messung 1 in den Kostenfunktion $c_{1,1}$ und $c_{3,1}$, jeweils an der Stelle $x_1 = 0$, auftreten, dem Matlab-Solver Probleme bereiten.

Um diese Vermutung zu prüfen, wird ein weiterer Testlauf mit geänderten Kostenfunktionen durchgeführt. Die Unstetigkeitsstellen entfallen, so daß sich für $c_{1,1}$ und $c_{3,1}$ (siehe Gleichungen 5.33 und 5.34) auch im Falle von $x_1 = 0$ Kosten größer als 0 ergeben. Für diese Messung 2 wurden die übrigen Parameter im Vergleich zur Messung 1 nicht verändert.

$$c_{1,1}(x_1) = 70 \tag{5.33}$$

$$c_{3,1}(x_1) = 150 - (0.5 \cdot x_1) \tag{5.34}$$

Aufgrund der geänderten Kostenfunktionen wird das Optimum nun immer erreicht, indem die komplette Laständerung l dem Klient 3 zugeordnet wird, da dies durch die Differenz zu Kostensenkungen führt. Entsprechend ergibt sich eine analytisch bestimmte optimale Verteilung äquivalent zu den Matlab-Ergebnissen in Abbildung 5.2(c). Allerdings addieren sich in jeder Messung zu den resultierenden Kosten noch $c_{1,1}(0) = 70$ Einheiten.

Wird die Konfiguration von Messung 2 mittels der Heuristik berechnet, ergeben sich die Zuordnungen und Kosten, die in Abbildung 5.3 dargestellt sind. Diese entsprechen nahezu exakt den analytisch bestimmten, optimalen Werten. Ist der Quotient aus zu generierender Last und

Anzahl der Klienten nicht ganzzahlig, wird dem Klient 1 jeweils eine Lasteinheit zugeordnet (angegeben in den Ziffern über den Balken). Aufgrund der Ergebnisrundung in der Berechnung der Kostenfunktion bleiben die Kosten jedoch optimal.

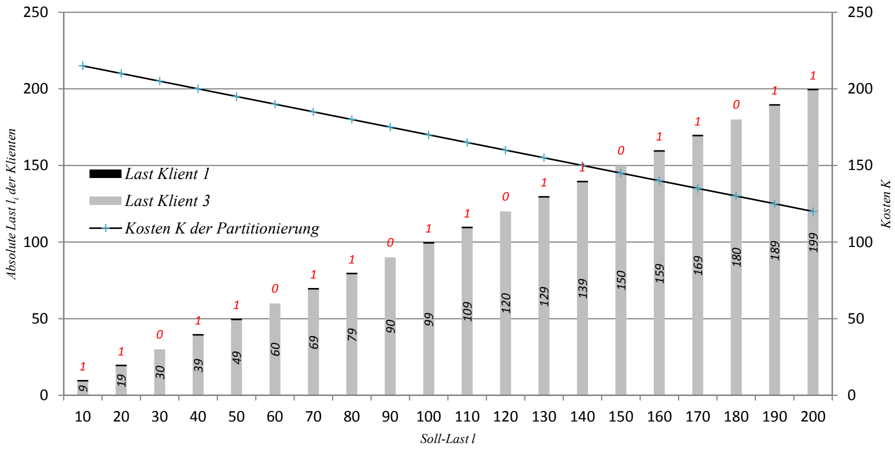


Abbildung 5.3.: Messung 2 - Verteilung der Lasten auf Klienten, resultierende Kosten (berechnet durch Heuristik). Die Ziffern über den Balken geben die Lasten an, die Klient 1 zugeordnet sind.

Beim Vergleich der Ergebnisse von Messung 2 mit den Matlab-Ergebnissen von Messung 1 scheint die Vermutung bestätigt, daß der $fmincon$-Solver Probleme hat, die Unstetigkeit an der Stelle $x_1 = 0$ zu finden bzw. zu nutzen.

Einfluß von average Bereits in der theoretischen Betrachtung des heuristischen Lösungsansatzes wurde der Einfluß des Parameters $average$ auf die Laufzeit und die Qualität der Partitionierung diskutiert. In Messung 1 war der Wert von $average$ statisch und die resultierenden Ergebnisse vor allem im Bereich einer Laständerung von 130 bis 150 suboptimal.

Im folgenden wird deshalb eine weitere Konfiguration geprüft, in der der Einfluß des Wertes $average$ analysiert wird. Diese besteht aus zwei Messungen für eine Laständerung von 100 und 150. Der Wert von $average$ wird für die Testläufe jeweils geändert. Als Kostenfunktionen kommen wieder die ursprünglichen Funktionen aus Gleichungen 5.27 bis 5.32 zum Einsatz. Weitere Details der Konfiguration sind in Tabelle 5.3 angegeben.

Als Ergebnis der Messung ist festzustellen, daß die Größe von $average$ im Falle dieser linearen Kostenfunktionen keinen Einfluß auf die Partitionierungsergebnisse hatte. In allen Messreihen wurden die gleichen Ergebnisse erzielt, wie sie für den Fall $average = offset/m$ in Abbildung 5.2(b) angegeben sind.

Allerdings variiert in Abhängigkeit von $average$ die resultierende Rechenzeit der Heuristik. Da die absolute Rechenzeit wegen der geringen Anzahl der Klienten in jedem Falle nur im Millisekundenbereich liegt, ist eine Betrachtung dieser nicht sinnvoll. Sie wird zu stark von Seiteneffekten, wie dem Scheduling des Betriebssystems, beeinflußt.

Parameter	Wert	Ergänzung
# Klienten	3	
# Dienste	1	
# Kostenfunktionsgruppen	3	
$average$	1, 2, 5, 10, ...	
# Schritte	1	
$offset$	100 und 150	2 unabhängige Durchläufe
# Messungswiederholungen	1	

Tabelle 5.3.: Konfiguration der Messung 3 - Einfluß von $average$

	$offset = 100$		$offset = 150$	
$average$	$\#Rek$	$\#Iter$	$\#Rek$	$\#Iter$
1	33	100	57	149
2	16	50	28	74
5	6	20	11	29
10	3	10	5	14
20	1	5	2	7
50	0	2	0	2
100	0	1	0	1
150	0	1	0	1

Tabelle 5.4.: Messung 3 - Anzahl Rekursionen und Iterationen bei verschiedenen Werten von $average$ (berechnet von Heuristik)

Die absolute Zeit ergibt sich in erster Linie aus der Anzahl der nötigen Rekursionen und Iterationen. Diese werden zur Laufzeit gezählt und sind somit seiteneffektunabhängig. Die Ergebnisse für Messung 3 sind in Tabelle 5.4 dargestellt. Die Anzahl der Rekursionen ($\#Rek$) umfaßt lediglich die Anzahl der rekursiven Aufrufe, nicht aber den initialen Ruf der Optimierungsfunktion. Die Anzahl der Iterationen ($\#Iter$) addiert alle Iterationen in jeder Rekursion auf.

Es ist an den Ergebnissen zunächst deutlich zu erkennen, daß kleine Werte von $average$ in jedem Falle zu häufigen Rekursionen und damit auch zu häufigen Iterationen führt. Mit jeder Verdopplung von $average$ halbiert sich in etwa auch die Anzahl der Rekursionen. Wächst $average$ über den tatsächlichen Durchschnitt aus dem $offset$ und der Anzahl der Klienten, ist keine Rekursion notwendig - zumindest für den Fall dieser Messung, in der die Zuordnung von $average$ auf die

Parameter	Wert	Ergänzung
# Klienten	3	$= m$
# Dienste	1	
# Kostenfunktionsgruppen	3	
$average$	$offset/m$	
# Schritte	20	
$offset$	10	
# Messungswiederholungen	1	

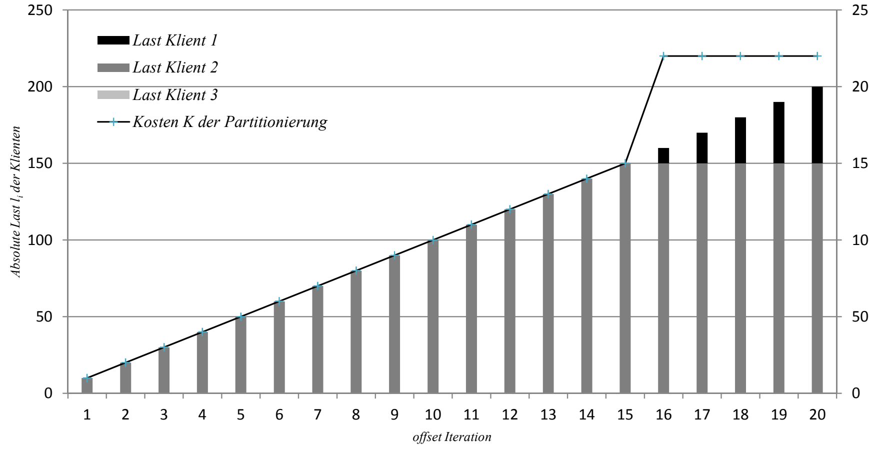
Tabelle 5.5.: Konfiguration der Messung 4 - Lastverteilung

betreffenden Klienten nicht die maximale Last dieser überschreitet. Auch das Verhältnis zwischen Rekursionen und Iterationen ist nahezu konstant und beträgt rund 3.

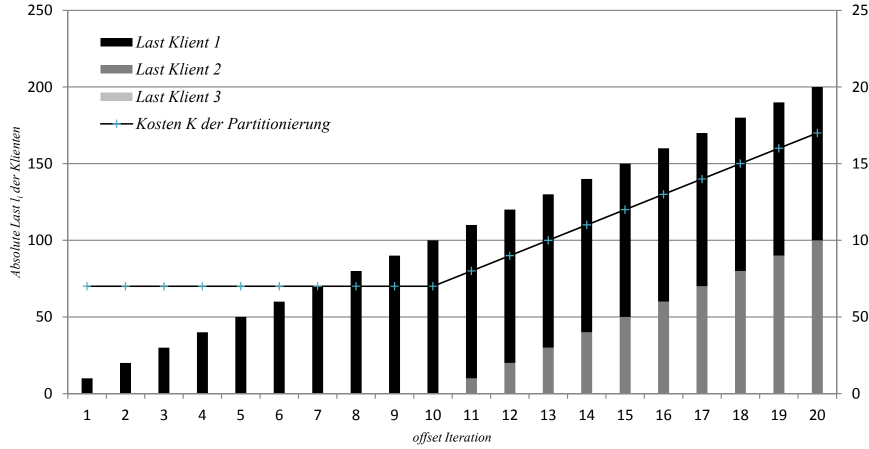
Mehrere offset-Schritte In den bisherigen Messungen wurden ausschließlich Einzellasten verschiedener Größen partitioniert, wobei die initiale Last eines jeden Klienten 0 betrug. Da für die Testbench jedoch eine kontinuierliche Arbeit auch auf bereits lastgenerierenden Klienten notwendig ist, wird im folgenden eine Messung 4 auf Basis der Konfiguration in Tabelle 5.5 durchgeführt. Im Gegensatz zur Messung 1 wird in jedem Schritt eine weitere Last von 10 Einheiten auf die Ergebnisse der vorherigen Optimierung hinzugefügt.

Das Hauptproblem bei dieser Konfiguration ist, daß die Ergebnisse eines offset-Schrittes stark von den Ergebnissen des bzw. der vorhergehenden Schritte abhängen. Ausgehend von der Annahme, daß in jedem Schritt die optimale Lösung gefunden wird, ergibt sich ein analytisch bestimmtes Optimum, wie es in Abbildung 5.4(a) dargestellt ist. Die Ergebnisse für die Berechnung durch die Heuristik und durch Matlab sind entsprechend in Abbildung 5.4(b) und 5.5(b) zu finden. Es fällt auf, daß alle Varianten komplett unterschiedliche Ergebnisse liefern. Die Ursache hierfür liegt in der Verteilung der ersten initialen Lasten, die in allen drei Ergebnisdiagrammen verschiedenartig auf die Klienten verteilt werden. Die darauf folgenden weiteren Lasten setzen auf dieser initialen Verteilung und den daraus resultierenden Kosten auf. In Folge entstehen komplett verschiedene Belegungen.

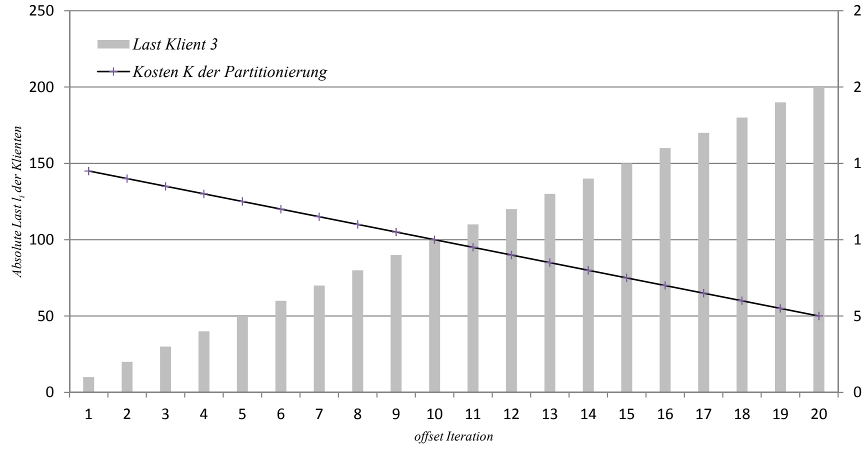
In Folge der Iterationen ergibt sich, daß ausgerechnet die aus Kostensicht ungünstigste Initiallösung von Matlab nach 20 Iterationen zum günstigsten Ergebnis führt. Das analytische Optimum, das von einer optimalen Verteilung in jeder Iteration ausgeht, führt hingegen in Summe zu den höchsten Kosten. Es ist zu beachten, daß dies ein zufälliger Effekt aus der Kombination der Kostenfunktionen ist und nicht verallgemeinert werden kann. Vielmehr muß aus den Ergebnissen geschlossen werden, daß eine qualitative Bewertung durch die direkten Auswirkungen früherer Iterationsergebnisse kaum möglich ist.



(a) Analytisch bestimmt



(b) Heuristik



(c) Matlab

Abbildung 5.4.: Messung 4 - Verteilung der Lasten auf Klienten, resultierende Kosten

Parameter	Wert	Ergänzung
# Klienten	12, 24, 48, ... , 768	$= m$
# Dienste	1	
# Kostenfunktionsgruppen	3	
$average$	$offset/m$	
# Schritte	3	
$offset$	$16,667 \cdot m$	ergibt 200, 400, ...
# Messungswiederholungen	1	

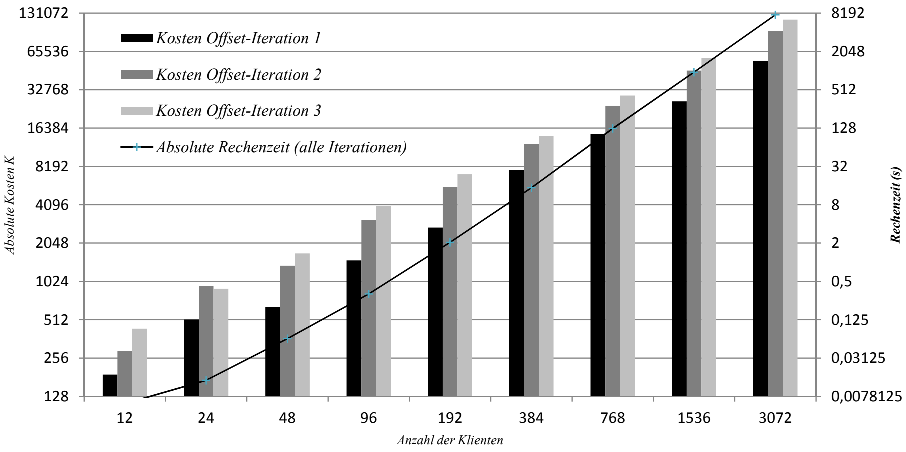
Tabelle 5.6.: Konfiguration der Messung 5 - Kosten und Ausführungszeiten

Klientenanzahl Ein weiterer relevanter Parameter mit Einfluß auf die Ergebnisse und insbesondere die Leistungsfähigkeit ist die Anzahl der Klienten. Die war bisher zur einfacheren Interpretation der Optimierungsergebnisse auf genau 3 festgelegt. In der folgenden Messung 5 wird die Anzahl der Klienten kontinuierlich verdoppelt. Da die Zahl der verschiedenen Kostenfunktionen, also die Funktionsgruppen, bei 3 bleibt und zur besseren Einschätzung der Optimierungsergebnisse eine gleichmäßige Verteilung aller Kostenfunktionen sinnvoll ist, wird als Startwert für die Anzahl der Klienten 12 gewählt. Damit ergibt sich, daß jede der Kostenfunktionen aus Gleichung 5.27 bis 5.29 genau viermal verwendet wird. Die übrigen Details der Konfiguration sind in Tabelle 5.6 zu finden.

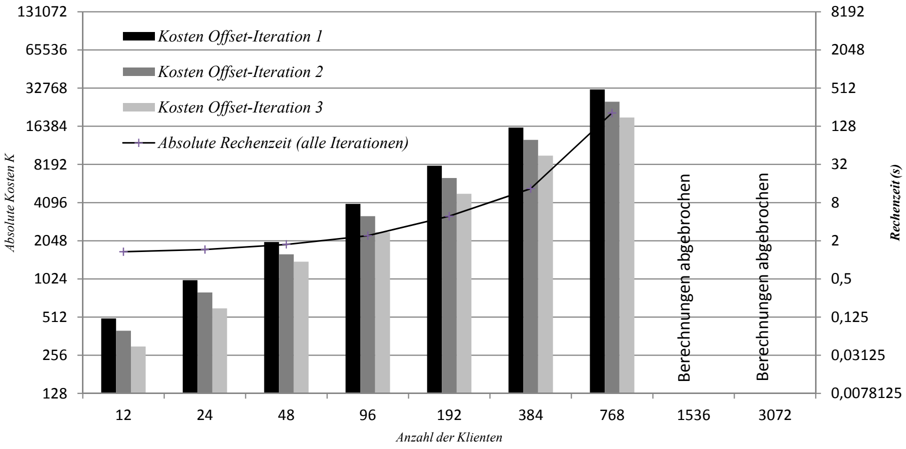
Die Ergebnisse der Partitionierung durch die Heuristik und durch Matlab sind in Abbildung 5.5 zu finden. Im Gegensatz zu den bisherigen Messungen werden hier die Kosten der errechneten Partitionierung nach jeder Iteration angegeben. Die hinterlegte Kurve entspricht der benötigten absoluten Rechenzeit. Es ist zu beachten, daß alle Werte in logarithmischen Skalen abgetragen sind.

Im Falle der Matlab-Ergebnisse ist es wenig überraschend, daß die Partitionierung zu vergleichbaren Ergebnissen wie in den bisherigen Messungen führt. Sämtliche Last wird auf die Klienten der Funktionsgruppe 3 mit der Kostenfunktion 5.29 verteilt. Entsprechend linear entwickeln sich die Kosten für die Iterationen und die verschiedenen Messungen mit steigender Zahl der Klienten und implizit steigender Last.

Prinzipiell ist dieser Trend natürlich auch bei der Partitionierung mittels der Heuristik zu erkennen. Jedoch sind die Kostenentwicklungen mit steigender Klientenanzahl und der steigenden Last nicht von einem konstanten Proportionalitätsfaktor geprägt. Beispielsweise betragen die Kosten für die erste Iteration mit 12 Klienten genau 190 ($offset = 200$). Mit der Verdopplung auf 24 Klienten ($offset = 400$) steigen die Kosten auf 514, was einem Faktor von rund 2,7 entspricht. Bei der nächsten Verdopplung steigen die Kosten jedoch „nur“ auf 643, was wiederum einen Faktor von 1,25 entspricht.



(a) Heuristik



(b) Matlab

Abbildung 5.5.: Messung 5 - Kosten und Ausführungszeiten der Partitionierung

Im Vergleich beider Varianten fällt auf, daß nach dem ersten Schritt die Kosten der Heuristiklösung weniger oder genau 50% der Kosten der Matlablösung betragen. Da sich in Matlab nach drei Iterationen die Kosten knapp halbieren, sind sie somit deutlich niedriger als in der zugehörigen Heuristiklösung. Hierbei handelt es sich um das gleiche Verhalten, wie es in Messung 4 diskutiert wurde (insbesondere Abbildungen 5.4(b) und 5.5(b)).

In Messung 5 kommt der Betrachtung der Berechnungszeiten besondere Bedeutung bei. In beiden Diagrammen sind die Berechnungszeiten für jede Messung, die für die entsprechende Klientenanzahl durchgeführt wurde, als Summe der Zeiten aller drei Schritte dargestellt. Die Berechnungszeit der einzelnen Offset-Schritte ergibt sich folglich durch Division mit der Anzahl der Schritte (3).

Da die Skalierung in beiden Diagrammen identisch ist, fällt zunächst die deutlich höhere Berechnungszeit von Matlab bei geringerer Anzahl von Klienten auf. Da es sich bei Matlab um ein umfängliches Programmpaket handelt, ist vermutlich die interne Generierung der entsprechenden Datenstrukturen die Ursache für diese verhältnismäßig große Setup-Zeit. Im Falle der Heuristik wird nur die reine Berechnungszeit gemessen, da hier die Datenstrukturen vorab angelegt werden können und nur als Referenzen an den Algorithmus übergeben werden müssen. Bei Matlab hingegen ist nur ein Aufruf des kompletten Solvers mit Übergabe der entsprechenden Parameter mittels Java Matlab Interface möglich, so daß die Erzeugung interner Datenstrukturen aus der Berechnungszeit nicht herausgefiltert werden kann. Da es sich bei diesen verhältnismäßig konstanten Setup-Zeiten nur um Werte zwischen 1 und 2 Sekunden handelt, können diese jedoch toleriert werden.

Im Gegensatz dazu beträgt die Berechnungszeit der Heuristik für eine Klientenanzahl unter 100 weniger als 0,5 Sekunden - kann also weitgehend vernachläßigt werden. In diesen Größenordnungen haben externe Einflüsse, wie bspw. das Betriebssystemscheduling, noch immer großen Einfluß auf die absolute Rechenzeit.

Deutlich interessanter ist die Entwicklung bei höheren Klientenzahlen, wo die Rechenzeiten in den zwei- bis vierstelligen Sekundenbereich vordringen. Werden die logarithmischen Skalen berücksichtigt, steigt die Rechenzeit der Heuristik linear mit der Zahl der Klienten. Der Anstiegsfaktor schwankt dabei zwischen 5 bis 8, wobei es sich hierbei um die Gesamtlaufzeit aller drei Schritte handelt. Wird dies mit der theoretisch betrachteten Komplexität der Berechnung verglichen (Vgl. Abschnitt 5.3.1), in der die Klientanzahl quadratisch eingeht, wäre bei einer Verdopplung der Klientenanzahl mit einem Faktor von 4 zu rechnen. Demzufolge liegen die praktischen Messungen mit einem Linearfaktor von 1,5 bis 2 zur quadratischen Klientenanzahl im Rahmen der theoretischen Betrachtung.

Eine Berechnung mit 1536 bzw. 3072 Klienten ist mit Matlab auf dem benutzen Desktopsystem nicht möglich. Matlab bricht in diesem Falle mit der Meldung „Out of memory“ ab, wie sie im Listing 5.6 dargestellt ist. Offensichtlich sind hier die Grenzen eines 32-Bit Systems erreicht. Zu Beginn der Arbeit stand kein 64-Bit Matlab zur Verfügung und ein späterer Wechsel hätte die Vergleichbarkeit der einzelnen Messungen in Frage gestellt.

```
Caused by: com.mathworks.jmi.MatlabException: Error using ==> ldl
Out of memory.
```

Listing 5.6: Fehlermeldung

Im Bereich von 100 bis 768 Klienten steigt die Rechenzeit vermutlich exponentiell an. Jedoch erlaubt die eingeschränkte Zahl erfolgreicher Berechnungen eine genaue Einschätzung dessen nicht.

Werden die Ergebnisse der Rechenzeiten verglichen, schneidet die Heuristik erwartungsgemäß deutlich besser ab. Da die absoluten Zeiten bei geringer Klientenanzahl auch gering sind, spielen die Differenzen zwischen beiden Verfahren keine nennenswerte Rolle. Jedoch führt die vermutlich exponentielle Steigerung der Rechenzeit von Matlab schnell zu praktisch nicht nutzbaren Verzögerungen. Jedoch sind auch die absoluten Berechnungszeiten der Heuristik bei großer Anzahl von Klienten hoch, so daß ein praktischer Einsatz im Einzelfall zu prüfen ist.

5.3.3.2. Nichtlineare, unstetige Kostenfunktionen

Lineare Kostenfunktionen, wie sie im vorhergehenden Abschnitt für die Messungen verwendet wurden, sind in einer Vielzahl praktischer Anwendungsfälle nicht geeignet, Nutzerverhalten und technische Aspekte des Tests zu modellieren. Wie bereits bei der Definition der Kostenfunktionen angeführt, sind vielmehr nichtlineare Funktionen, die ggf. auch über Unstetigkeiten verfügen, realistischer. Demzufolge sollen im folgenden auch derartige Kostenfunktionen zur Messung der Leistungsfähigkeit von Matlab und der Heuristik herangezogen werden. Im Gegensatz zu den Messungen mit linearen Funktionen steht dabei der Vergleich zu einem analytisch bestimmten Optimum nicht im Vordergrund, da dieses aufgrund der verwendeten Funktionen schwierig zu ermitteln ist.

Parallel zum Wechsel der Kostenfunktionen werden auch mehrere Dienste eingeführt - in diesem Fall drei. Die Klienten werden in fünf Kostenfunktionsgruppen aufgeteilt. Die Funktionen sind im Anhang A als Gleichungen A.1 bis A.11 zu finden. Auch die maximalen Lasten werden an die Kostenfunktionsgruppe gebunden und sind in Gleichungen A.12 bis A.17 angegeben.

Bei genauer Betrachtung der Kostenfunktionen fällt auf, daß die Last einiger Dienste j nun nicht ausschließlich von der Last x_j des zugehörigen Dienstes abhängig ist, sondern auch von den Lasten x_k anderer Dienste k. Auf diese Weise können spezielles Nutzerverhalten oder auch technologische Abhängigkeiten modelliert werden.

Abweichungen zwischen Soll- und Ist-Last Anhand der Kostenfunktionen und der Konfiguration 6, gegeben in Tabelle 5.7, wird nun zunächst untersucht, inwieweit gültige Lösungen durch die Heuristik und Matlab gefunden werden und wie die Relation zwischen den entsprechenden Kosten ist.

Im Zentrum der Messung steht die Frage nach Korrektheit der erzeugten Testpartitionierung. Korrektheit definiert sich über die Nebenbedingung in Gleichung 5.6, die besagt, daß sich die Summe aller von Klienten generierten Lasten für einen Dienst an die geforderte Last des Szenarios annähern muß.

Wie bereits in der theoretischen Analyse im Abschnitt 5.3.1 erwähnt, ist für den Matlab-Solver zu berücksichtigen, daß die Optimierungsvariablen reellwertig sind. Das Testpartitionierungsproblem definiert jedoch Lasten, also die Optimierungsvariablen, als ganzzahlig an. Da im Falle nichtlinearer Kostenfunktionen auch nichtganzzahlige Werte für die Variablen zu lokalen Minima führen können und damit als Ergebnis der Optimierung mittels Matlab zurückgegeben werden, müssen diese Ergebnisse angepaßt werden.

Das verwendete Verfahren hat dabei sowohl elementaren Einfluß auf die Korrektheit der Ergebnisse im soeben diskutierten Sinn wie auch auf die resultierenden Kosten. Im folgenden werden deshalb zunächst zwei Verfahren verwendet, die als Skalarrunden und Ausgleichsrunden bezeichnet werden. Details zur Umsetzung dieser beiden Varianten sind im Anhang B erläutert.

Parameter	Wert	Ergänzung
# Klienten	10	$= m$
# Dienste	3	
# Kostenfunktionsgruppen	5	
$average$	$offset/m$	
# Schritte	1	
$offset$	$\{30, 15, 1\}$ $\{60, 30, 2\}$ ... $\{600, 300, 20\}$	für die drei Dienste
# Messungswiederholungen	1	

Tabelle 5.7.: Konfiguration der Messung 6 - Verhältnis Soll- und Ist-Last

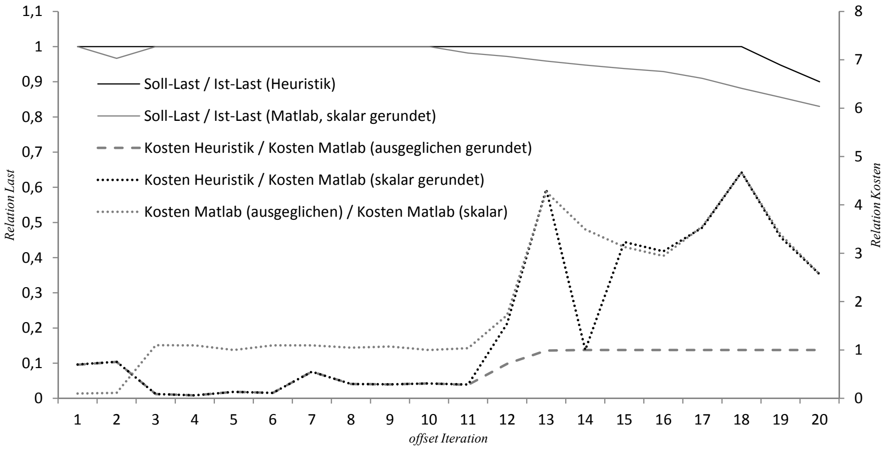
Die Ergebnisse der Partitionierung mit der Heuristik und den beiden Matlab-Varianten sind, aufgeteilt nach den drei Diensten, in Abbildung 5.6 dargestellt. In den Ergebnisdiagrammen sind verschiedene Größen dargestellt. Die nicht gepunkteten Kurven repräsentieren den Quotienten aus der Soll-Last und die durch die Heuristik sowie Matlab errechnete Ist-Last. Die punktierten Kurven repräsentieren hingegen die Verhältnisse der Kosten dieser einzelnen Lösungen.

Bei der Heuristik und der ausgeglichenen Rundung ist bei allen Diensten erkennbar, daß das Verhältnis zur Soll-Last zunächst 1 beträgt - also die komplette Laständerung auf die Klienten verteilt wurde. Erst ab einem bestimmten Wert der Laständerungen fällt die generierte Ist-Last. Im Falle von Dienst 1 geschieht dies ab einer Soll-Last größer als 540, in Dienst 3 ab einer Soll-Last größer 10. Dies ist keine Ungenauigkeit der Algorithmen sondern dem Erreichen der Maximallasten der Klienten geschuldet. Bei 10 Klienten liegt diese nämlich bei 540 bzw. 10 für die beiden genannten Dienste. Es kann aus dieser Messung folglich geschlossen werden, daß ε in der Gleichung 5.6 für diese Varianten jeden beliebig kleinen Wert größer als 0 annehmen kann.

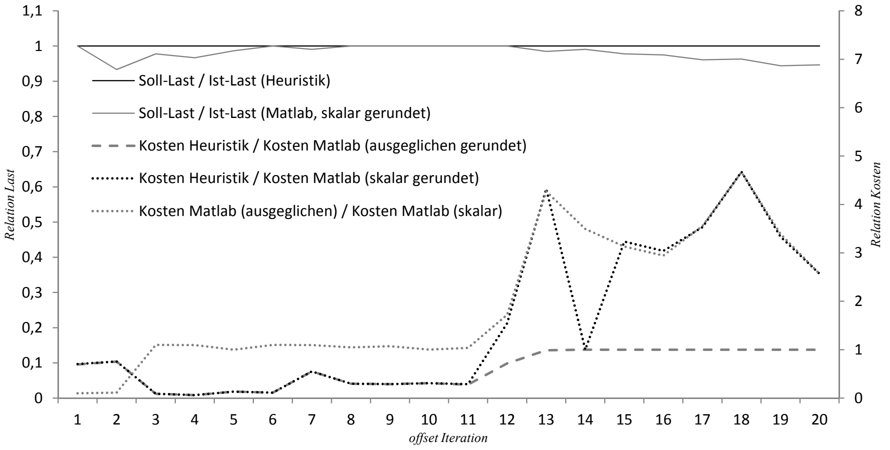
Anders ist dies im Fall der Skalarrundung. Hier fällt bei den Diensten 1 und 2 auf, daß sowohl im niedrigen als auch im hohen Laständerungsbereich mehr oder minder starke Abweichungen von bis zu 10% auftreten. Besonders auffällig wird es im Fall von Laständerungen kleiner als 4 bei Dienst 3. Hier sind Abweichungen von 100% meßbar.

Da diese Abweichungen den Validierungsprozeß stark beeinflussen können, stellt die Skalarrundung keine brauchbare Option dar. In folgenden Messungen wird deshalb ausschließlich das Ausgleichsrunden verwendet.

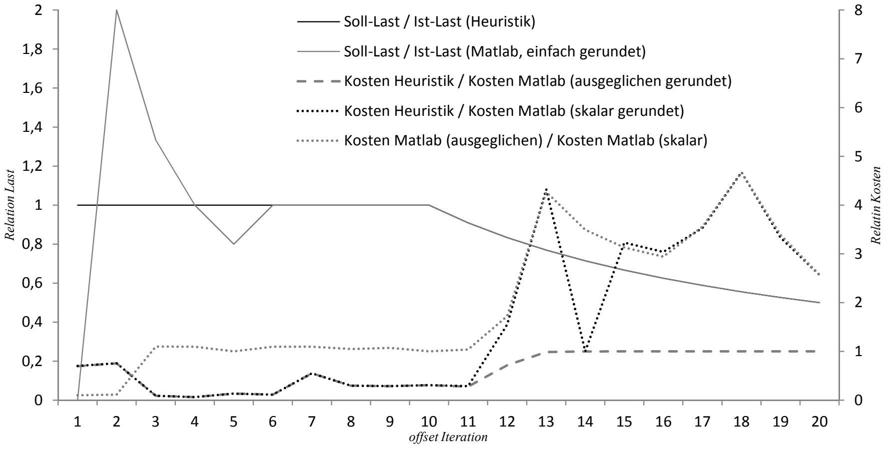
Es gilt jedoch zu prüfen, in welchem Maße das Ausgleichsrunden die resultierenden Kosten beeinflußt. Hierzu sind in den Diagrammen 5.6 auch die Kostenrelationen aufgezeigt, wobei diese



(a) Dienst 1



(b) Dienst 2



(c) Dienst 3

Abbildung 5.6.: Messung 6 - Lastverhältnisse zwischen Soll- und Ist-Last einzelner Dienste, resultierende Gesamtkosten

Parameter	Wert	Ergänzung
# Klienten	10	$= m$
# Dienste	3	
# Kostenfunktionsgruppen	5	
$average$	$offset/m$	
# Schritte	20	
$offset$	$\{30, 15, 1\}$	für die drei Dienste
# Messungswiederholungen	1	

Tabelle 5.8.: Konfiguration der Messung 1 - Verhältnis Ist- uns Soll-Last

immer die Gesamtkosten aller Dienste darstellen, und somit in den Diagrammen aller drei Dienste identisch sind.

Auffällig ist zunächst, daß die Quotienten der Heuristikkosten und der Matlabkosten, jeweils für den Fall der skalaren Rundung und der Ausgleichsrundung, nahezu identisch sind. Das Verhältnis kleiner 1 zeigt, daß die Kosten der Heuristik zum Teil um einige Faktoren unter denen der gerundeten Matlab-Lösung liegen. Steigen die Lasten und damit die Optimierungsspielräume, so steigt auch das Kostenverhältnis zwischen der Heuristik und der ausgeglichenen Rundung und konvertiert gegen ein Verhältnis von 1. Im Gegensatz dazu steigt das Verhältnis zwischen den Kosten der Heuristik und der skalar gerundeten Matlab-Lösung deutlich über 1 ab dem Punkt, an dem die erzeugten Lasten nicht mehr übereinstimmen. Die geringere Last, die durch das Runden der Matlab-Ergebnisse generiert wird, wirkt sich in der gegebenen Konfiguration massiv aus, so daß die Kosten der Matlab-Lösung deutlich sinken.

Der Vollständigkeit halber ist noch der Quotient der Kosten auf Basis der Ausgleichungsrundung und der Sklararrundung eingetragen. Der Quotient nahe 0 bei niedriger Gesamtlast resultiert in erster Linie aus den Abweichungen, die bei allen Diensten in diesem Bereich durch die skalare Rundung entstehen. Im folgenden Abschnitt liegt der Quotient leicht über 1, so daß die ausgeglichene Rundung etwas höhere Kosten verursacht als die skalar gerundete Variante.

Kostenrelation Heuristik / Matlab Ähnlich wie im Falle der linearen Funktionen ist auch für nichtlineare Funktionen die Entwicklung der Kosten von Heurikstik- und Matlablösung interessant. Hierzu wird eine Messung 7 definiert, die 20 identische Laständerungen berechnet. Die entsprechende Konfiguration ist in Tabelle 5.8 angegeben. Im Gegensatz zur Messung 6 werden nun die Lasten in jedem Schritt erhöht, so daß die Optimierungsergebnisse eines vorhergehenden Schrittes Einfluß auf den aktuellen Schritt haben.

Die Ergebnisse der Messung sind in Abbildung 5.7 dargestellt. Da die absoluten Werte einen großen Bereich überspannen, sind wieder Relationen zwischen Ist- und Soll-Last bzw. zwischen

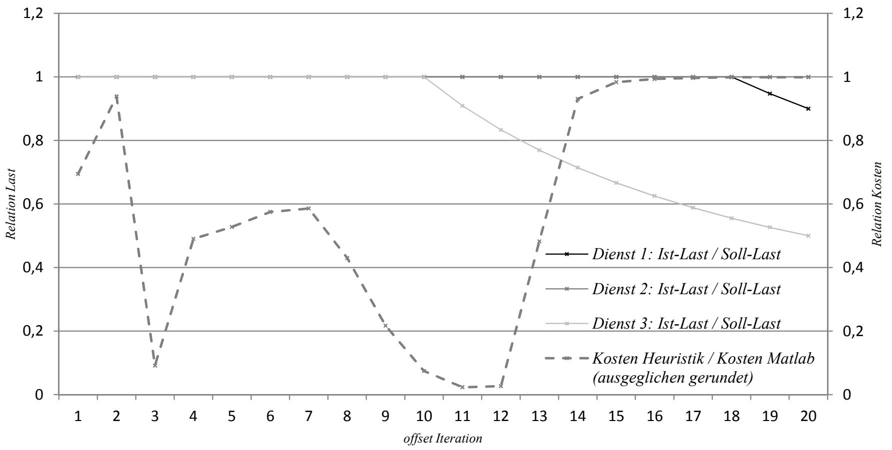


Abbildung 5.7.: Messung 7 - Vergleich Ist-/ Soll-Last, resultierende Kosten

den resultierenden Kosten der Optimierung mittels Heuristik und mittels Matlab aufgezeigt. Im Fall von Matlab wird die Ausgleichsrundung verwendet. Die auf diese Weise generierten Ist-Lasten sind in allen Schritten identisch mit denen der Heuristik.

Auch für diesen Test, in dem die absolute Last in jedem Schritt sich aus der Summe der Laständerungen aller Schritte ergibt, gelten natürlich die Lastgrenzen. Demzufolge wird im Schritt 18 die maximale Last für Dienst 1 und bereits im Schritt 10 die maximale Last für Dienst 3 erreicht. Entsprechend zwangsläufig reduziert sich das Verhältnis aus Ist- und Soll-Last auf Werte kleiner als 1.

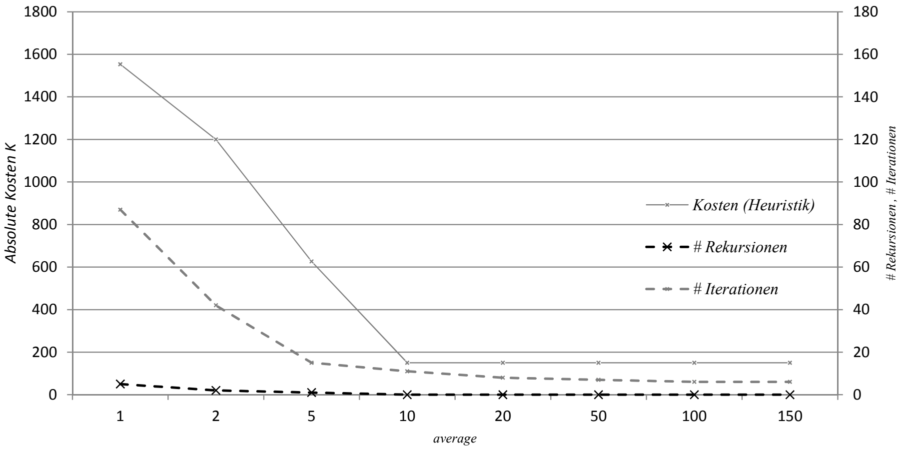
Weniger offensichtlich ist das Kostenverhältnis zwischen der Heuristik- und der Matlablösung. Der Quotient aus beidem liegt in allen Messungen mehr oder weniger deutlich unter 1, so daß die Heuristik immer Lösungen mit geringeren Kosten findet. Dabei bewegt sich die Relation in der ersten Hälfte der Messung zwischen $0,02$ und $0,94$ - schwankt also sehr stark. Werden die absoluten Kosten betrachtet, liegt die Ursache in den Schwankungen der Matlab-Ergebnisse. Während die Kosten der Heuristik monton mit den Lasten wachsen, gibt es bei Matlab Fälle, in denen die Kosten trotz steigender Lasten fallen. Entsprechenden Einfluß hat dies auf das dargestellte Verhältnis. Es ist anzunehmen, daß Matlab im Fall sehr kleiner Kostenverhältnisse keine der globalen Minima in den Kostenfunktionen gefunden hat.

Erreicht die absolute Last im zweiten Teil der Messung die maximal durch die Klienten erzeugbare, werden die Spielräume der Optimierung geringer, so daß die meisten gültigen Lösungen nahezu gleiche Kosten verursachen. Entsprechend nähert sich das Verhältnis zwischen Heuristik und Matlab auch nahezu der Gleichheit, was einem Quotienten von 1 entspricht.

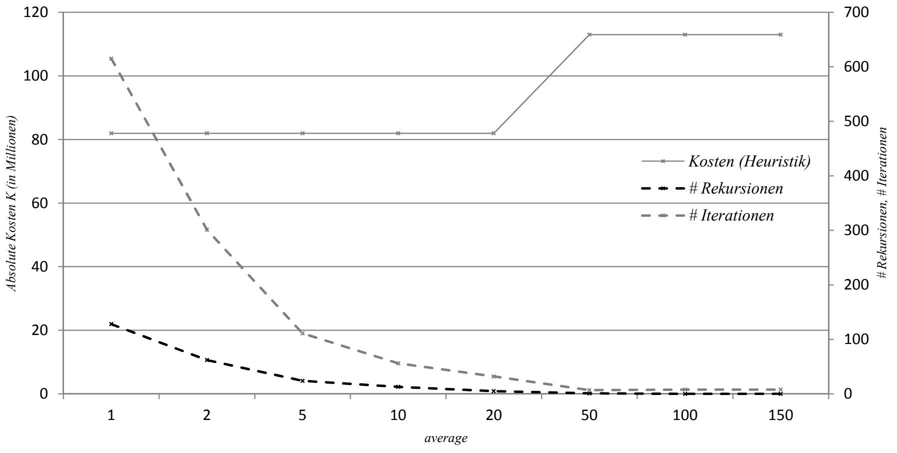
Einfluß von average Der Einfluß von $average$ auf die Berechnung und Lösung wurde schon für den Fall linearer Kostenfunktionen betrachtet. Aber gerade im nichtstetigen, nichtlinearen Fall sollte neben dem offensichtlichen Einfluß auf die Anzahl der Rekursionen und Iterationen auch ein Einfluß auf die Qualität der Ergebnisse bestehen. Zur Überprüfung wird eine Konfiguration, die in Tabelle 5.9 angeben ist, verwendet.

Parameter	Wert	Ergänzung
# Klienten	10	
# Dienste	3	
# Kostenfunktionsgruppen	5	
$average$	1, 2, 5, 10, ...	
# Schritte	1	
$offset$	$\{60, 30, 2\}$ $\{500, 300, 10\}$	zwei unabhängige Durchläufe mit leichter und starker Auslastung
# Messungswiederholungen	1	

Tabelle 5.9.: Konfiguration der Messung 8 - Ermittlung Rekursionen und Iterationen



(a) Einfluß von average (geringe Last)



(b) Einfluß von average (hohe Last)

Abbildung 5.8.: Messung 8 - Anzahl der Rekursionen und Iterationen, Kosten der Lösung

Die Messung wird unabhängig für zwei Fälle durchgeführt. Zum einen wird die Last, die für jeden Dienst erzeugt wird, gering gehalten, so daß eine Testpartitionierung einfach möglich ist. Im zweiten Fall hingegen wird eine Last für jeden Dienst vorgegeben, die nahe dem Maximum liegt, welches von allen Klienten überhaupt erzeugt werden kann. Die Ergebnisse von Matlab werden nicht betrachtet, da $average$ ein interner Parameter der Heuristik ist. Die Ergebnisse der Messungen mit der Heuristik sind in der Abbildung 5.8 für beide Fälle getrennt dargestellt.

Die Balken im Diagramm geben die Kosten der jeweiligen Lösung an. Im Gegensatz zur Variante mit linearen Kostenfunktionen ist nun tatsächlich ein Einfluß von $average$ auf die Kosten der gefundenen Testpartitionierung zu erkennen. Allerdings ist die Auswirkung zwischen den beiden Fällen genau entgegengesetzt. Während bei der geringen Auslastung (Abbildung 5.8(a)) die Last mit steigendem Wert von $average$ abnimmt, ist sie bei hoher Last nahezu konstant und steigt erst um ca. 25%, wenn $average$ derart große Werte erreicht, daß für einige Klienten von vornherein die maximale Last überschritten wird.

Es ist aus den Tests für lineare und nichtlineare Kosten also keine belastbare, allgemeine Aussage über den genauen Einfluß von $average$ auf die Qualität der Lösung - also die aus der gefundenen Partitionierung folgenden Kosten - möglich. Es ist anzunehmen, daß der Einfluß stark von der Art der Kostenfunktionen abhängt.

Im Gegensatz dazu verhalten sich die Anzahl der Rekursionen und Iterationen, die zum Berechnen der Lösung notwendig sind, wie erwartet und vergleichbar mit den Ergebnissen der Messungen mit linearen Kostenfunktionen.

Laufzeit Genau wie bei der Betrachtung linearer Kostenfunktionen ist die Laufzeit in Abhängigkeit der Problemkomplexität interessant. Neben der Anzahl der Klienten, die bereits im linearen Fall analysiert wurde, trägt auch die Anzahl der Dienste als Parameter der Komplexität bei. Dies wurde bereits bei den theoretischen Betrachtungen der Laufzeit im Abschnitt 5.3.1 erwähnt und soll im folgenden bestätigt werden.

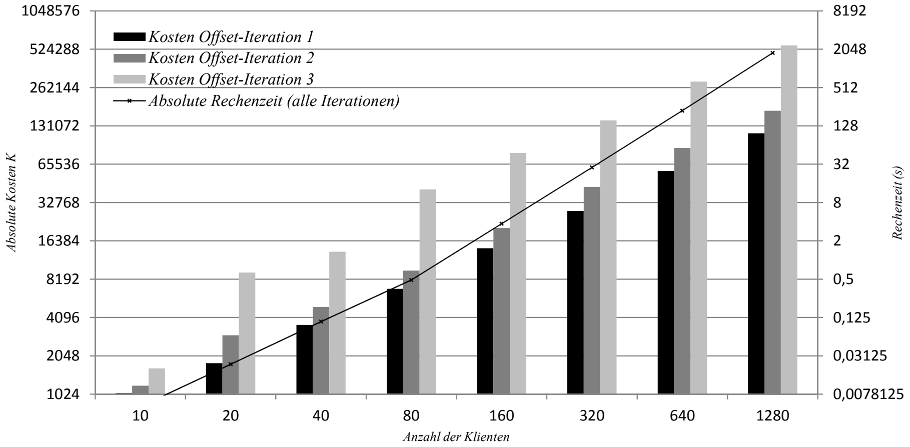
Zunächst wird wieder der Einfluß der Klientenanzahl untersucht. Um die Vergleichbarkeit zu bisherigen Ergebnissen dieses Abschnittes zu gewährleisten, wird auch in dieser Konfiguration von 3 Diensten ausgegangen. Um die Anzahl der Klienten als ganzzahlige Vielfache der Funktionsgruppen zu garantieren, wurde der Initialwert für die erste Messung auf 10 Klienten gesetzt und mit jeder Messung verdoppelt. Details der Konfiguration sind in Tabelle 5.10 zu finden.

Die Ergebnisse für die Ausführung sind in der Abbildung 5.9 jeweils für die Lösung mittels der Heuristik und mittels Matlab (Ausgleichsrunden) aufgeführt. Vergleiche können auch zu den Ergebnissen der linearen Kostenfunktionen in den Diagrammen der Abbildung 5.5 gezogen werden.

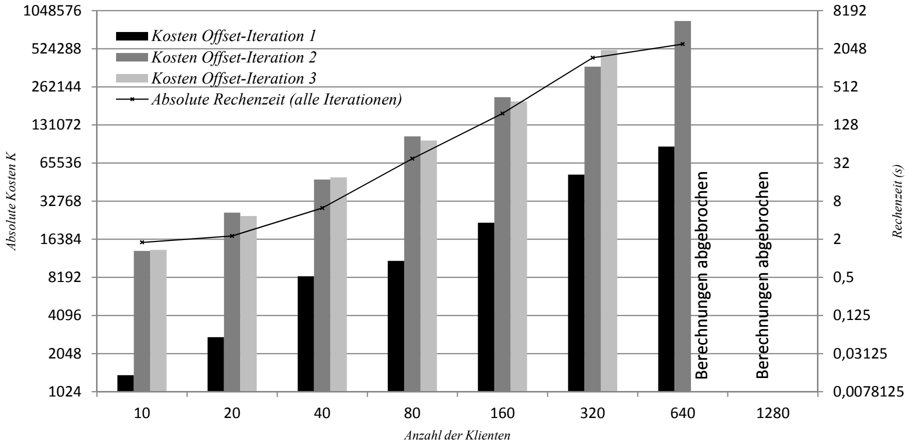
Die absoluten Kosten der gefundenen Lösungen sind nur in Relation der Heuristik- und Matlablösung interpretierbar. Genau wie in den bisherigen Messungen ist ersichtlich, daß die Heuristik deutlich günstigere Lösungen findet. In einigen Fällen liegt zwischen beiden Lösungen ein Faktor von 10. Auffällig ist, daß die Kosten nach dem ersten Schritt der verschiedenen

Parameter	Wert	Ergänzung
# Klienten	10, 20, ..., 1280	$= m$
# Dienste	3	
# Kostenfunktionsgruppen	5	
$average$	$offset/m$	
# Schritte	3	
$offset$	$\{60, 30, 2\} \cdot (m/10)$	
# Messungswiederholungen	1	

Tabelle 5.10.: Konfiguration der Messung 9 - Kosten und Ausführungszeiten



(a) Heuristik



(b) Matlab (ausgeglichen gerundet)

Abbildung 5.9.: Messung 9 - Kosten und Ausführungszeiten der Partitionierung

Messungen relativ gleich sind, während sie dann im zweiten Schritt bei Matlab meist deutlich ansteigen.

Bei den zugehörigen Berechnungszeiten zeigt sich in der ersten Messung mit wenigen Klienten zunächst das gleiche Verhalten wie bei den linearen Funktionen. Während die Heuristik nur wenige Millisekunden zur Lösung benötigt, sind es bei Matlab ca. 2 Sekunden. Die Ursachen hierfür wurden bereits diskutiert.

Die Heuristik verhält sich qualitativ ähnlich wie im Fall der linearen Funktionen. Mit jeder Verdopplung der Klientenzahl steigt die Laufzeit zum Finden einer Partitionierung um das 6- bis 8-fache. Auch hier gilt, daß es sich um die Gesamtlaufzeit aller 3 Schritte handelt, die in jedem Meßpunkt durchzuführen sind. Quantitativ verdoppelt sich die Laufzeit der Heuristik von der Messung 5 (lineare Kostenfunktionen) zur Messung 9. Da die Zahl der Dienste linear in die theoretische Komplexitätsbetrachtung der Heuristik eingeht, ist dieses Verhalten zu erwarten.

Bei der Partitionierung von Matlab fällt zunächst auf, daß das Problem fehlenden Speichers diesmal sehr viel eher auftritt. Bereits im letzten Schritt mit 640 Klienten tritt der Fehler, wie im Listing 5.6 beschrieben, auf. Zurückzuführen ist dies auf die Tatsache, daß die generierte Anzahl der Optimierungsvariablen in der Zielfunktion als Produkt aus Klientenanzahl und Dienstanzahl gebildet wird (Vgl. Listing 5.3) und damit der benötigte Speicher zur Lösung nicht nur mit der Klienten- sondern auch mit der Dienstanzahl steigt.

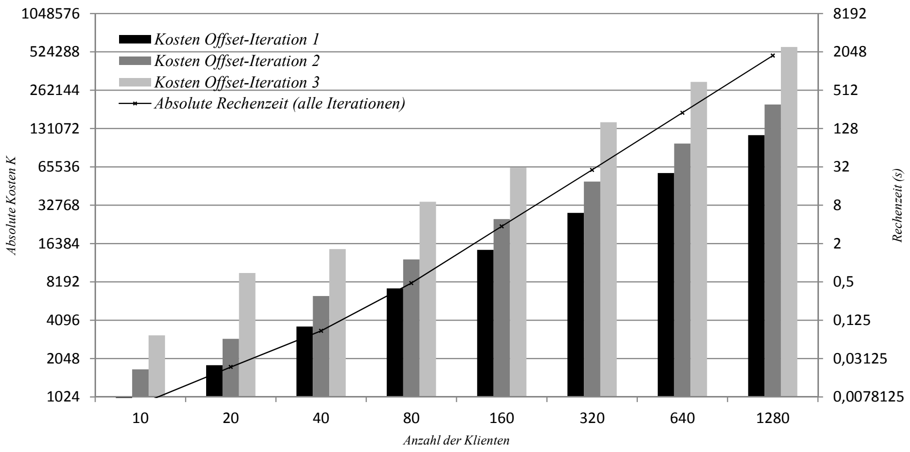
Mit der geringen Anzahl an Meßwerten ist eine belastbare Aussage über das Wachstum der Matlab-Laufzeit schwierig. Wird der Wert für 640 Klienten ignoriert - er wird durch die fehlende Berechnung des dritten Schritts verfälscht - ist zwar kein linearer aber auch kein exponentieller Anstieg zu erkennen. Zur Berechnung eines Polynoms, welches den Anstieg beschreibt, werden mehr Werte vor allem bei höheren Klientenzahlen benötigt. Die Werte für die geringe Klientenzahlen werden vermutlich durch ein internes Setup beeinflusst, wie bei den Ergebnissen zu linearen Kostenfunktionen bereits diskutiert wurde.

Auf Grund der sequentiellen Optimierung der einzelnen Dienste durch die Heuristik kann auch die Sequenz der einzelnen Dienste die Ergebnisse beeinflussen. Entsprechend wurde die Konfiguration 9 (Tabelle 5.10) für eine weitere Messung genutzt, in der die Dienstreihenfolge in der Definition geändert wurde. Dienst 3 wurde zu Dienst 1 und umgekehrt. Entsprechend wird von der Heuristik zunächst die Partitionierung für den Dienst ausgeführt, dessen Kosten in der Gleichung A.11 definiert sind.

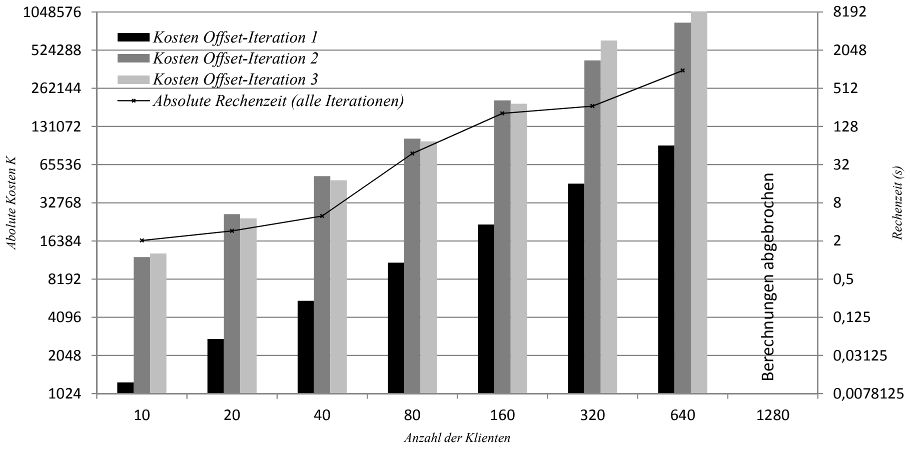
Da die Kosten der einzelnen Dienste auch von den Lasten anderer Dienste abhängen, sind Kostenänderungen bei der Partitionierung mittels der Heuristik zu erwarten. In Matlab hingegen wird das komplette Optimierungsproblem zur Lösung übergeben und berechnet, so daß ein Einfluß der Dienstreihenfolge nicht zu erwarten ist.

Die Ergebnisse der Paritionierung sind in der Abbildung 5.10 zu finden. Bis auf den ersten Lauf mit 10 Klienten, wo die Kosten bei der Partitionierung durch die Heuristik auf rund das doppelte steigen, existieren kaum nennenswerte Differenzen. Offensichtlich hat die Dienstreihenfolge auf diese Konstellation der Kostenfunktionen nur einen geringen Einfluß.

Auffälliger ist hingegen die Auswirkung auf die Partitionierung in Matlab. Zum einen ist in diesem Fall die Berechnung des dritten Schrittes mit 640 Klienten noch möglich. Zum anderen liegt die Berechnungszeit für 320 Klienten um Größenordnungen unter der der ursprünglichen Dienstreihenfolge. Die Kosten der gefundenen Lösung sind jedoch weitestgehend identisch. Offensichtlich gibt es Konstellationen, in denen Matlab schnell gegen eine günstige Lösung konvergiert. Da jedoch bereits in der nächsten Messung die Berechnungszeit wieder deutlich ansteigt, ist es schwierig, diesen Effekt vorherzusagen oder gar zu nutzen.



(a) Heuristik



(b) Matlab (ausgeglichen gerundet)

Abbildung 5.10.: Messung 9 - Kosten und Ausführungszeiten der Partitionierung bei geänderter Sequenzialisierung der Dienste

Abschließend soll eine Messung den direkten Einfluß der Klientenanzahl auf die Komplexität und die Laufzeit prüfen. Wie erwähnt sollte die Zahl der Dienste als Linearfaktor in die resultierende Komplexität der Heuristik eingehen.

Die Konfiguration ist in Tabelle 5.11 zu sehen. Zur Messung des Parameters wurden die bisher genutzten Dienste 1 und 2 je nach Anzahl der Dienste kopiert. Dienst 3 wird nicht genutzt. Die Laständerung beträgt in jedem Schritt für ungeradzahlige Dienste 960 Einheiten

Parameter	Wert	Ergänzung
# Klienten	160	$= m$
# Dienste	2, 4, 8, 16,... 128	$= n$
# Kostenfunktionsgruppen	5	
$average$	$offset/m$	
# Schritte	3	
$offset$	$\{960, 480\}^n$	
# Messungswiederholungen	1	

Tabelle 5.11.: Konfiguration der Messung 10 - Ausführungszeiten

und für geradzahlige Dienste 480. Gemessen wird für 160 Klienten, da in diesem Bereich die Berechnungszeiten der Heuristik bei mehreren Sekunden liegen und damit meßbar sind.

Abbildung 5.11 verdeutlicht die Ergebnisse der Messung. Alle Achsen sind logarithmisch abgetragen, so daß das lineare Verhalten ersichtlich ist. Sowohl die Anzahl der Iterationen als auch die daraus resultierende absolute Ausführungszeit verdoppeln sich jeweils mit einer Verdopplung der Dienstzahl. Da die Anzahl der Rekursionen von der Last und der Anzahl der Klienten beeinflußt wird und diese für die ungeradzahligen und geradzahligen Dienste gleich sind, bleibt die Anzahl der Rekursionen unabhängig von der Anzahl der Dienste ebenso gleich.

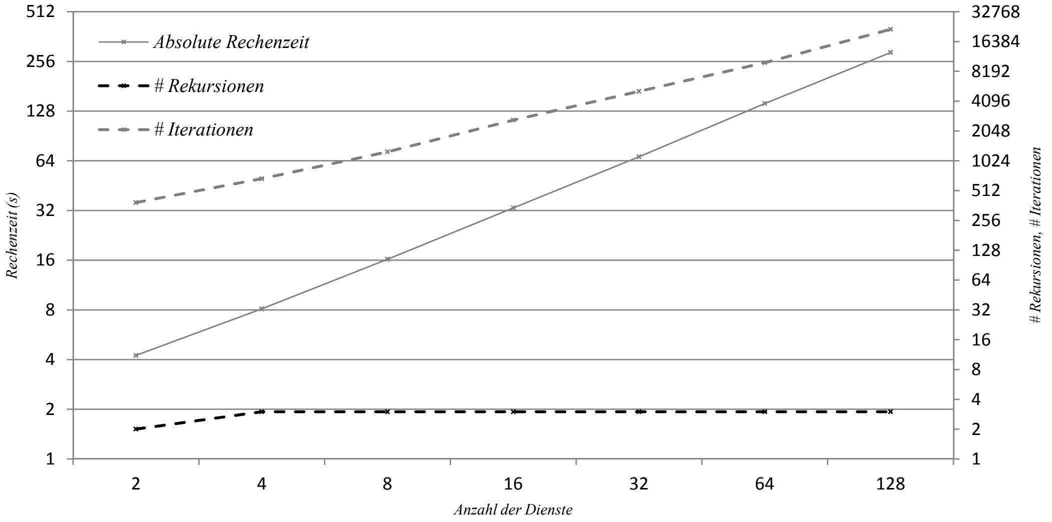


Abbildung 5.11.: Messung 10 - Einfluß der Dienstanzahl auf Rechenzeit

Nach ersten Versuchen wurde die Optimierung mittels Matlab für diese Konfiguration nicht genutzt. Bereits bei zweistelligen Dienstzahlen wächst hier die absolute Berechnungszeit auf Werte von mehreren Stunden. Es liegt die Vermutung nahe, daß die gewählte Form der

Funktions- und Parameterübergabe für derartig große Optimierungsprobleme ungeeignet ist. Beispielsweise wird die erzeugte Zielfunktion (Vgl. Abschnitt 5.3.2) bereits im Fall von 32 Diensten mehrere Megabyte groß. Eine Änderung der Parameterübergabe ist kaum möglich. Allenfalls der Aufruf des Solvers kann von JMI auf andere APIs umgestellt werden (Vgl. Abschnitt 5.3.2 über das Evaluierungssystem). Die Quantität der zu übergebenden Parameter bliebe in diesem Fall jedoch gleich, so daß kaum mit nennenswerten Verbesserungen zu rechnen ist. Eine deutliche Vereinfachung wäre allenfalls durch Einschränkungen bei der dienstübergreifenden Kostenabhängigkeit denkbar, was jedoch die Lösung zu stark in der allgemeinen Anwendbarkeit einschränken würde.

5.4. Zusammenfassung und Schlußfolgerung

Das zurückliegende Kapitel beschäftigte sich mit dem Testpartitionierungsproblem und es wurden zwei Ansätze zur Lösung vorgestellt. Hierzu wurde das Problem zunächst formal als Optimierungsaufgabe definiert. Auf Grund der Anforderungen, die im letzten Kapitel diskutiert wurden, ergibt sich ein nichtlineares Optimierungsproblem unter Nebenbedingungen.

Zur Lösung des Problems wurden zwei Verfahren vorgestellt: der fmincon-Solver von Matlab sowie eine speziell für das Problem der Testautomatisierung zugeschnittene Heuristik. Diese Heuristik verteilt die zu generierende Last eines Dienstes zunächst gleichmäßig auf alle verfügbaren Klienten und iteriert dann Optimierungen auf dieser Verteilung durch Verschieben entsprechender Zuteilungen. All dies wird sequentiell für alle Dienste ausgeführt. Gibt es für einen Dienst Abweichungen zwischen der zu generierenden und der verteilten Last, wird das Verfahren mit der entsprechenden Differenz rekursiv gerufen.

Zur Bestimmung der Leistungsfähigkeit und dem Vergleich beider Lösungen wurden diese implementiert. Ein Evaluationssystem führte dann die Testpartitionierung anhand verschiedener Konfigurationen und in Abhängigkeit relevanter Parameter sowohl unter der Nutzung der Heuristik als auch mit Matlab aus. Die Ergebnisse wurden gespeichert, verglichen und entsprechend ausgewertet.

In Auswertung der zahlreichen Messreihen wird deutlich, daß die Heuristik der Matlab-Lösung in Qualität und Leistung fast immer überlegen ist. Insbesondere bei den praxisrelevanten nichtlinearen Kostenfunktionen sind die erzeugten Lösungen bezüglich der Kosten besser und werden häufig in Bruchteilen der Zeit im Vergleich zur Matlab-Lösung errechnet.

Allerdings steigen die absoluten Zeiten zur Berechnung einer Lösung auch unter Nutzung der Heuristik schnell auf ein erhebliches Maß an. Wird die Anzahl der Klienten mit der Anzahl der Dienste multipliziert, steigt die Berechnungszeit ab einem Wert von rund 1000 für jede Testpartitionierung auf einen zweistelligen Sekundenwert. Entsprechende Verzögerungen sind bei einer Anwendung zu Berücksichtigen.

Aufgrund der deutlich besseren Ergebnisse wird in den folgenden Kapiteln nur noch die Heuristik als Lösungsverfahren für die Testpartitionierung verwendet. Für Einsatzszenarien wie das Mobilfunkbeispiel ist die Handhabung von wenigen hundert Klienten und einer einstelligen Anzahl von Diensten praktikabel.

6. Implementierung

In den zurückliegenden Kapiteln wurde das allgemeine Konzept und die daraus abgeleitete Architektur der Testbench sowie alllgemeine Ansätze zur Lösung des algorithmischen Problems der Testpartitionierung vorgestellt.

Im Rahmen der Arbeit wurde auch eine prototypische Implementierung der Kernkomponenten entwickelt. Grundlage bildet das eingangs genannte Beispiel eines zellulären Mobilfunknetzes - in diesem Falle ein UMTS-Netz. Das Ziel ist die Realisierung einer Testbench, die Mobiltelefone und Computer mit Datennetzmodem als Klienten nutzt, um das bestehende UMTS-Netz durch steuerbare Lasten zu validieren.

Die Implementierung entstand ursprünglich in Kooperation mit einem industriellen Netzinfrastrukturentwickler und dem Ziel, die Testbench bei Feldtests in einem realen Mobilfunknetz einzusetzen. Nach anfänglichen Tests in einem realen Mobilfunknetz wurde zur detaillierten Evaluierung der Funktions- und Leistungsfähigkeit des Konzeptes die bestehende Implementierung in einen Simulator integriert und Ergebnisse durch entsprechende Simulationsläufe generiert.

6.1. Beispielsystem

Wie bei der Beschreibung des Beispiels erläutert, wird das komplette Mobilfunknetzwerk in diesem Beispiel zum System Under Test. Die interne Architektur und Organisation des Netzes ist dabei unerheblich. Wichtiger sind vielmehr die Definition der zu testenden Dienste und die zugehörige Auswahl sowie Adaption der Klienten.

Die Arbeiten an der Testbench begannen im Jahr 2008. Folgende zu dieser Zeit relevanten Dienste wurden ausgewählt, um mittels der Testbench getestet zu werden:

1. Sprachanruf
2. Short Message Service (SMS)
3. Multimedia Message Service (MMS)
4. TCP-basierter Upload
5. TCP-basierter Download
6. UDP-Stream

Dienst	Aktion	Parameter
Sprachanruf	Anruf starten	Dauer, Empfängerrufnummer
SMS	SMS versenden	Textlänge, Empfängerrufnummer
MMS	MMS versenden	Anhanggröße, Empfängerrufnummer
TCP-Upload	Datei hochladen	Dateigröße, Zieladresse
TCP-Download	Datei runterladen	Quelladresse (implizit Dateigröße, Protokoll)
UDP-Stream	UDP Datenpakete senden	Datenrate, Paketgröße, Zieladresse

Tabelle 6.1.: Übersicht über die Elementaraktionen der Testbench, Ausbaustufe 1

Auch über die Integration weiterer Dienste, wie Videoanrufe, wurde nachgedacht. Jedoch wurden die Ideen aufgrund mangelnder Realisierung auf den Mobiltelefonen verworfen.

Die Umsetzung erfolgte in zwei Phasen. In einem ersten Schritt sind Implementierungen der Testservermoduls und der Testklienten für verschiedene Plattformen entstanden. Diese Variante, im folgenden Ausbaustufe 1 genannt, erlaubt Testingenieuren, mittels einer graphischen Oberfläche einzelne Elementaraktionen auf den Testklienten auszulösen und zu überwachen. Die Ergebnisse der jeweiligen Elementaraktionen werden in einer Datenbank gespeichert und in der graphischen Oberfläche dargestellt. Ein Überblick über die Testbench dieser Ausbaustufe ist in Abbildung 6.1 dargestellt.

In dieser Ausbaustufe ist folglich keine Automatisierung vorgesehen und somit auch die Definition der Last nicht notwendig. Vielmehr sind die Elementaraktionen und deren Parameter relevant. Eine entsprechende Übersicht der Aktionen für die einzelnen Dienste sind in Tabelle 6.1 zu finden.

Gestartet werden die Elementaraktionen mittels der graphischen Oberfläche. Benötigte Parameter, wie Rufnummern oder URLs, werden vom Bediener manuell vorgegeben. Die Oberfläche erlaubt auch das Erstellen von Skripten mit festem Zeitverhalten, so daß ein minimales Maß an Automatisierung erreicht wird.

Die Ausbaustufe 1 wurde im Rahmen der erwähnten Industriekooperation umgesetzt und auch in einem realen UMTS-Mobilfunknetzwerk getestet. Im Vergleich zum Validierungskonzept, wie es in den zurückliegenden Kapiteln vorgestellt wurde, sind jedoch deutliche Einschränkungen hinzunehmen. Insbesondere die Möglichkeit, auf Änderungen in der Klientenstruktur dynamisch und automatisiert zu reagieren, fehlt in der Ausbaustufe 1.

Zur Überprüfung von Funktionsfähigkeit und Kernparametern des Konzeptes wurde eine Ausbaustufe 2 umgesetzt, die auch das Testautomatisierungsmodul auf Basis der in Abschnitt 5.2.3 vorgestellten Heuristik enthält.

Details zur Implementierung der einzelnen Module sowie die Integration der Simulationsumgebung werden in den folgenden Abschnitten beschrieben.

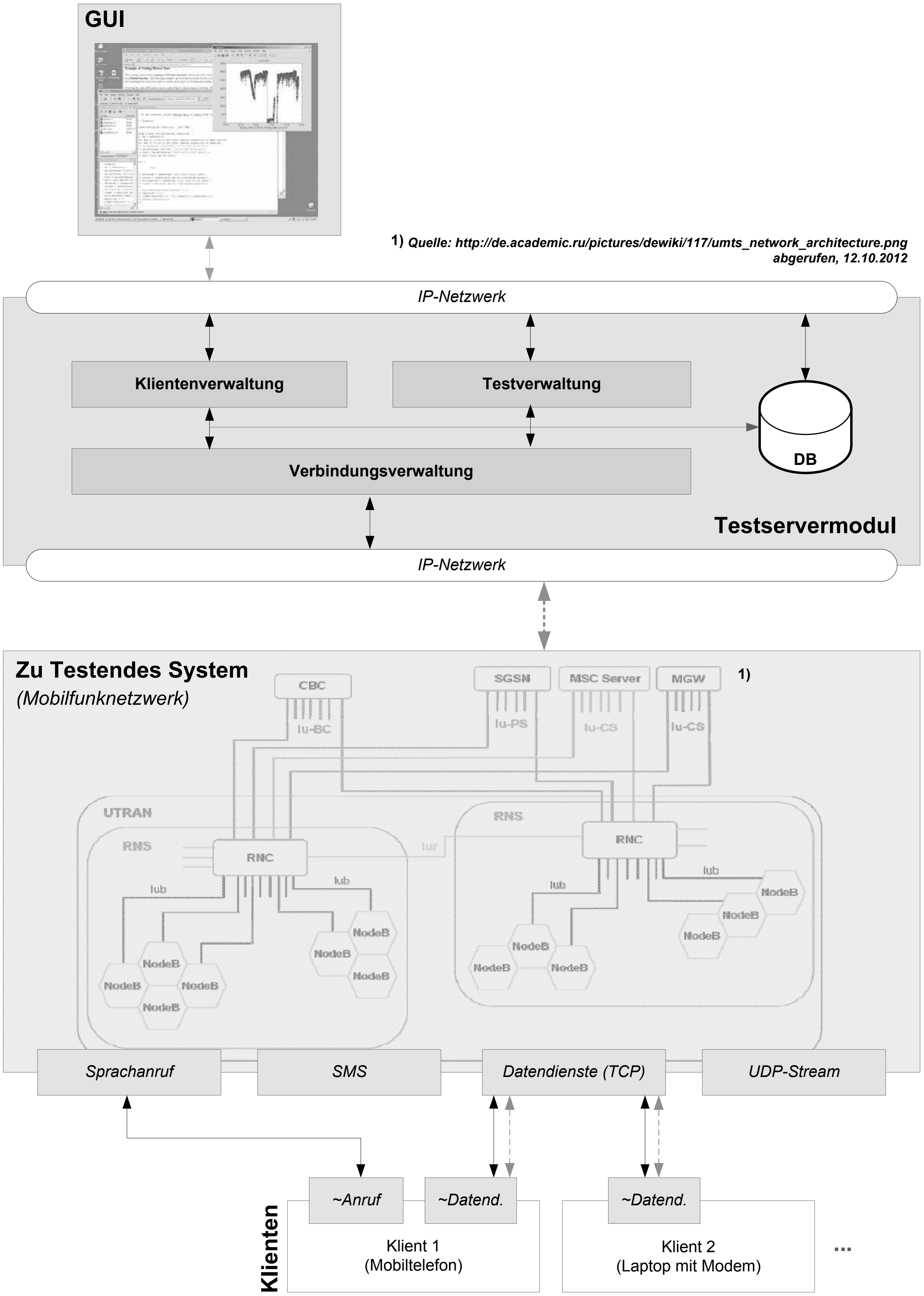


Abbildung 6.1.: Testbench-Adaption für das Beispiel Mobilfunknetzwerk, Ausbaustufe 1. Die Kommunikation zwischen Testservermodul und Klienten erfolgt über die TCP-Datendienste des SUT.

6.2. Architektur

Die Architektur, die für Ausbaustufe 1 umgesetzt wurde, ist in Abbildung 6.2 dargestellt. Enthalten sind, wie im Konzept angegeben, das Testservermodul sowie die Klienten mit den Testklienten. Nicht enthalten ist das Automatisierungsmodul. Stattdessen kommt eine graphische Nutzeroberfläche (GUI) zum Einsatz, die das Auslösen von Elementaraktionen erlaubt sowie den laufenden Test und dessen Ergebnisse visualisert.

Alle Komponenten sind über TCP/IP-Netwerke verbunden. Dies trifft insbesondere auf die Verbindungen zwischen dem Testservermodul und den Klienten zu. Hier werden Dienste des eigentlich zu testenden Systems genutzt, um die Kommunikation mit den Testklienten zu ermöglichen. In diesem speziellen Falle werden hierfür sogar Dienste genutzt, für die im Laufe des Tests Lasten erzeugt werden sollen - nämlich die Dienste für „TCP-basierten Upload“ und „TCP-basierten Download“.

Auf diese Weise lassen sich im Vergleich zum Aufbau eines dedizierten Kommunikationsnetzes, bspw. über ein zweites Mobilfunk- oder WLAN-Netz, deutlich Kosten und Aufwände sparen. Die Umsetzung führt jedoch auch zu einigen Einschränkungen und Besonderheiten, die für die Implementierung und auszuführende Tests beachtet werden müssen. Das naheliegendste Problem ist, daß ein Klient, der die Mobilfunkverbindung verliert, auch für das Testservermodul nicht mehr erreichbar ist. Entsprechende Ping-, Timeout- und Pollingmechanismen müssen realisiert werden, um dieses Ereignis zu erkennen bzw. einen Klienten wiederzuverbinden. Durch die Nutzung des Mobilfunknetzwerkes treten desweiteren zahlreiche Probleme durch Netzwerksicherheitsrestriktionen, wie Firewalls, private Netzwerke usw., auf.

Zwischen den einzelnen Komponenten werden Informationen über XML-Nachrichten ausgetauscht. Zur Verringerung des Datenaufkommens zwischen Testservermodul und Klienten werden die Nachrichten dort in WAP Binary XML (WBXML) serialisiert (siehe [2]). Hierbei werden die häufig sehr großen Tag- und Attributnamen durch kurze, binäre Schlüssel ersetzt. Eine Tabelle auf Seiten der Sender und Empfänger erlaubt es, die Echtnamen umzuwandeln bzw. wiederherzustellen.

Um einem Mißbrauch, insbesondere der Klienten vorzubeugen, werden alle Nachrichten mit Signaturen versehen. Dadurch ist es dem Empfänger möglich, die Herkunft der Nachrichten zu überprüfen.

Die Planungen für Ausbaustufe 1 zielten auf den parallelen Einsatz von ca. 150 bis 200 Klienten auf den drei großen Campus der Technischen Universität Chemnitz ab.

6.2.1. Testklient

Aus organisatorischen und technischen Gründen war es das Ziel, handelsübliche Mobiltelefone bzw. Computer mit UMTS-Modems als Klienten einzusetzen. Um die Kosten für die Anschaffung

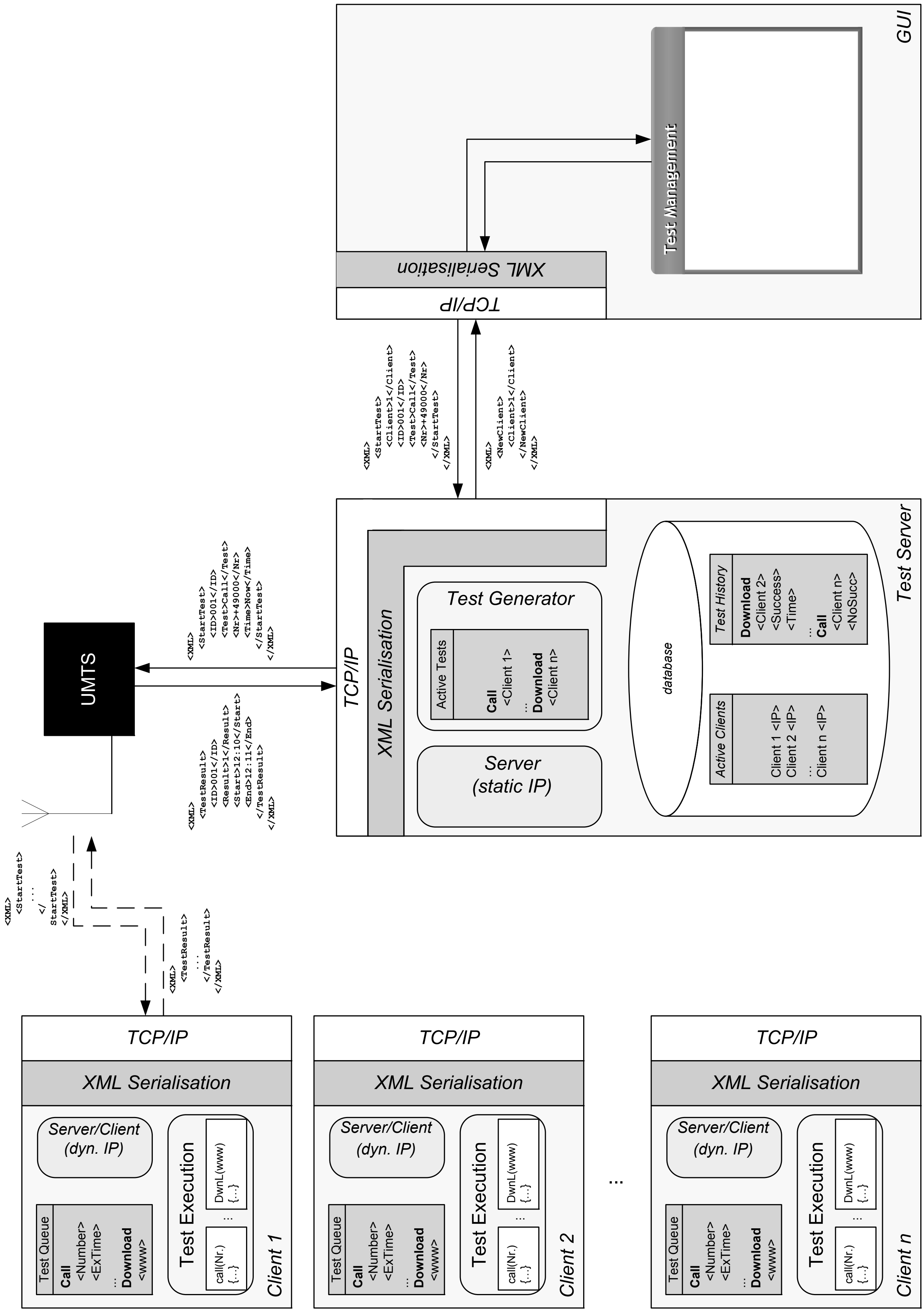


Abbildung 6.2.: Detailierte Systemstruktur

einiger hundert Endgeräte in vertretbaren Größenordnungen zu halten, sollten Mobiltelefone der unteren Preiskategorien zum Einsatz kommen.

Zum Konzeptionszeitpunkt der Testbench im Jahr 2008 stellte die Programmierbarkeit der preisgünstigen Mobiltelefone die größte Hürde dar. Viele Hersteller setzten auf proprietäre Betriebssysteme, für die keine Entwicklungsumgebungen verfügbar waren. Die einzige Möglichkeit, eigene Applikationen auf die Geräte zu bringen, lag in der Nutzung von Java. Speziell für „Ressourcenbeschränkte Geräte“, wie Mobiltelefone, PDAs u.ä., wurde von Java 2 (Java 1.2) eine Micro Edition abgeleitet (J2ME, Vergleich [76], S. 2). Diese enthält neben entsprechenden virtuellen Maschinen für die einzelnen Plattformen auch Bibliotheken zur Nutzung der wichtigsten Funktionen. Im Vergleich zu herkömmlichen Java Standard Editionen für Desktopsysteme unterscheiden sich vor allem die Mechanismen zur Ausgabe auf den kleinen Displays sowie zur Eingabe mittels Telefontastaturen und Touchscreens.

J2ME wurde entsprechend gewählt, den Testklienten für Mobiltelefone umzusetzen. Da auch Laptops als Klienten zum Einsatz kommen, wurde desweiteren eine Portierung für die Standard Edition von Java (in diesem Falle in Version 1.6) erstellt. Da sich J2ME und J2SE auch in den Kommunikationsbibliotheken unterscheiden, wurde der Zugriff auf diese sowie auf einige spezielle Hardwarekomponenten entsprechend adaptiert.

Details der Struktur eines Testklienten sind in Abbildung 6.3 dargestellt. Für die Funktion sind vor allem das *ConnectionManagment* und die *TestExecutionEngine* entscheidend. Zur Nutzung der eigentlichen zu testenden Dienste werden die Java-internen Bibliotheken (API) verwendet, die wiederum auf entsprechende betriebssystemeigene Bibliotheken zugreifen. Je nach Verfügbarkeit umfassen diese APIs Zugriffsmöglichkeiten auf das Anrufsystem (Sprachanruf), das Nachrichtenmanagement (SMS, MMS) und TCP/IP-Sockets.

Das ConnectionManagement ist für die Verbindung und den Nachrichtenaustausch mit dem Testservermodul zuständig. Aufgrund der angesprochenen Sicherheitsmechanismen des SUT hat es sich als praktikabel erwiesen, wenn der Klient versucht, als TCP-Client eine Verbindung zum Testservermodul aufzubauen. Dies wird nach der Initialisierung der Anwendung bzw. nach dem Verlust der Verbindung in zufälligen Zeitabständen versucht. Empfangene Nachrichten werden mittels der Signatur auf die Herkunft geprüft und in XML dekodiert.

Empfangene und auch zu sendende Nachrichten werden in Warteschlangen vorgehalten, so daß die einzelnen Komponenten des Testklienten asynchron arbeiten können. Dies ermöglicht die Entkopplung der einzelnen Aktivitäten durch die Nutzung von Threads. Außerdem können Nachrichten für den Fall von Verbindungsproblemen gepuffert werden.

Die TestExecutionEngine verwaltet alle anstehenden bzw. laufenden Elementaraktionen. Wird eine neue Aktion empfangen, wird geprüft, inwieweit Vorbedingungen, wie zum Beispiel die Startzeit, erfüllt sind und auf den entsprechenden Dienst zum Zeitpunkt überhaupt zugegriffen werden kann. Ist dies der Fall, wird die entsprechende Aktion als eigener Thread gestartet. Für die verschiedenen Dienste existieren jeweils eigene Implementierungen, für die dann entsprechende Instanzen angelegt werden. Da diese Implementierungen auf die Dienst APIs der Virtuellen

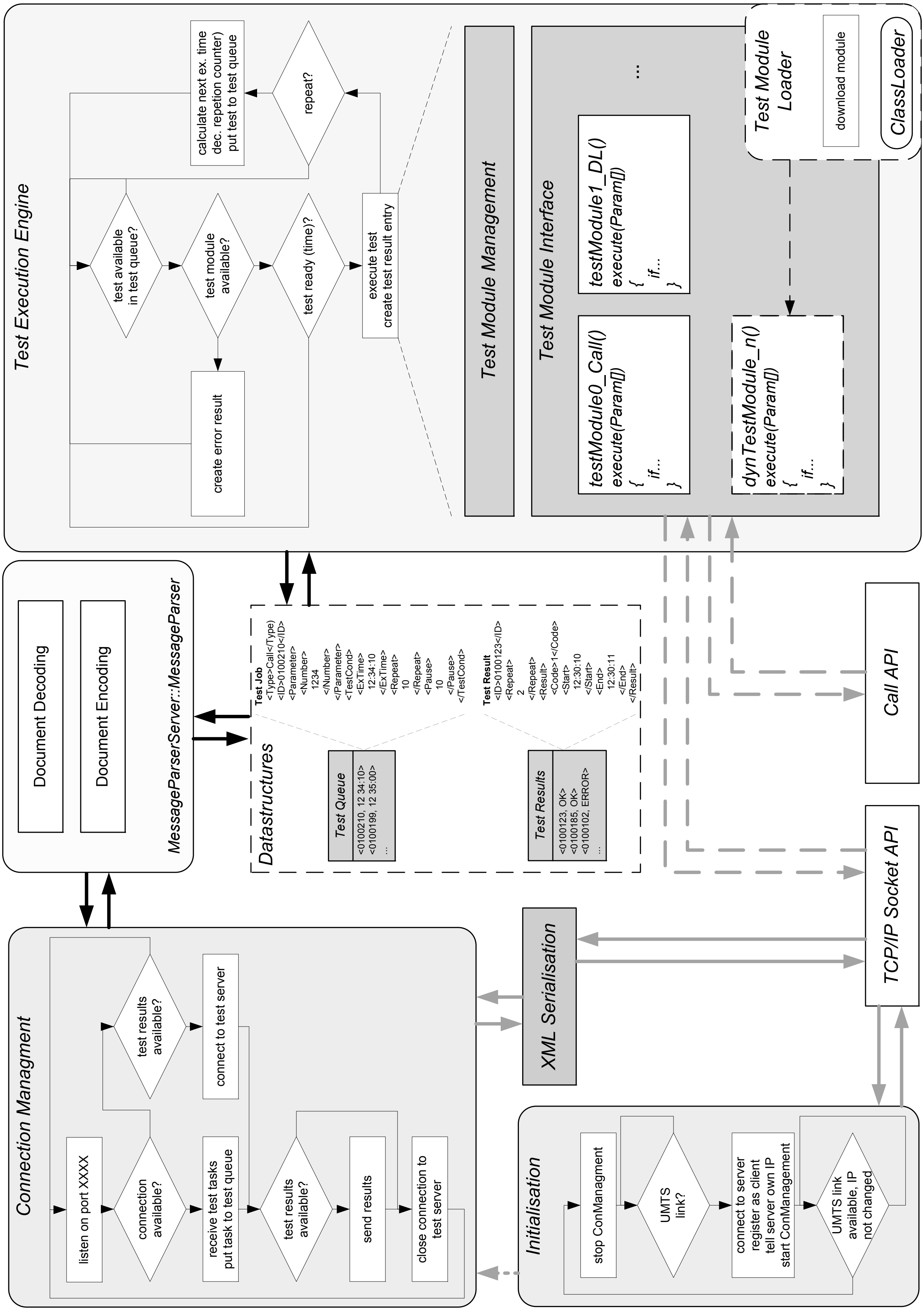


Abbildung 6.3.: Implementierungsdetails Testklient

Maschine bzw. des Gerätes zugreifen, unterscheiden sie sich u. U. auf verschiedenen Plattformen und müssen dementsprechend gerätespezifisch angepaßt werden.

Ist eine solche Elementaraktion beendet, generiert der zugehörige Thread einen Bericht, der Informationen über Erfolg oder Mißerfolg sowie testspezifische Parameter enthält. Dieser Bericht wird von der TestExecutionEngine in die Sendewarteschlange geschrieben und von dort weiter an das Testservermodul übertragen. Ggf. wird die Elementaraktion nach einer definierten Zeit wiederholt.

6.2.2. Testservermodul

Das Testservermodul ist ebenfalls in der Standardversion von Java (1.6) realisiert und läuft auf einem Linux-Server. Die Speicherung sämtlicher Einstellungen und der Testergebnisse erfolgt in einem externen Datenbankmanagementsystem. In diesem Falle kommt MySQL zum Einsatz, welches auf dem gleichen Server läuft. Zur Kommunikation mit den Klienten ist es zwingend erforderlich, daß der Server Zugriff auf das Mobilfunknetzwerk hat sowie durch die Mobiltelefone erreichbar ist. Der Netzwerkname bzw. die IP muß den Klienten bekannt sein.

Hauptaufgaben des Testservermoduls ist die Verwaltung der Klienten bzw. der Kommunikation zu diesen sowie die Verwaltung der Tests. Entsprechend ergibt sich die Struktur, die in Abbildung 6.4 dargestellt ist.

Nach der Initialisierung des Testservermoduls stellt das Client-Management einen TCP-Server auf einem definierten Port zur Verfügung. Zu diesem Port können sich Testklienten verbinden. Wird eine Verbindung aufgebaut, werden Identifikationsmerkmale des Klienten mit bekannten Geräten verglichen. Aus Sicherheitsgründen wird die Verbindung zu unbekannten Klienten wieder beendet. Andernfalls wird der Klient zur Datenstruktur ActiveClients hinzugefügt bzw. bei Neuverbindung dessen Zustand aktualisiert. Auch der zugehörige Socket wird gespeichert, so daß jederzeit Nachrichten mit dem Klienten ausgetauscht werden können.

Um Veränderungen in der Erreichbarkeit zu erkennen, werden alle verfügbaren Klienten regelmäßig angepingt. Werden die Ping-Nachrichten vom Klienten nicht bestätigt, wird dieser aus der Liste der aktiven Klienten entfernt. Parallel zu den Klienten instanziert das Client-Management auch einen TCP-Server auf einem definierten Port. Zu dem kann sich die graphische Nutzeroberfläche oder in Ausbaustufe 2 auch das Automatisierungsmodul als TCP-Client verbinden. In Ausbaustufe 1 wird der Parallelbetrieb mehrerer Nutzeroberflächen unterstützt.

Wie auch im Testklienten werden sämtliche empfangene Nachrichten von den Klienten bzw. der Oberfläche in Warteschlangen abgelegt, so daß die einzelnen Komponenten des Testservermoduls unabhängig voneinander und als eigene Threads laufen können.

Wird von einer verbundenen Nutzeroberfläche eine Nachricht zum Start eines Elementarauftrages empfangen, prüft der Test-Starter zunächst, ob dieser Elementarauftrag überhaupt gestartet werden kann. Dazu wird neben syntaktischen Angaben, bspw. eine korrekte Rufnummer, geprüft, ob der

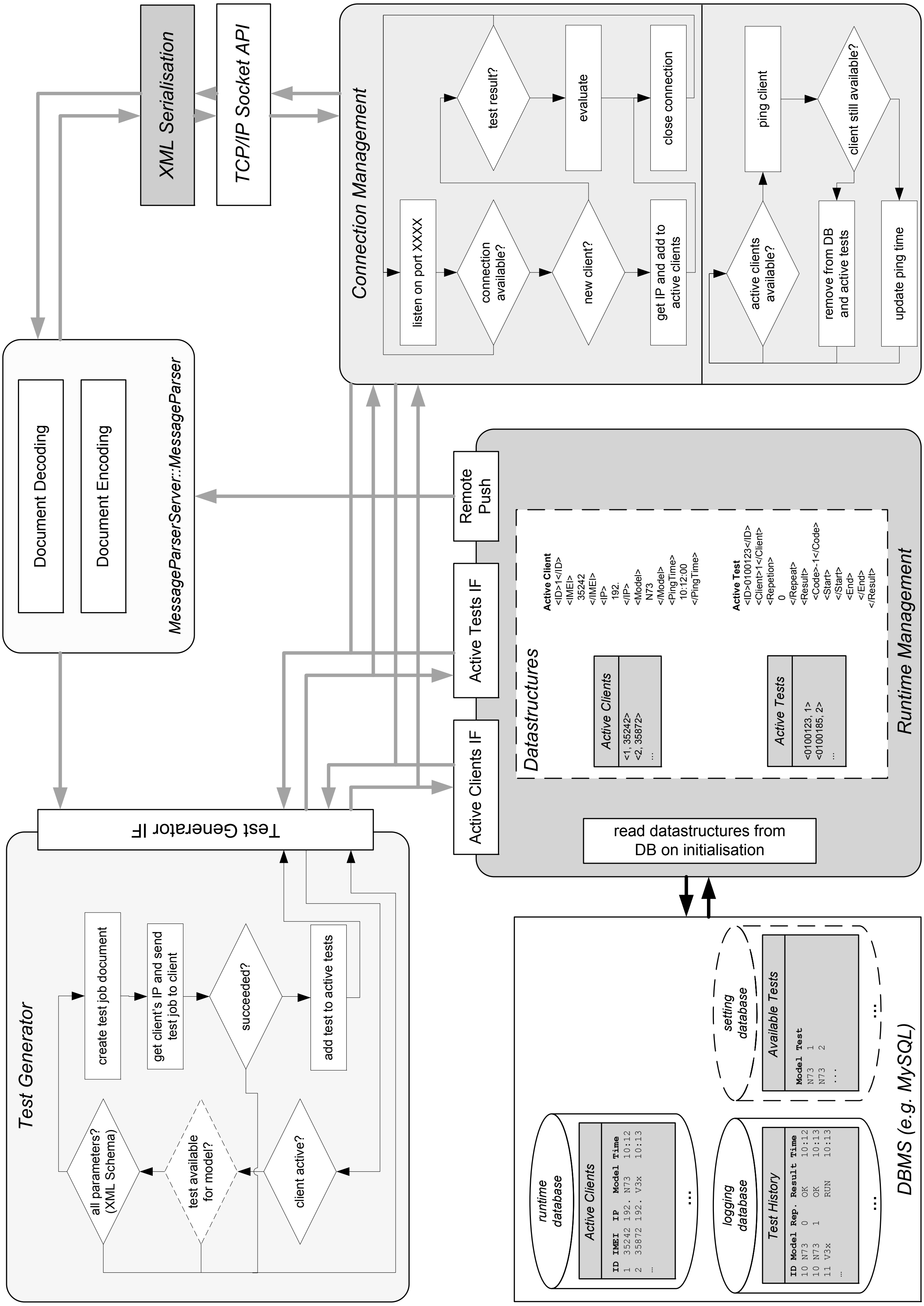


Abbildung 6.4.: Implementierungsdetails Testserver

betreffende Klient verfügbar und überhaupt technisch in der Lage ist, den entsprechenden Dienst zu nutzen. Sind alle Vorbedingungen erfüllt, wird eine entsprechende Nachricht an den Klienten vorbereitet und gesendet.

Bei erfolgreicher Übermittlung wird eine Datenstruktur Test mit allen zugehörigen Informationen angelegt und einer Liste aller aktiver Tests (*ActiveTests*) hinzugefügt. Nach Empfang eines Berichts von einem der Testklienten wird der Status des Tests entsprechend aktualisiert. Ist er beendet, wird der Test aus *ActiveTests* entfernt und in der Datenbank archiviert. Mit Hilfe von *ActiveTests* kann damit während der Laufzeit die aktuelle Last bzw. im Falle von Ausbaustufe 1 die Anzahl der laufenden Elementaraktionen ermittelt werden.

Um im Falle von Problemen Informationen zu laufenden Tests wiederherstellen zu können, werden die Datenstrukturen *ActiveClients* und *ActiveTests* während der Laufzeit mittels des Object-Relation-Mappers *Hibernate*[1] ständig mit der Datenbank synchronisiert.

6.2.3. Nutzeroberfläche

Die Erzeugung der Elementaraktionen erfolgt in Ausbaustufe 1 ausschließlich über die graphische Benutzeroberfläche. Auch diese ist mittels Java Standard Edition (Version 1.6) implementiert und kommuniziert über TCP-Sockets mit dem Testservermodul.

Die Oberfläche zeigt die verfügbaren Klienten sowie die laufenden Tests an. Zum Start einer Elementaraktion muß der Nutzer den entsprechenden Klienten und einen Dienst auswählen. Alle benötigten Parameter sowie Einstellungen über Startzeit, Endzeit und mögliche Wiederholungen müssen manuell hinterlegt werden. Eine Folge von Elementaraktionen kann zu Skripten zusammengefaßt werden.

6.2.4. Automatisierungsmodul

Um das Konzept dieser Arbeit überprüfen zu können, ist die Ausbaustufe 1 nur bedingt geeignet. Die Einführung eines Automatisierungsmoduls und damit auch die Erweiterung des Beispiels um das Modellkonzept der Last ist für Bewertungen unumgänglich. Entsprechend wurde prototypisch eine Minimalversion umgesetzt.

Dieser Prototyp ist in der Funktion stark eingeschränkt und auf das oben beschriebene Beispielsystem zugeschnitten. Es werden nur noch drei Dienste betrachtet, die sich in ihrer Natur grundlegend unterscheiden. Parallel dazu wird die Last für die einzelnen Dienste wie in Tabelle 6.2 definiert und quantifiziert.

Das Automatisierungsmodul ist ebenso in Java realisiert und verbindet sich als TCP-Client mit dem Testservermodul. Den Kern bildet der Testpartitionierungsalgorithmus auf Basis der Heuristik, wie er in Abschnitt 5.2.3 eingeführt wurde. Die Java-Implementierung, die auch für die Evaluierung

---

[1] siehe http://www.hibernate.org/

in Abschnitt 5.3 verwendet wurde, kann unverändert für das Testautomatisierungsmodul übernommen werden. Die Kostenfunktionen für die einzelnen Klienten sind weiterhin direkt im Programm hinterlegt. Ebenso werden die Testszenarien, also die Vorgaben für die Last in Abhängigkeit der Zeit, direkt im Programmcode hinterlegt. Für einen Prototypen sind diese statischen Einschränkungen hinnehmbar.

Dienst	Aktion	Parameter	1 Lasteinheit
TCP-Download	Datei runterladen	Quelladresse	10 kByte/s
TCP-Stream	Konstanten Datenstrom empfangen	Datenrate	400 Byte/s
Sprachanruf	Anruf starten	Dauer	1 Anruf

Tabelle 6.2.: Übersicht über die Dienste und Lasten, Ausbaustufe 2

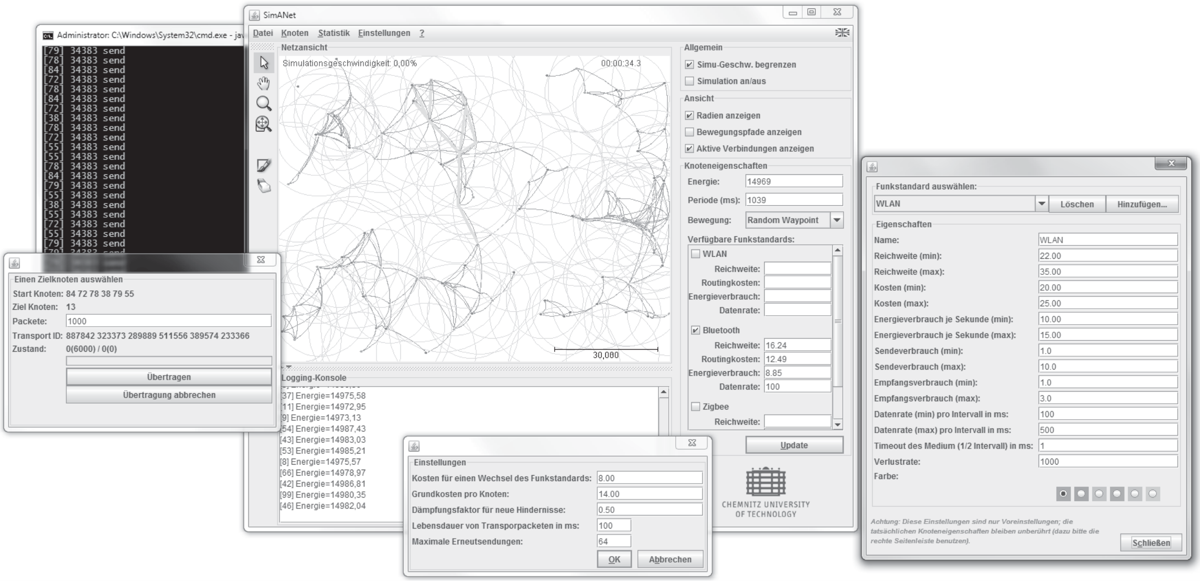
6.3. Simulation

Ausbaustufe 1 wurde weitestgehend vollständig umgesetzt und eingerichtet. Nach den Tests in einem realen UMTS-Netz ist die Simulation des SUT die praktikabelste Variante zur weiteren Überprüfung der Implementierung von Ausbaustufe 1 und zur Evaluierung der Funktions- und Leistungsfähigkeit von Ausbaustufe 2. Die Verwendung eines Simulators erlaubt die Ermittlung der Ist-Größen und -zustände des SUTs bzw. der Klienten. Durch die unmittelbare Verfügbarkeit dieser Informationen wird auch das Logging in Relation zu den Kontrollgrößen des Automatisierungssystems ermöglicht. Desweiteren sind mögliche Erweiterungen oder Fehlerkorrekturen an den jeweiligen Komponenten wesentlich einfacher auszurollen, als dies im Falle eines Tests in realen Mobilfunknetzwerken möglich wäre.

Eine zielführende Realisierung bietet ein Simulator, der das SUT, ggf. auch die Klienten und die Kommunikationsrelationen zwischen beiden simuliert. Genaugenommen muß ein solcher Simulator in der Lage sein, die zu testenden Dienste zu modellieren und simulierten Klienten den Zugriff auf diese zu ermöglichen. Die simulierten Klienten müssen die Möglichkeit bieten, den Testklienten als Skript oder Programm auszuführen und mit dem außerhalb des Simulators laufenden Testservermodul, wie es im obigen Abschnitt beschrieben ist, zu kommunizieren.

Der Einsatz eines Netzwerksimulators liegt also nahe. Von diesen existieren zahlreiche Vertreter mit jeweiligen Besonderheiten und Funktionen. Bekannte Simulatoren sind bspw. ns2 (siehe [49]), OMNeT++ (siehe [91]) oder das Simulink-basierte TrueTime (siehe [21]). Alle dienen in erster Linie der detaillierten Simulation drahtgebundener oder drahtloser Kommunikationsmedien oder -netzwerke bis auf die Ebene der Bitübertragungsschichten. Neue Technologien werden durch Forschungs- und Entwicklungsarbeiten in diese Simulatoren integriert (Vergleich bswp. [71]).

Die genaue Simulation von Ereignissen und Effekten in der Bitübertragungsschicht ist für die Testbenchvalidierung nicht notwendig. Vielmehr ist es entscheidend, daß der Simulator auch in der Lage ist, viele Klienten mit den jeweiligen Testklientenapplikationen zu handhaben. Demzufolge



```
Administrator: C:\Windows\System32\cmd.exe - jav
[79] 34383 send
[78] 34383 send
[84] 34383 send
[72] 34383 send
[78] 34383 send
[84] 34383 send
[72] 34383 send
[38] 34383 send
[78] 34383 send
[72] 34383 send
[55] 34383 send
[55] 34383 send
[78] 34383 send
[84] 34383 send
[79] 34383 send
[55] 34383 send
[38] 34383 send
[55] 34383 send
[72] 34383 send
[55] 34383 send
[79] 34383 send
```

Einen Zielknoten auswählen
Start Knoten: 84 72 78 38 79 55
Ziel Knoten: 13
Packete: 1000
Transport ID: 887842 323373 289889 511556 389574 233366
Zustand: 0(6000) / 0(0)

Simulationsgeschwindigkeit: 0,00%
00:00:34.3

```
[37] Energie=14975,58
[11] Energie=14972,95
[9] Energie=14973,13
[54] Energie=14987,43
[43] Energie=14983,03
[53] Energie=14985,21
[8] Energie=14975,57
[66] Energie=14978,97
[42] Energie=14986,81
[99] Energie=14980,35
[46] Energie=14982,04
```

Allgemein
☑ Simu-Geschw. begrenzen
☐ Simulation an/aus

Ansicht
☑ Radien anzeigen
☐ Bewegungspfade anzeigen
☑ Aktive Verbindungen anzeigen

Knoteneigenschaften
Energie: 14969
Periode (ms): 1039
Bewegung: Random Waypoint
Verfügbare Funkstandards:
☐ WLAN
Reichweite:
Routingkosten:
Energieverbrauch:
Datenrate:
☑ Bluetooth
Reichweite: 16.24
Routingkosten: 12.49
Energieverbrauch: 8.85
Datenrate: 100
☐ Zigbee
Reichweite:

Einstellungen
Kosten für einen Wechsel des Funkstandards: 8.00
Grundkosten pro Knoten: 14.00
Dämpfungsfaktor für neue Hindernisse: 0.50
Lebensdauer von Transporpacketen in ms: 100
Maximale Erneutsendungen: 64

Funkstandard auswählen: WLAN

Eigenschaften
Name: WLAN
Reichweite (min): 22.00
Reichweite (max): 35.00
Kosten (min): 20.00
Kosten (max): 25.00
Energieverbrauch je Sekunde (min): 10.00
Energieverbrauch je Sekunde (max): 15.00
Sendeverbrauch (min): 1.0
Sendeverbrauch (max): 10.0
Empfangsverbrauch (min): 1.0
Empfangsverbrauch (max): 3.0
Datenrate (min) pro Intervall in ms: 100
Datenrate (max) pro Intervall in ms: 500
Timeout des Medium (1/2 Intervall) in ms: 1
Verlustrate: 1000
Farbe: ● ○ ○ ○ ○ ○

Achtung: Diese Einstellungen sind nur Voreinstellungen; die tatsächlichen Knoteneigenschaften bleiben unberührt (dazu bitte die rechte Seitenleiste benutzen).

Abbildung 6.5.: Überblick über die SimANet Oberfläche, Vergleich [93]

ist der Einsatz eines Simulators sinnvoll, der auf Applikationsebene arbeitet. Ein solcher Simulator ist SimANet.

6.3.1. SimANet

SimANet ist ein ereignisgetriebener Simulator für vernetzte, eingebettete Systeme, der im Rahmen der Forschungsarbeiten zu Ambient Networks an der Professur Technische Informatik der Technischen Universität Chemnitz entstand (Vergleich [92] S. 107 ff., [94]). Der Name ist ein Akronym für „Simulation environment for Ambient Networking“.

Der Simulationskern und auch die Modelle für Knoten- und Kommunikationstechnologien sind darauf ausgerichtet, daß jeder Knoten im Netzwerk simultan auf mehrere drahtlose Kommunikationsmedien zugreifen kann. Durch die Simulation verschiedener Medienzugriffs-, Forwarding- und Routingstrategien kann SimANet dazu genutzt werden, zuverlässige und dennoch energieeffiziente Netzwerke für eingebettete Systeme zu entwickeln. Durch die Unterstützung von Bewegungsmodellen für Knoten und die modellbasierte Simulation von Hindernissen auf die Ausbreitung von elektro-magnetischen Wellen können auch derart dynamische Auswirkungen analysiert werden.

Die Simulation sämtlicher Kommunikationsvorgänge erfolgt ausschließlich auf Basis mathematisch-statistischer Modelle, nicht auf Basis physikalischer Effekte. Da dies weit weniger aufwändig ist, ist SimANet in der Lage, sehr viele Knoten in großen Netzwerken zu verwalten. Entsprechend sind auch die Datenstrukturen zur internen Verwaltung und Berechnung angelegt. Eine Erweiterung zum Betrieb auf Parallelrechnern ist ebenso verfügbar (Vergleich [95]).

Nahezu alle Funktionen des Simulators sind über eine graphische Oberfläche kontrollierbar, so daß ein schneller Einstieg in die Nutzung des Simulators möglich ist. In Abbildung 6.5 sind Details der Oberfläche zu sehen.

6.3.2. Kopplung an Testbench

SimANet wird verwendet, um die Dienste des SUT bereitzustellen und die Testklienten zu simulieren. Im Vordergrund steht jedoch nicht die Simulation der eigentlichen Dienst- bzw. Netzwerktätigkeiten, sondern die Generierung einer dynamischen Klientenstruktur, die über virtuell bereitgestellte Betriebsmittel auch virtuelle Dienste des SUT nutzt. Gesteuert wird alles durch die Testbench in Ausbaustufe 2, die nicht Teil der Simulation ist, sondern nur mit den Klienten im Simulator über ein virtuelles Gateway - ebenfalls Teil des Simulators, jedoch nicht Teil der Simulation - kommuniziert.

In Abbildung 6.6 ist die resultierende Architektur dargestellt.

Zur Umsetzung dieses Ansatzes waren einige Änderungen bzw. Erweiterungen an SimANet notwendig. An erster Stelle stand die Möglichkeit, ein Mobilfunknetzwerk mit Mobilgeräten umsetzen zu können. Da der Simulator ursprünglich zur Simulation von Ad-hoc Netzwerken konzipiert war, war eine Modellierung hierarchischer Netze, wie es zelluläre Netze sind, nicht möglich. Enstprechend wurden Knoten eingeführt, die im Netzwerk als Zugangspunkte[2] für Klienten funktionieren.

Ursprünglich war es für SimANet nicht vorgesehen, daß simulierte Knoten eigenen Code ausführen können. Lediglich die Steuerung über globale Skripte war vorgesehen. Für die Simulation im Rahmen dieser Arbeit ist es jedoch notwendig, daß jeder simulierte Knoten eine eigene Instanz des Testklienten ausführt. Sowohl der Simulator wie auch eine Variante des Testklienten liegen in der Standard Edition von Java vor. Durch Änderungen an der Implementierung der Knoten und Anpassungen am Testklient ist es ohne große Aufwände möglich, es jedem simulierten Knoten zu ermöglichen, eine Instanz des Testklienten zu starten und während der Simulation zu verwalten.

Eine dritte Erweiterung dient der Modellierung der Dienste selbst. Da diese Dienste in der Simulation keine Funktion erbringen müssen sondern nur die lastbedingte Verfügbarkeit modelliert werden muß, wird in SimANet eine zentrale Komponente zur Dienstverwaltung integriert. Diese verwaltet alle im vorangegangenen Abschnitt genannten Dienste. Testklienten, die einen Dienst nutzen möchten, können durch Anfrage an diese Komponente Ressourcen wie einen Socket oder einen virtuellen Anrufkanal reservieren und entsprechend nutzen. Dies trifft auf auf die Sockets zur Kommunikation zwischen Testservermodul und Klienten zu. Auch diese werden von der Dienstverwaltung bereitgestellt.

Bei den datenbasierten Diensten des Beispiels wird ein angeforderter Socket eines Klienten zu einem Ziel außerhalb der Simulation, bspw. dem Testservermodul, in zwei Sockets

---

[2] Englisch: Access Points

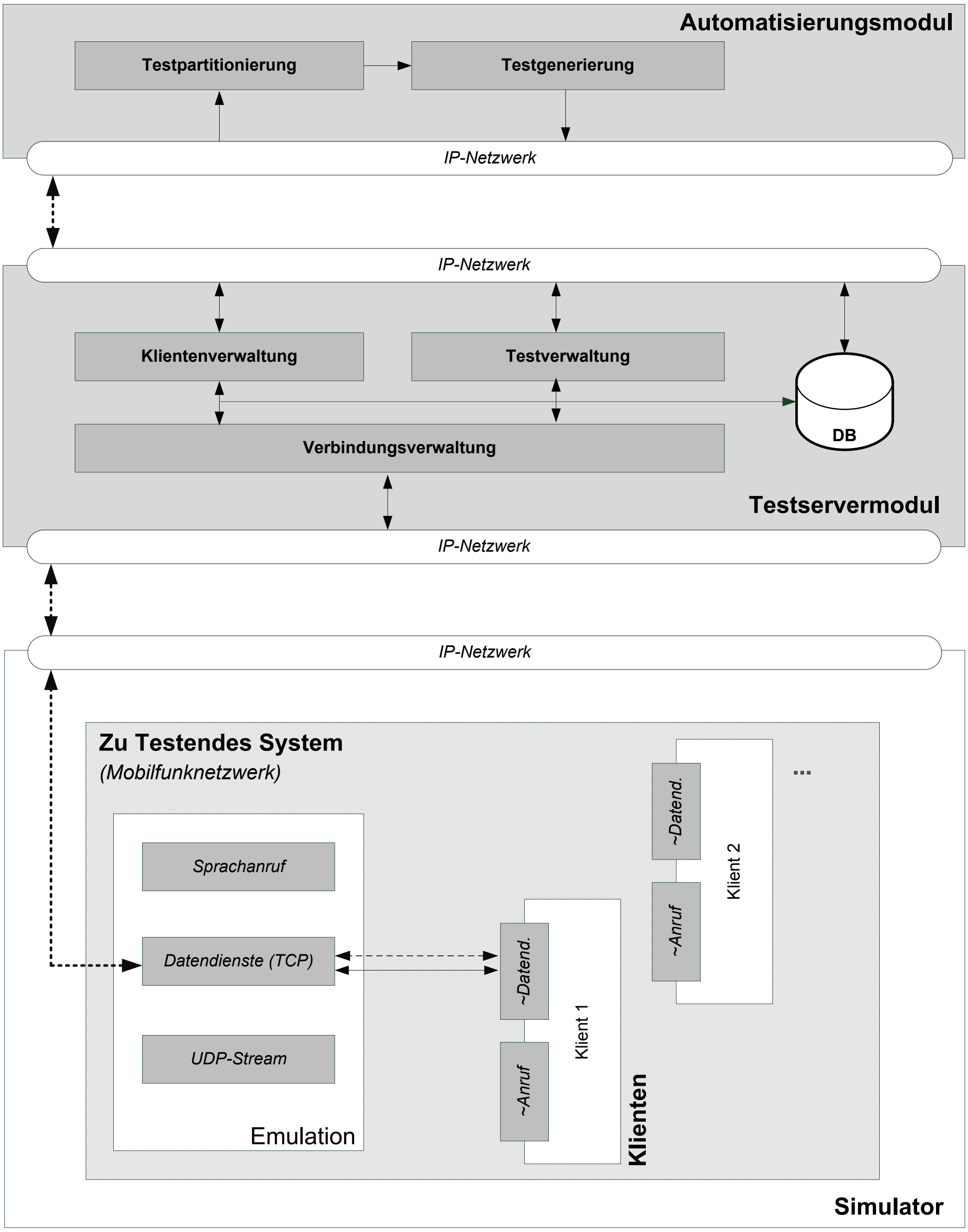


Abbildung 6.6.: Integration des Simulators zur Validierung der Testbench

aufgebrochen: Klient - Dienstverwaltung und Dienstverwaltung - Ziel. Die Dienstverwaltung kopiert entsprechend alle Datenströme zwischen beiden Sockets. Der Zugangspunkt, der im realen zellulären Netzwerk die Daten weiterleiten würde, bleibt hier außen vor. Er dient ausschließlich dazu, die Erreichbarkeit von Klienten zu definieren. Nur den Klienten, die sich in Reichweite eines Zugangspunktes befinden, werden Ressourcen durch die Dienstverwaltung zugewiesen.

Mit diesen Mechanismen ist es zum einen möglich, die wechselnde Erreichbarkeit der Mobilfunkgeräte im Mobilfunknetzwerk zu simulieren. Zum anderen erlaubt die zentrale Dienstverwaltung das Logging der Ressourcen, bspw. die quantitative Nutzung der Sockets, und bietet Eingriffsmöglichkeiten, wenn sich ein Klient aus dem simulierten Empfangsbereich eines Zugangspunktes bewegt.

Neben diesen funktionalen Erweiterungen waren Adaptionen notwendig, die der Auswertung der Simulation bzw. des Tests dienen. Insbesondere die Daten der Dienstverwaltung im SimANet sind für das Logging interessant, da dadurch direkte Soll-Ist-Vergleiche generiert werden können. Die entsprechenden Informationen müssen zur Generierung kombinierter Logdateien an einer zentralen Stelle gesammelt werden - in diesem Falle im Automatisierungsmodul.

Der Informationsaustausch zwischen Simulator und Automatisierungsmodul kann natürlich ebenso über einen Socket erfolgen. Da jedoch beide Komponenten in Java realisiert sind und der Austausch dieser Informationen kein funktionaler Bestandteil der Testbench ist, ist die Nutzung eines *Remote Procedure Call* Ansatzes einfacher umzusetzen. Die entsprechende Realisierung in Java wird *Remote Method Invocation* (RMI) genannt (siehe [41]). Zur Umsetzung wird in SimANet ein RMI-Server implementiert, der über dedizierte Methoden entsprechenden RMI Klienten ermöglicht, Informationen, wie die Anzahl laufender Anrufe oder aktuelle Datenraten, abzurufen. Das Testautomatisierungsmodul implementiert einen solchen Klienten.

Es ist zu beachten, daß der direkte Austausch von Informationen zwischen Simulation und Testbench ausschließlich dem Logging dient. Die gewonnen Informationen werden nicht für die Testautomatisierung genutzt, da diese in realen Systemen nicht zur Verfügung stehen oder nur sehr aufwändig zu gewinnen wären.

6.4. Zusammenfassung

Zur Evaluierung des vorgestellten Konzeptes wurden in diesem Kapitel Details einer prototypischen Implementierung vorgestellt, die auf dem Beispiel der Validierung von Mobilfunknetzwerken basiert.

Projektbedingt wurde die Implementierung in zwei Stufen geteilt. Die erste Stufe realisierte die Testklienten sowie das Testservermodul und erlaubt die Bedienung mittels einer einfachen graphischen Oberfläche. In einer zweiten Ausbaustufe wurde die graphische Oberfläche durch eine Implementierung des Automatisierungsmoduls ersetzt, welches die Heuristik zur

Testpartitionierung umsetzt. Die jeweiligen Module arbeiten als eigenständige Programme und kommunizieren über TCP-Sockets.

Während ein reales Mobilfunknetzwerk für erste Untersuchungen mit der Ausbaustufe 1 verwendet werden konnte, mußte für Stufe 2 eine Simulation des zu testenden Systems genutzt werden. Hierzu wurde der SimANeT-Simulator entsprechend angepaßt und weiterentwickelt, so daß Dienste virtuell zur Verfügung gestellt und quantifiziert werden können.

Die bestehende Kombination aus Implementierung der Ausbaustufe 2 und des Simulator erlaubt eine Evaluierung des Konzeptes.

7. Ergebnisse

In den zurückliegenden Kapiteln wurde ein Konzept zur Validierung heterogener Systeme sowie dessen partielle Implementierung vorgestellt. Im folgenden Kapitel werden die Ergebnisse vorgestellt, die mit dem Konzept erreicht wurden.

Zunächst wird das Konzept anhand quantitativer Betrachtungen bewertet. Ein Großteil der Ergebnisse wurden mit Hilfe der Kombination aus Testbench und Simulator, wie es im zurückliegenden Kapitel vorgestellt wurde, experimentell ermittelt. Dadurch wird es möglich, die Leistungsfähigkeit des Konzeptes bzw. der existierenden Implementierung in absoluten Größen einschätzen zu können. Es schließt sich eine Diskussion über Stärken und Schwächen des Systems an, die eher der qualitativen Bewertung des Konzeptes dient und priorisierte Anwendungsfelder aufzeigen soll.

7.1. Quantitative Bewertung

Mit der im Abschnitt 6.3 vorgestellten Kombination aus Simulator und Testbench werden einige Messreihen durchgeführt, die zum einen die Funktionsfähigkeit des Konzeptes demonstrieren und zum anderen erste Größenordnungen für verschiedene Parameter liefern sollen.

7.1.1. Rahmenbedingungen

Sämtliche Komponenten, also Testbench, Simulator, Testklienten und die simulierten Knoten usw., werden auf dem gleichen Desktopcomputer ausgeführt. Folglich sind die gewonnenen absoluten Werte - vor allem für die Zeit - nur bedingt aussagekräftig, da durch Ressourcenkonflikte (Prozessor, Netzwerk, ...) Abhängigkeiten zwischen den gewonnenen Werten entstehen, die in einem realen Testbench-Aufbau nicht auftreten würden. Unter bestimmten Szenarien müssen außerdem eine sehr große Zahl von Sockets durch die JVM bzw. das Betriebssystem auf dem einen Rechner verwaltet werden. Es ist schwierig zu beurteilen, welchen Einfluß dies auf das zeitliche oder auch funktionale Verhalten des Evaluierungssystems hat.

Im Vordergrund der Messungen stehen nicht die Ergebnisse der Testpartitionierung, da diese bereits im Kapitel 5 ausführlich bewertet wurden. Vielmehr liegt das Verhalten des Gesamtsystems im Fokus, das sich aus den Aktivitäten aller Module, der Testklienten und der Simulation des SUT ergibt. Insbesondere durch Latenzen bei der Berechnung bzw. bei der Kommunikation entstehen Auswirkungen auf das Soll/Ist-Verhältnis der Lasterzeugung, was im Laufe der Messungen analysiert wird.

Es gilt hierbei zu beachten, daß keine Kopplung zwischen der Simulationszeit von SimANet und den Zeiten der Testbench besteht. Alle Zeiten, die in den Aufzeichnungen für die Messungen verwendet werden, sind absolute Zeiten der nativen Ausführung der jeweiligen Programme.

Für die Messungen werden die jeweilig relevanten Parameter synchron alle 10ms sowie asynchron bei relevanten Zustandsänderungen in der Testbench in eine Log-Datei geschrieben.

7.1.2. Dienste

Betrachtet werden im folgenden die drei Dienste, die in Tabelle 6.2 vorgestellt wurden. Diese drei wurden bewußt ausgewählt, da sie sich konzeptionell und in der technischen Umsetzung stark unterscheiden und damit die Analyse verschiedener Parameter ermöglichen.

Der einfachste und praktisch naheliegendste Dienst ist der Sprachanruf, der durch Dienst 3 repräsentiert wird. Soll vom simulierten Klienten ein Anruf gestartet werden, wird beim Dienstmanagement des Simulators ein entsprechender Kanal angefordert. Da die Funktion selbst nicht simuliert wird, ist der einzige relevante Parameter, der durch den Testgenerator im Automatisierungsmodul vorgegeben werden muß, die Zeit, die der Anruf dauern soll. Diese Zeit wird zufällig innerhalb eines Intervalls gewählt. Ein einmal gestarteter Anruf kann nicht abgebrochen werden. Die Last für einen Sprachanruf wird mit der Anzahl der laufenden Anrufe quantifiziert. Als Einschränkung gilt, daß jeder Klient zu einem Zeitpunkt nur höchstens einen Anruf tätigen kann. Dementsprechend beträgt die Last für jeden Klienten nur 0 oder 1.

Dienst 2 ist ein Streaming-Dienst auf TCP-Basis. Dieser kontaktiert einen Server und lädt ein einstellbares Datenvolumen pro Sekunde von diesem herunter. Dieser Dienst ist damit praktisch mit einem Videoportal vergleichbar, von dem Mobilfunknutzer Videos streamen können. Die Last definiert sich hier als Datenvolumen pro Zeit, wobei 400 Byte/s genau einer Lasteinheit entsprechen und sich die Lasten als Vielfaches der Grundeinheit ergeben. Jeder Klient kann nur einen Streaming-Socket öffnen, auf diesem aber veränderliche Lasten (also Datenvolumen pro Zeit) generieren. Das Datenvolumen kann folglich von der Testbench eingestellt werden. Das generierte Datenvolumen wird von der Dienstimplementierung im Testklienten durch einen entsprechenden Zeitmechanismus gesteuert.

Im Gegensatz zum Sprachanrufdienst, wo die Testbench einen Anruf auf einem Klienten nur starten kann, kann sie beim Streamingdienst die Last folglich detaillierter auf 400 Byte/s genau einstellen. Das trifft auch für negative Änderungen, also das Verringern der Last, zu. Der Abbruch eines Sprachrufs ist hingegen nicht möglich, so daß eine negative Laständerung erst nach Ablauf der voreingestellten Anrufzeit eintritt. Es wäre natürlich möglich, auch hier einen Abbruch umzusetzen. Für die Evaluierung des Testbenchkonzeptes ist es in diesem Fall sinnvoller, diese Unterschiede zu belassen.

Dienst 1 ist ebenso wie Dienst 2 ein TCP-basierter Datendienst. Jedoch werden hier per HTTP Dateien von einem Webserver auf den Klienten heruntergeladen. Es spiegelt also die Standardaktion eines Mobilfunknutzers wieder, wenn dieser im Internet surft. Als Last wird ebenso das

Datenvolumen pro Zeit definiert, wobei eine Lasteinheit 10 kByte/s entspricht. Im Gegensatz zum Streamingdienst ist jedoch die Datenrate nicht einstellbar. Es können nur Downloads einzelner Dateien gestartet werden. Die daraus resultierende Datenrate hängt also vom jeweiligen Socket, dessen Implementierung und der Bandbreite des physischen Kanals ab. Die Dauer des Downloads ergibt sich wiederum aus der Datenrate und der Größe der Datei. Es können mehrere Downloads - folglich auch mehrere Sockets - pro Klient gestartet, aber nicht abgebrochen werden.

Die Beschaffenheit des physischen Kanals wird nicht detailliert simuliert. Jedoch wird vom Dienstmanagement des Simulators nach verschiedenen Modellen die Datenrate eines Sockets aktiv beeinflußt. Da es für die Testbench bei diesem Dienst nicht möglich ist, die Last direkt zu beeinflußen, ist es bereits jetzt offensichtlich, daß der Wert der simulierten Datenrate deutlichen Einfluß auf die Soll/Ist-Relation der Lasten hat. Details werden im Zusammenhang mit den jeweiligen Messreihen diskutiert.

Im Automatisierungsmodul werden die Dienste für die jeweiligen Klienten durch die gleichen Kostenfunktionen beschrieben, wie sie zur Evaluierung verwendet und in den Gleichungen A.1 bis A.11 beschrieben wurden. Es existieren damit auch in den folgenden Messungen fünf Kostenfunktionsgruppen.

Die Maximallasten der einzelnen Klienten der jeweiligen Kostenfunktionsgruppen sind für Dienst 2 und 3 identisch zu den in Gleichungen A.12 bis A.17 angegebenen. Für Dienst 1 wurden die Maximallasten auf die in den Gleichungen 7.1 bis 7.5 gegebenen Werte angepaßt. Zu verstehen sind diese derart, daß Dienst 1 für die Klientengruppe 1 eine Maximallast von $3 \cdot 10 = 30$ kByte/s generieren kann.

$$a_{1,1} = 3 \tag{7.1}$$

$$a_{2,1} = 2 \tag{7.2}$$

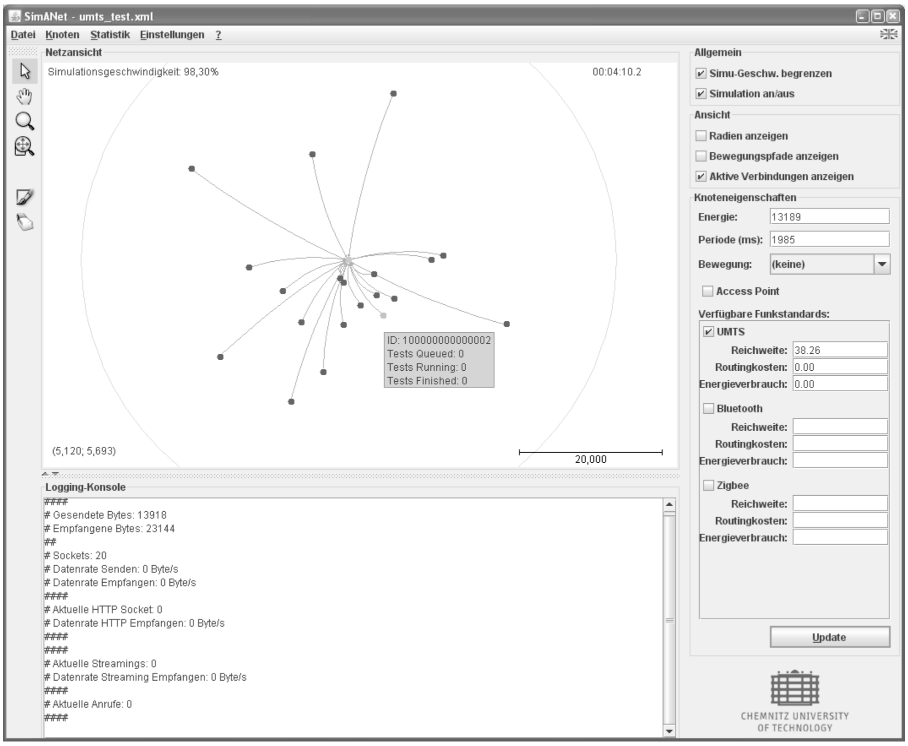
$$a_{3,1} = 1 \tag{7.3}$$

$$a_{4,1} = 1 \tag{7.4}$$

$$a_{5,1} = 1 \tag{7.5}$$

7.1.3. Statische Netze

Mit den ersten Messungen wird die Lasterzeugung der Dienste einzeln überprüft. Um die Ergebnisse möglich nachvollziehbar zu halten, wird zunächst auf einen rein statischen Aufbau des SUT und der Klienten gesetzt. Im Simulator wird ein virtuelles Mobilfunknetzwerk mit einem Zugangspunkt und 20 Klienten angelegt. Alle Klienten unterstützen alle drei Dienste und befinden sich während der kompletten Zeit der Messung in Reichweite des Zugangspunktes - sind also damit für die Lastgenerierung verfügbar. Ein Screenshot dieser Konfiguration in SimANet ist in Abbildung 7.1 zu sehen.



ID: 100000000000002
Tests Queued: 0
Tests Running: 0
Tests Finished: 0

```
####
# Gesendete Bytes: 13918
# Empfangene Bytes: 23144
##
# Sockets: 20
# Datenrate Senden: 0 Byte/s
# Datenrate Empfangen: 0 Byte/s
####
# Aktuelle HTTP Socket: 0
# Datenrate HTTP Empfangen: 0 Byte/s
####
####
# Aktuelle Streamings: 0
# Datenrate Streaming Empfangen: 0 Byte/s
####
# Aktuelle Anrufe: 0
####
```

Allgemein
☑ Simu-Geschw. begrenzen
☑ Simulation an/aus

Ansicht
☐ Radien anzeigen
☐ Bewegungspfade anzeigen
☑ Aktive Verbindungen anzeigen

Knoteneigenschaften
Energie: 13189
Periode (ms): 1985
Bewegung: (keine)
☐ Access Point

Verfügbare Funkstandards:
☑ UMTS
Reichweite: 38.26
Routingkosten: 0.00
Energieverbrauch: 0.00
☐ Bluetooth
Reichweite:
Routingkosten:
Energieverbrauch:
☐ Zigbee
Reichweite:
Routingkosten:
Energieverbrauch:

Update

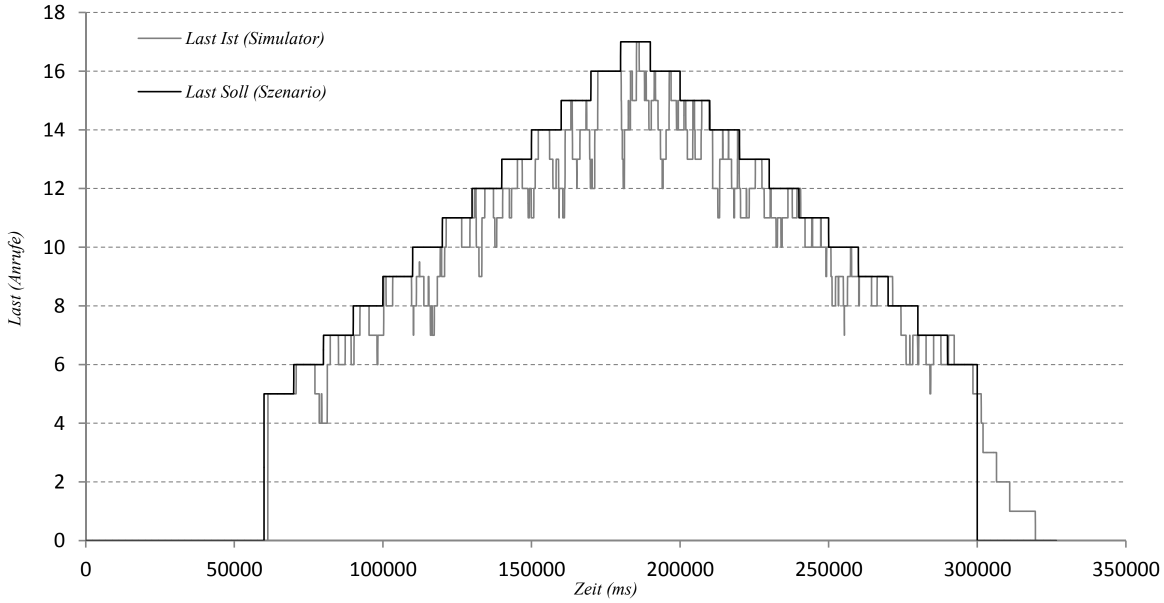
Abbildung 7.1.: Netzwerkkonfiguration in SimANet für statisches Klientennetz

Für den jeweiligen Dienst werden zwei Szenarien angelegt und unabhängig voneinander ausgeführt. Beide starten bei einer Last von 0 und erhöhen dann im Verlauf der Zeit die Last in festen Schritten. Bei Szenario 1 dauert jeder Schritt 10 Sekunden, bei Szenario 2 hingegen 60 Sekunden. Die Größe der Laständerung in jedem der Schritte hängt von Dienst und der assozieren Definition der Last ab. Nach Erreichen des Maximums wird die Last in ebenso vielen Schritten auf 0 zurückgefahren.

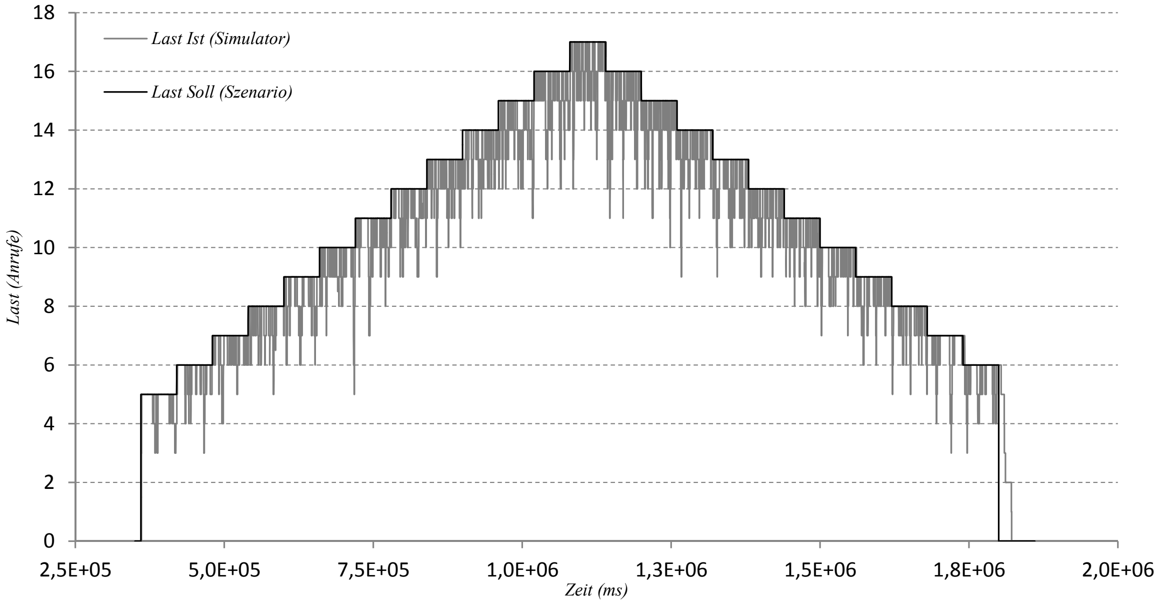
7.1.3.1. Sprachanrufdienst und Streaming-Dienst

Die berechnete Soll-Last, also der Wert des Szenarios zum entsprechenden Zeitpunkt, sowie die im Dienstmanagment des Simulators allokierten Lasten, die mittels Java-RMI durch die Testbench abgefragt werden, werden geloggt. Die Ergebnisse der Ausführung von Szenario 1 und 2 mit dem Sprachanrufdienst sind in Abbildung 7.2 dargestellt. Jede Lasteinheit entspricht einem laufenden Anruf, so daß in der Spitze der Messung 17 Anrufe auf die 20 Klienten verteilt werden müssen.

An den Messkurven kann die tatsächliche Last im Vergleich zum Szenario in Abhängigkeit von der Zeit abgelesen werden. Es wird deutlich, daß die Ist-Last den Vorgaben weitestgehend folgt, es jedoch häufig zu kurzfristigen Einbrüchen kommt. Diese werden durch das Ende der Anrufe verursacht. Wird die beim Start vorgegebene Dauer des Anrufs erreicht, wird dieser durch den Testklient beendet und eine entsprechende Meldung an das Testservermodul gesendet. In Folge muß das Automatisierungsmodul die Differenz wieder neu partitionieren. Da die Nachrichten für den Bericht und den Start ebenso Zeit benötigen, wie die Verarbeitung und Berechnung in der



(a) Dienst 3, Szenario 1



(b) Dienst 3, Szenario 2

Abbildung 7.2.: Vergleich Soll-/Ist-Last für Dienst 3

Testbench, entstehen zwischen dem Ende eines Anrufs und dem Start eines neuen entsprechende Verzögerungen. An vielen Stellen kommt es auch zu Überlagerungen mehrerer Adaptionen, so daß der Einbruch der Ist-Last in den Messungen im extremen Falle zu einer Differenz von 5 (Anrufen) führt. Am Ende des Testszenarios, ab ca. 300 Sekunden, ergibt sich eine längerfristige Differenz. Diese entsteht durch die Tatsache, daß Anrufe nicht abgebrochen werden können und diese erst nach Ablauf der eingestellten Zeit durch den Klienten beendet werden.

An dieser Stelle sei darauf hingewiesen, daß bei der Testpartitionierung und im Testgenerator keine Vorausplanung der Lasten erfolgt. Dementsprechend wird die Anrufdauer unabhängig von einer in naher Zukunft zu erwartenden stark fallenden Soll-Last gesetzt.

Da die Last eines jeden Klienten exakt gesetzt werden kann und die Testklienten technisch auch in der Lage sind, diese Vorgaben genau umzusetzen, kommt es zu keinem Zeitpunkt zum Überschreiten der vorgegebenen Lasten.

Genaue numerische Analysen für die Testläufe mit Szenario 1 und 2 sind in Tabelle 7.1 gegeben. So ist ersichtlich, daß in Szenario 1 die durchschnittliche Abweichung zwischen Soll- und Ist-Last $0,984$ Lasteinheiten - also rund 1 Anruf - beträgt. Bei einer durchschnittlichen Last von rund 10 Einheiten bedeutet dies also eine Abweichung von rund 10%. Dieser Wert wird auf Basis der diskreten Meßzeitpunkte der Log-Datei berechnet. Dieses Logging erfolgt sowohl in bestimmten Zeitabständen als auch asynchron nach dem Auftreten von Änderungsmeldungen. Folglich ist die Zeitdifferenz zweier aufeinanderfolgender Log-Einträge nicht zwangsläufig identisch und es können bei den Berechnungen der durchschnittlichen Abweichung Ungenauigkeiten entstehen. Prinzipiell werden nur Werte für den Zeitraum, in dem die Soll-Last größer als 0 ist, betrachtet.

Eine alternative Bewertung bietet die Betrachtung des Integrals der Soll- bzw. Ist-Last über der Zeit. Dieses läßt sich numerisch anhand der Meßreihen durch Berechnung der Lastflächen zwischen zwei Meßzeitpunkten berechnen. Wird die Zeit in Millisekunden in die Berechnung einbezogen, ergibt sich für die Soll-Last (Szenario 1) eine Gesamtfläche von 2640000 und für die Ist-Last von 2463495. Demzufolge wird ungefähr 93,5% der zu generierenden Last tatsächlich erzeugt.

Neben diesen Vergleichen für die Last sind in Tabelle 7.1 auch Latenzen angegeben. Δt_1 entspricht der Verzögerung zwischen dem Sprung der Soll-Last von 0 auf 5 (in Szenario 1 zum Zeitpunkt $60s$) und dem entsprechenden Erreichen der Ist-Last. Δt_2 entspricht der Verzögerung beim Sprung von 5 auf 6 und Δt_3 beim Fall von 6 auf 0. Es ist offensichtlich, daß Δt_3 mit ca. 20 Sekunden deutlich über den übrigen Verzögerungen liegt, was im oben beschriebenen Auslaufen der Anrufe begründet liegt.

Szenario 2 unterscheidet sich von Szenario 1 in der Zeit, in der auf jeder der Lastäquivalente verweilt wird. Dadurch sollen mögliche Einflüße durch kurzfristige Änderungen ausgeschlossen werden. Verglichen mit den Ergebnissen für Dienst 3 ergeben sich keine relevanten Unterschiede zwischen Szenario 1 und 2.

Dienst 2 und 3 unterscheiden sich in einigen Details. Der wichtigste Unterschied ist, daß es bei Dienst 2 keine physikalische oder technologische Entsprechung der Last gibt. Im Falle

	Dienst 3		Dienst 2	
	Szenario 1	Szenario 2	Szenario 1	Szenario 2
∅ Abweichung	0,984	0,776	265	138
$\int Ist$	2463495	14829340	1074685222	6356107390
$\int Soll$	2640014	15840100	1056012600	6335978400
$\int Ist / \int Soll$	0,913	0,936	1,018	1,003
$\Delta t_1 (ms)$	1281	1328	2016	2578
$\Delta t_2 (ms)$	828	1328	2187	2484
$\Delta t_3 (ms)$	19563	21766	20938 (4579)	38875 (4750)

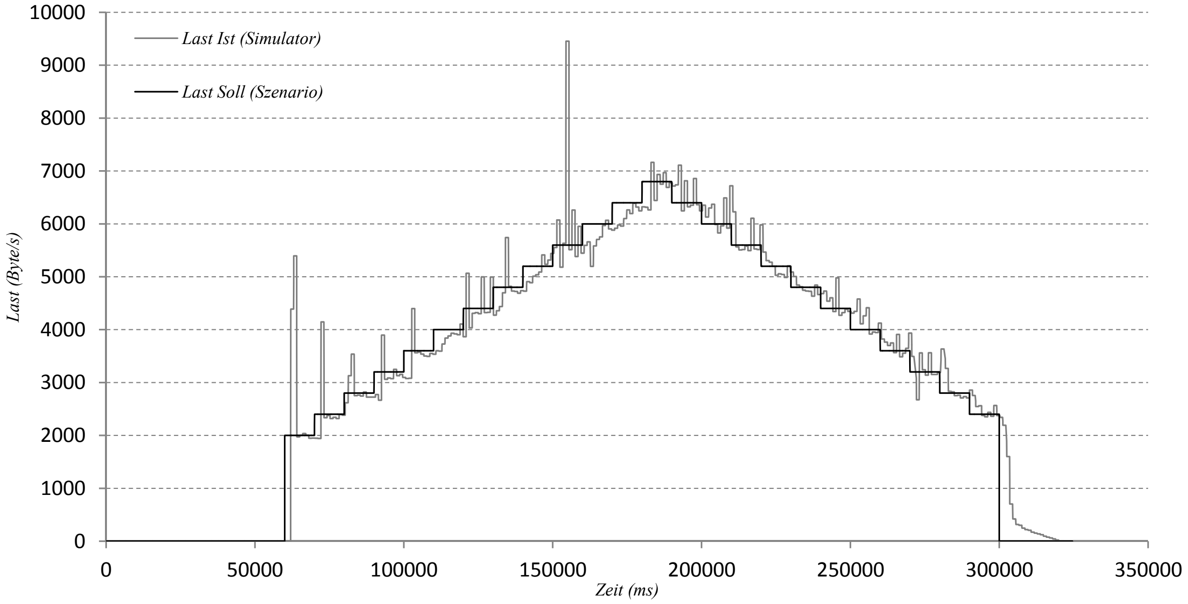
Tabelle 7.1.: Numerische Auswertung eines Testlaufs für Dienst 3 und Dienst 2

des Streamingdienstes wird die Last in Byte/s quantifiziert. Um gegebene Last zu erzeugen, existieren verschiedene Möglichkeiten. Bspw. kann ein Klient einen TCP-Socket öffnen. Oder mehrere Klienten können jeweils einen TCP-Socket öffnen und in Summe die entsprechende Last generieren. Desweiteren können die Testklienten für die Dienst 2 auch negative Laständerungen entsprechend umsetzen.

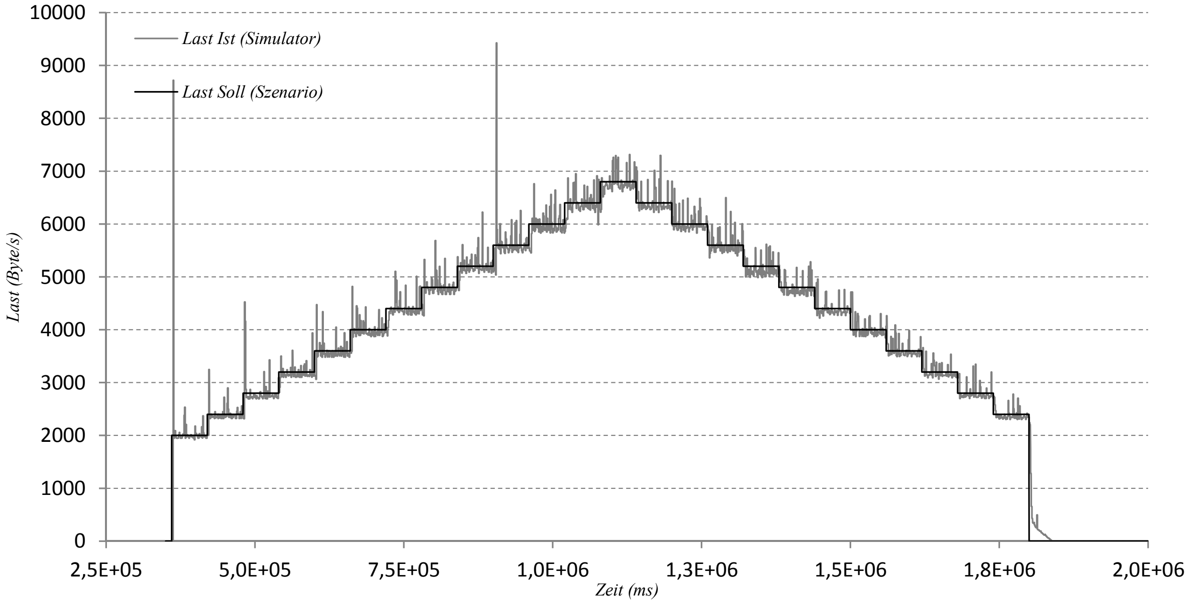
Inwieweit sich dies auf die Ausführung der Tests auswirkt, wird in den Messungen für Dienst 2 in Abbildung 7.3 und in den entsprechenden numerischen Auswertungen in Tabelle 7.1 dargestellt. Im Vergleich zu dem Messungen mit Dienst 3 fällt auf, daß die negativen Lastdifferenzen nicht auftreten. Dies ist darauf zurückzuführen, daß in Dienst 2 die Last an den Klienten (indirekt) eingestellt werden kann und das Testautomatisierungsmodul nicht auf Rückmeldungen der Testklienten reagieren muß. Auffällig sind hingegen die beiden massiven Lastspitzen, die auch in beiden Szenarien auftreten. Die erste dieser Spitzen tritt beim Wechsel von Last 0 auf 2000 $Byte/s$ auf und resultiert aus einem ungünstigen Effekt der Berechnung der Logging-Werte - kann also als Meßfehler bezeichnet werden. Die Ursache für die zweite Spitze bei ca. 150 Sekunden bzw. ca. 900 Sekunden konnte nicht abschließend geklärt werden. Es ist jedoch zu vermuten, daß auch hier eine ungünstige Konstellation im Logging-System zur Fehlmessung führt.

Die weiteren Schwankungen sind auf Ungenauigkeiten in der Generierung der Datenströme der einzelnen Sockets zurückzuführen. In entsprechenden Threads werden die geforderten Daten pro Zeit zwischen Java-Streams kopiert. Da jedoch die Zeitsteuerung der Threads nicht hundertprozentig genau arbeitet und auch vom Betriebssystem bzw. der JVM beeinflußt wird, kann es zu leichten Abweichungen kommen. Diese liegen im Durchschnitt bei ca. 250 Byte/s. Bei der Betrachtung des Lastintegrals werden bei Dienst 2 Abweichungen von nur 2% bei Szenario 1 bzw. sogar weniger als 1% im Falle von Szenario 2 erreicht.

Der Tabelle 7.1 kann entnommen werden, daß bei Dienst 2 deutlich bessere Annäherungen an die Soll-Last als bei Dienst 3 erreicht werden. Jedoch stehen dem etwas höhere Latenzen gegenüber. Diese entstehen bei steigender Last durch die Verzögerung zum Aufbau der benötigten Sockets und die Generierung der Datenströme. Im Falle von Δt_3 trat bei beiden Szenarien ein Problem beim



(a) Dienst 2, Szenario 1



(b) Dienst 2, Szenario 2

Abbildung 7.3.: Vergleich Soll-/Ist-Last für Dienst 2

Schließen eines der Sockets auf, so daß dieser erst nach Erreichen eines Timeouts geschlossen wurde. Entsprechend groß sind die Latenzen, nämlich rund 20 bzw. 40 Sekunden, bis die Ist-Last auf 0 gesunken ist. Werden diese Sockets von der Betrachtung ausgeschlossen, entsteht in beiden Szenarien eine Verzögerung von knapp 5 Sekunden. Dieser Wert ist deutlich niedriger im Vergleich zu Dienst 3, da beim Streamingdienst nicht auf das Auslaufen von Aktionen gewartet werden muß.

Im direkten Vergleich von Dienst 2 und 3 schneidet also die Variante des Streamingdienstes (Dienst 2), bei dem jeder Testklient die zu generierende Last innerhalb gewisser Genauigkeitsgrenzen automatisch erbringt, günstiger ab. Dies ist wenig überraschend, da die Verzögerung von der Meldung an den Testklienten bis zur Partitionierung durch das Automatisierungssmodul, wie sie im reaktiven Dienst 3 entstehen, entfallen.

7.1.3.2. Download-Dienst

Die Schwachpunkte der anderen beiden Dienste finden sich auch in Dienst 1, so daß in diesem Falle mit schlechteren Ergebnissen zu rechnen ist. Ähnlich wie in Dienst 2 gibt es hier keine direkte Kopplung zwischen der Last und der technischen Umsetzung dieser. Auch im Falle der HTTP-Downloads müssen Sockets aufgebaut werden, über die die Datenströme dann die Last erzeugen. Im Falle von Dienst 1 ist jedoch die genaue Einstellung der Last in den Testklienten nicht möglich. Beim Herunterladen einer Datei bestimmen Parameter des physikalischen Kanals oder möglicherweise auch Ressourcenbeschränkungen auf Seiten des Servers, des Netzes oder des Klienten, mit welcher Bandbreite übertragen werden kann. Selbst die ausgewählte Datei hat in Abhängigkeit von der Bandbreite nur Einfluß auf die Dauer eines Ladevorgangs, nicht jedoch auf die Bandbreite selbst. Eben diese schwankenden Zeiten für das Herunterladen führen ebenso dazu, daß Laständerungen in Dienst 1 genau wie in Dienst 3 reaktiv nach Benachrichtigung des Testservermoduls durch das Automatisierungsmodul korrigiert werden müssen. Die entsprechenden Verzögerungen werden ebenso Einfluß auf die Soll-/Ist-Relation der Lasten haben.

Als Implementierung des HTTP-Klienten in den Testklienten wird der HttpClient[1] aus dem „HTTP Components“ Programm der Apache Software Foundation verwendet. Dieser wurde derart angepaßt, daß statt eines TCP-Sockets der JVM ein virtueller Socket des Dienstmanagments in SimANet verwendet wird. Es wird für die folgenden Messungen zunächst angenommen und durch das Dienstmanagment sichergestellt, daß jeder virtuelle Socket eine Bandbreite von 50 kByte/s hat.

Es liegt nahe, daß bei konstanten Bandbreiten die Größe der herunterzuladenden Dateien Einfluß auf die Ergebnisse haben. Zur Überprüfung werden Messungen unter drei Bedingungen durchgeführt, wie sie in Tabelle 7.2 aufgeführt sind. Die Dateien werden von einem HTTP-Server bereitgestellt, der nicht auf dem zur Simulation genutzten Rechner läuft. Um Zugriffsbeschränkungen auf einzelne Dateien durch den Server zu vermeiden, liegen alle Dateien

---

[1] Package org.apache.http.client.HttpClient, siehe http://hc.apache.org

Messung	Szenario	Parameter
1	1	Einheitliche Dateigröße, ca. 120 kB
2	1	4 verschiedene Dateigrößen, 20 kB... 120 kB
3	1	4 verschiedene Dateigrößen, 90 kB... 240 kB
4	2	4 verschiedene Dateigrößen, 90 kB... 240 kB

Tabelle 7.2.: Definition Messungen Dienst 1

	Messung 1	Messung 2	Messung 3	Messung 4
∅ Abweichung (Byte/s)	15668	239333	11830	9464
$\int Ist$	24331328402	21434674612	25537169229	$1,513 \cdot 10^{11}$
$\int Soll$	27033876480	27034101760	27033359360	$1,622 \cdot 10^{11}$
$\int Ist / \int Soll$	0,900	0,793	0,945	0,933
$\Delta t_1 (ms)$	3078	2969	3016	3672
$\Delta t_2 (ms)$	7141	7032	7078	6719
$\Delta t_3 (ms)$	1203	5156	2156	1547
$\Delta t_4 (ms)$	3500	9110	20859	12328

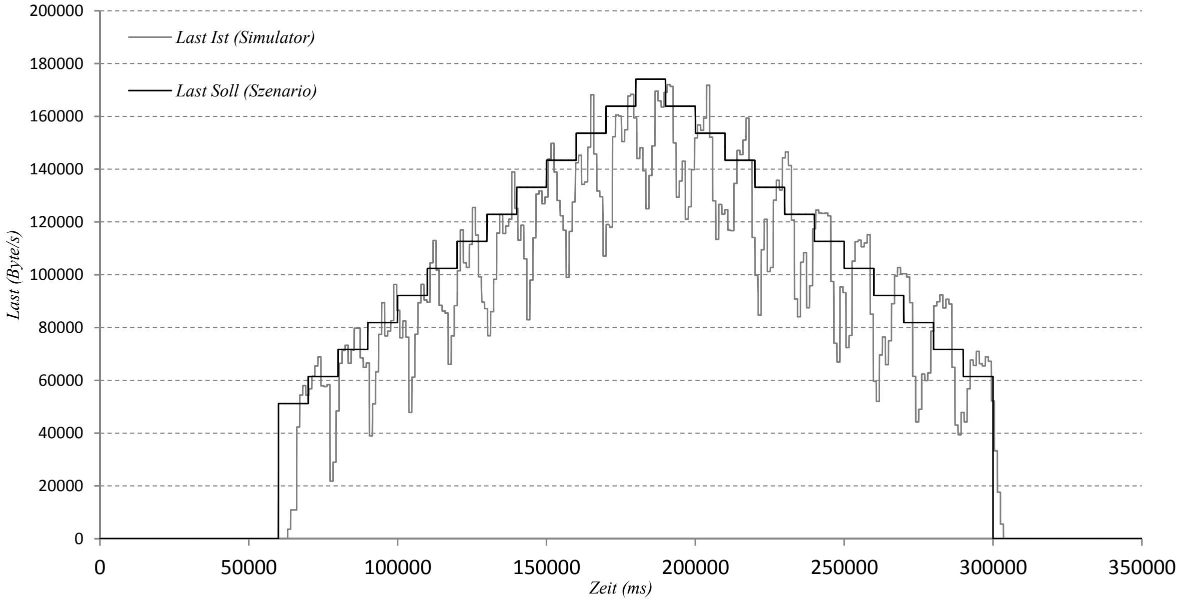
Tabelle 7.3.: Numerische Auswertung der Meßreihen für Dienst 1 (siehe Tabelle 7.2)

in mehreren Kopien vor. Der Testgenerator iteriert die Dateiinstanzen, so daß ein zeitlich paralleler Zugriff auf die gleiche Datei vermieden wird.

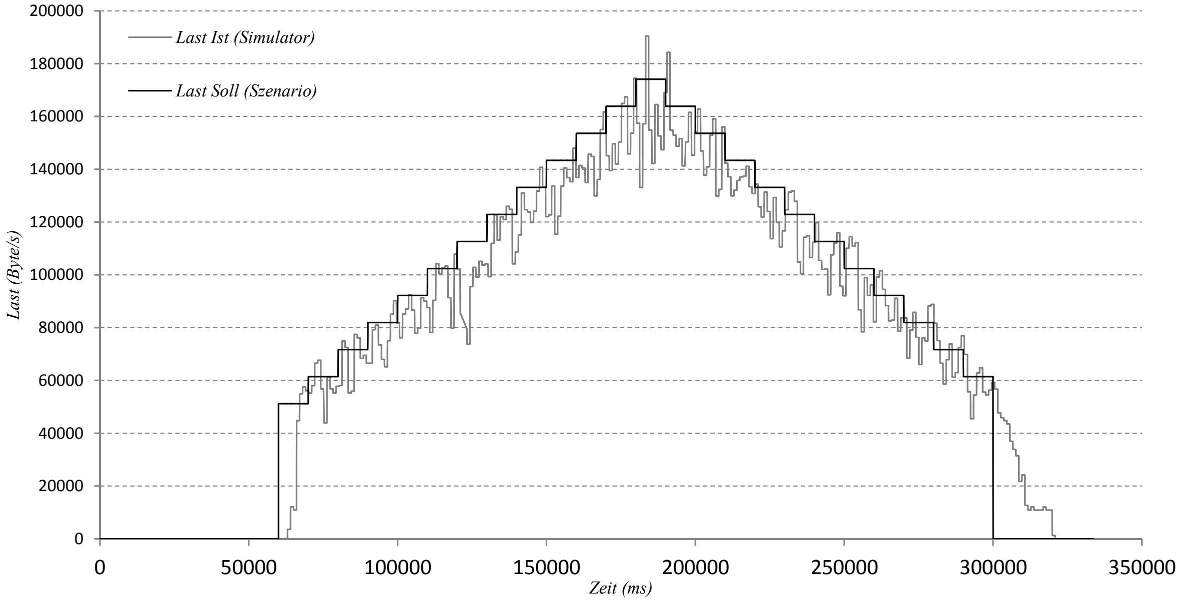
Die Ergebnisse der Ausführung sind in Tabelle 7.3 und in den Abbildungen 7.4 sowie 7.5 dargestellt.

Dateien gleicher Größe führen bei identischen Bandbreiten auch zu weitestgehend gleichen Downloadzeiten. Demzufolge enden Downloads, die zu ähnlichen Zeiten gestartet wurden, auch zu ähnlichen Zeiten. Dies wiederum führt zu starken Lasteinbrüchen, bis das Automatisierungmodul nachregeln kann. Abbildung 7.4(a) veranschaulicht diesen Effekt deutlich, vor allem auch im Vergleich zu den übrigen Messungen, bei denen Dateien variierender Größen zum Herunterladen genutzt wurden. Über die Laufzeit des Szenarios addieren sich diese Lastschwankungen, so daß in Summe nur ca. 90% der Soll-Last tatsächlich erzeugt wurde.

Im Gegensatz dazu fallen die Lasteinbrüche bei variierenden Dateigrößen deutlich kleiner aus. Allerdings fällt in Messung 2, deren Ergebnisse in Abbildung 7.5(a) dargestellt sind, die Abweichung der Ist-Amplitude auf. Die maximale Ist-Last weicht um ca. 35kByte/s von der Soll-Last ab. Verursacht wird dies durch relativ kleine Dateien, deren Download bei einer Bandbreite von 50 kByte/s teilweise weniger als eine Sekunde in Anspruch nimmt. Demzufolge im Testklient für den „Rest“ der Sekunde keine Last mehr generiert.

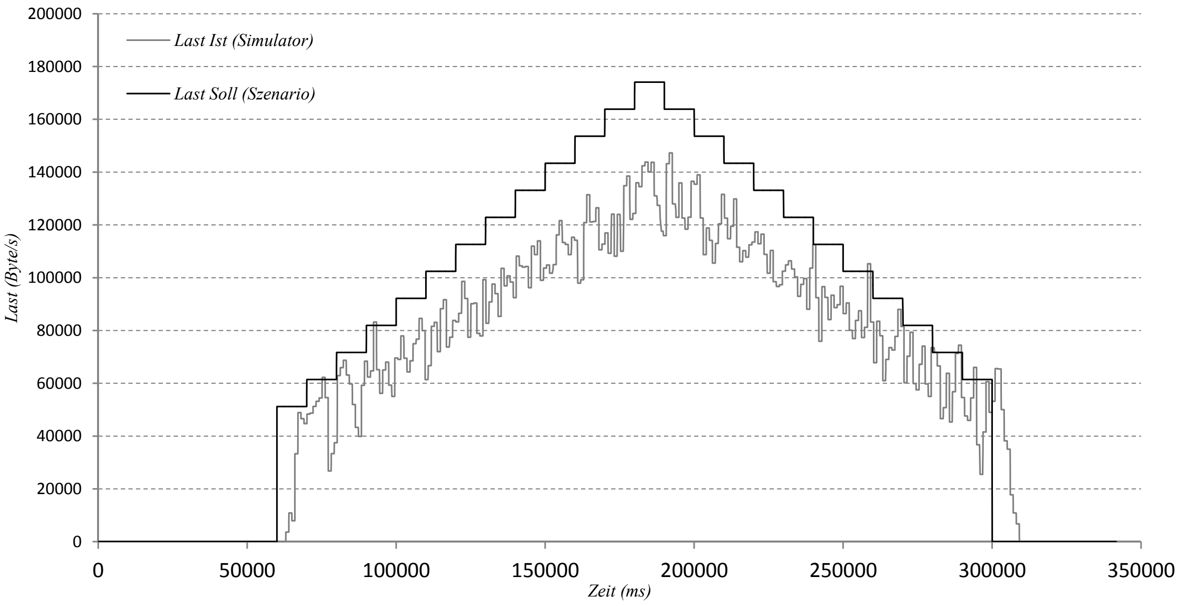


(a) Dienst 1, Einheitliche Dateigröße, Szenario 1

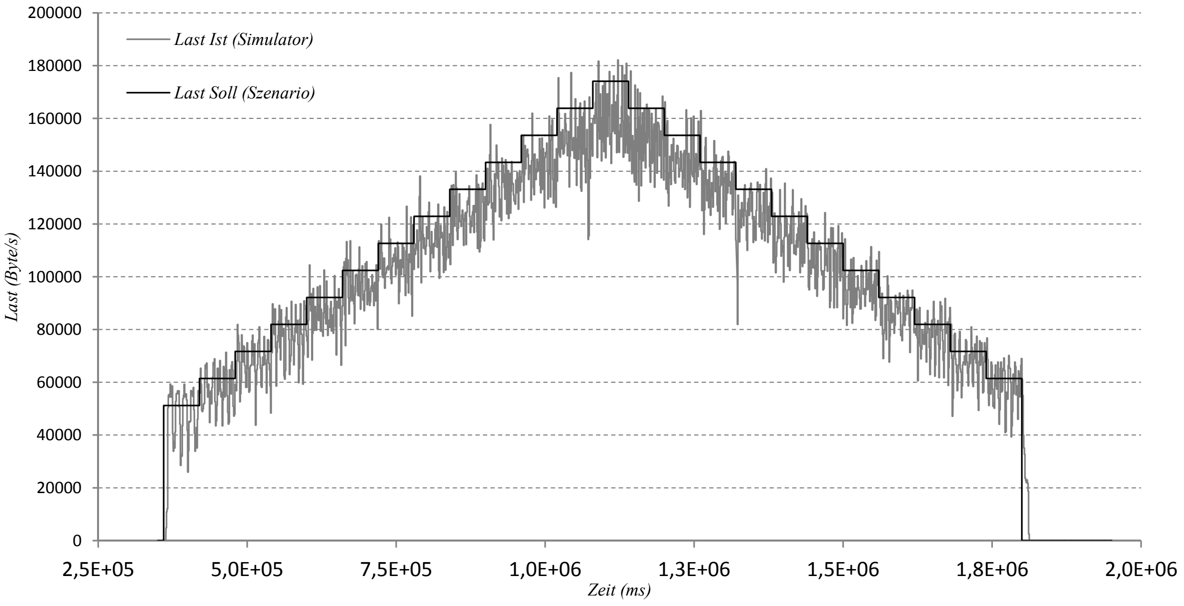


(b) Dienst 1, Differierende Dateigrößen $> 100kB$, Szenario 1

Abbildung 7.4.: Vergleich Soll-/ Ist-Last für Dienst 1, verschiedene Dateigrößen



(a) Dienst 1, Differierende Dateigrößen $< 100kB$, Szenario 1



(b) Dienst 1, Differierende Dateigrößen $< 100kB$, Szenario 2

Abbildung 7.5.: Vergleich Soll-/ Ist-Last für Dienst 1 unter verschiedenen Bedingungen

Dieser Effekt entsteht genaugenommen durch eine Ungenauigkeit des Automatisierungsmoduls, welches bei einer geschätzten Ladezeit von unter einer Sekunde mehr Last generieren müßte. In Summe über die gesamte Laufzeit addieren sich die Abweichungen derart, daß nur 80% der Soll-Last tatsächlich generiert wird. Auch die durchschnittliche Abweichung ist im Vergleich zu den anderen Messungen entsprechend hoch.

Wird die Dateigröße erhöht, daß ein einzelner Ladevorgang deutlich über einer Sekunde liegt, werden auch die entsprechenden Soll-Lasten erreicht. Dies gilt für Messung 3 (Szenario 1) ebenso wie für Messung 4, für das Szenario 2 genutzt wurde. In beiden Messungen wird sowohl die Soll-Last in jedem Schritt als auch eine Gesamtlasterzeugung von rund 95% im Vergleich zum Soll erreicht.

Für die Bewertung des zeitlichen Verhaltens wurde ein zusätzlicher Wert erfaßt. In Tabelle 7.3 gibt der Wert Δt_1 die Zeit an, die vom ersten Anstieg der Soll-Last bis zur Generierung der Last durch den ersten Testklienten vergeht. Δt_2 entspricht der Verzögerung bis zum Erreichen der Soll-Last während Δt_3 dann die Verzögerung zwischen dem Soll-Lastanstieg von 50 auf 60 kByte/s und dem entsprechenden Einstellen der Ist-Last angibt. Die Latenz beim Absenken der Last von 60 auf 0 kByte/s wird schließlich in Δt_4 angegeben.

Die Werte für Δt_1 und Δt_2 variieren in den verschiedenen Messungen für Dienst 1 kaum. Auffällig ist jedoch, daß sie im Vergleich zu den Verzögerungen des ebenso TCP-Socket basierten Dienst 2 (siehe Tabelle 7.1) deutlich höher sind. Ursache hierfür ist das in Dienst 1 zugrundeliegende HTTP-Protokoll, das vor dem eigentlichen Nutzdatenabruf noch Kontroll-Header austauschen muß.

Auffällig sind hingegen die starken Schwankungen der anderen beiden Verzögerungswerte. Bei Δt_3 treten Probleme in der Messung selbst auf. Es kann der Fall eintreten, daß zum Zeitpunkt des Soll-Lastanstiegs die Ist-Last durch abgeschlossene Downloads bereits deutlich unter der ursprünglichen Soll-Last liegt und hier nachgeregelt werden muß. In Folge ist die Verzögerung des Ist-Lastanstieges, der durch die Soll-Laststeigerung ausgelöst wird, nicht eindeutig zu isolieren.

Bei der Verzögerung Δt_4 ist zunächst zu erwarten, daß in Messung 2, bei der kleine, schnell herunterladbare Dateien verwendet werden, auch die Last schnell auf 0 fällt. Im Vergleich zu Messung 3 ist dies auch der Fall. Jedoch ist der absolute Wert mit ca. 10 Sekunden sehr hoch und insbesondere im Verhältnis zu Messung 1 groß. Offensichtlich hängt diese Messung von zahlreichen Parametern ab, wie bspw. den Zuständen der Klienten. Folglich läßt eine Bewertung von Einzelmessungen keine belastbaren Rückschlüsse zu.

7.1.3.3. Anteil der Kontrolldaten

Neben den Lasten, die durch die Implementierungen der Testklienten erzeugt werden, können zusätzliche Lasten durch die Kontrollkommunikation zwischen Testklient und Testservermodul entstehen. Dies ist genau dann der Fall, wenn die Kommunikation über einen der zu validierenden Dienste läuft. Beim Mobilfunkbeispiel ist dies natürlich der Fall in Form von TCP-Sockets, die durch die Testklienten über das Mobilfunknetzwerk hin zum Testservermodul aufgebaut werden.

Da diese Kanäle bidirektional genutzt werden, entstehen also Wechselwirkungen mit allen TCP-basierten Diensten.

Im folgenden wird durch entsprechende Messungen ermittelt, welches Datenvolumen für den Informationsaustausch entsteht und welchen Anteil es an dem in den Szenarien geforderten Lasten hat. Da die Verbindung zwischen Testklienten und Testservermodul über dedizierte Sockets realisiert wird, können die zugehörigen Datenvolumen im Evaluierungssystem einfach zugeordnet werden.

Um das absolute Volumen bewerten zu können, eignen sich Messungen an Dienst 3 besonders, da hier keine TCP-Lasten erzeugt werden. Die Ergebnisse, die in Abbildung 7.6(a) angegeben sind, wurden der gleichen Meßreihe entnommen, deren Ergebnisse in Abbildung 7.2 dargestellt sind.

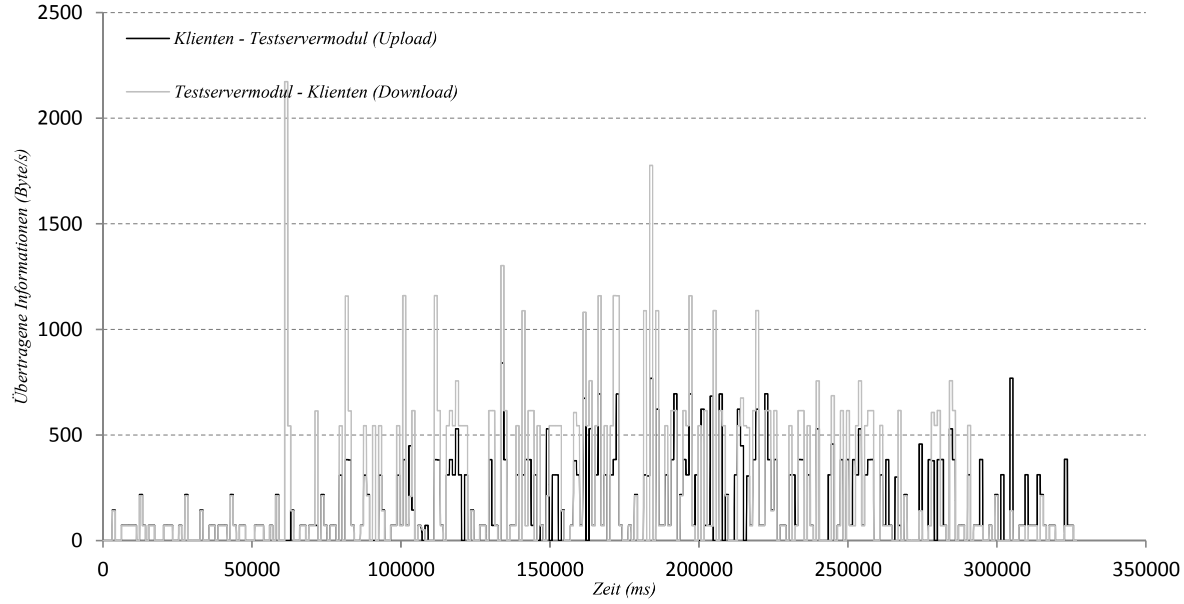
Im Diagramm ist die Summe des Datenvolumens aller Testklienten pro Zeit aufgetragen - getrennt nach Down- und Upload (aus Sicht der Testklienten). Im ersten Bereich, bis ca. 60 Sekunden Szenarienlaufzeit, werden lediglich Pings zwischen dem Testservermodul und den Klienten ausgetauscht. Entsprechend niedrig ist das Aufkommen mit im Schnitt ca. 50 Byte/s. Zum Zeitpunkt 60s steigt im Szenario die Last von 0 auf 5, so daß in diesem Moment fünf Anrufe gestartet werden müssen. Entsprechend hoch ist das Datenaufkommen für einen kurzen Zeitraum. Bei genauerer Betrachtung fällt auf, daß die Volumenspitzen im Download in Klassen von Vielfachen von rund 500 Byte/s eingeteilt werden können. Das Verhalten des Uploads ist chaotischer.

Es gilt zu beachten, daß das absolute Datenvolumen von der Art der verwendeten Dienstimplementierungen abhängt. Bspw. verursacht Dienst 2 wesentlich weniger Kontrollaufwände, da nur Änderungen der zu erbringenden Lasten kommuniziert werden müssen. Im Falle von Dienst 1 müssen hingegen zur Aufrechterhaltung der Last beständig neue Downloads gestartet werden.

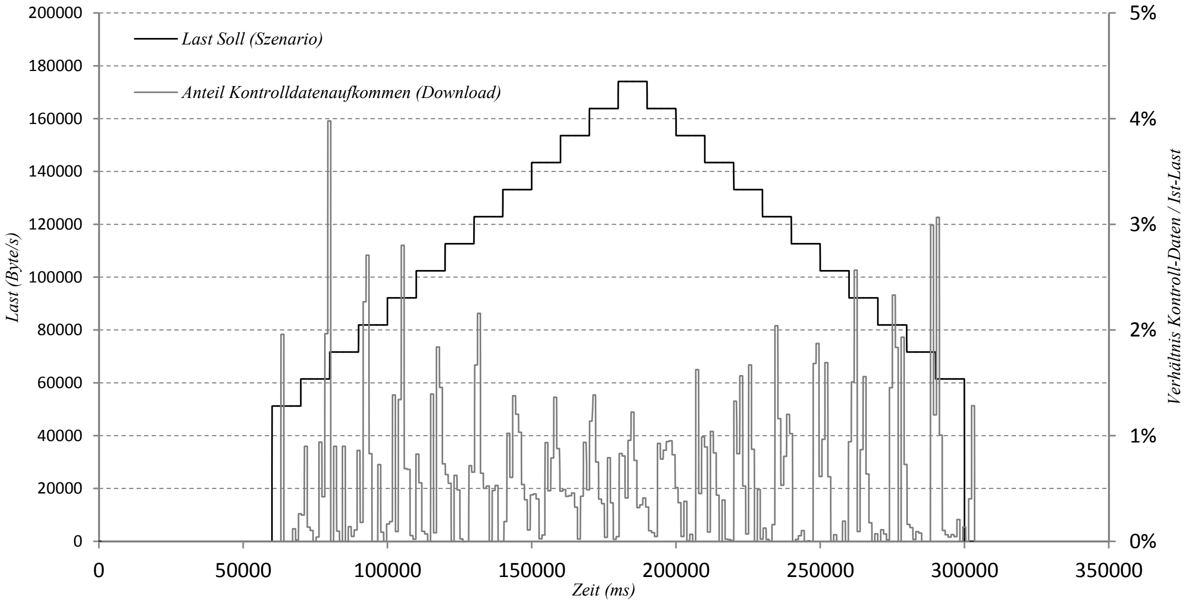
Offensichtlich besteht in diesem Falle eine direkte Abhängigkeit zwischen zu erzeugender Last und den Kontrolldaten. Die Frage ist folglich, inwieweit bei der Überschneidung der verwendeten Technologien für den Dienst und die Kontrollkommunikation eine relevante Beeinflußung der Ist-Last entsteht. In Abbildung 7.6(b) ist deshalb das Verhältnis zwischen dem Kontrolldatenvolumen und dem Datenvolumen für Dienst 1, also der Ist-Last, dargestellt. Zugrunde liegt erneut die Meßreihe aus dem vorangegangen Abschnitt, deren Ergebnisse in Abbildung 7.4(a) zu finden sind. Zur leichteren Nachvollziehbarkeit wurde die Variante mit dem Download identischer Dateien gewählt. Da im Dienst 1 nur das Download-Volumen betrachtet wird, ist der Kontroll-Upload nicht aufgeführt.

Der Anteil der Kontrolldaten ist eher gering - in einigen wenigen Spitzen gerade einmal 4% des gesamten Datenaufkommens. Steigen während der Szenarienlaufzeit Soll- und Ist-Last, fällt der Anteil des Kontrollaufkommens merklich. In der Spitze der Soll-Last (170 kByte/s) fällt der Anteil der Kontrolldaten auf unter 1%. In diesem Falle steigt also das absolute Datenaufkommen mit steigender Last an, das relative fällt jedoch.

Auch diese Entwicklung ist abhängig vom jeweiligen Dienst und der darauf definierten Last sowie dem jeweiligen Szenario. Für das Mobilfunkbeispiels und die in den Messungen verwendeten



(a) Absolutes Kontrolldatenaufkommen, Steuerung von Dienst 3



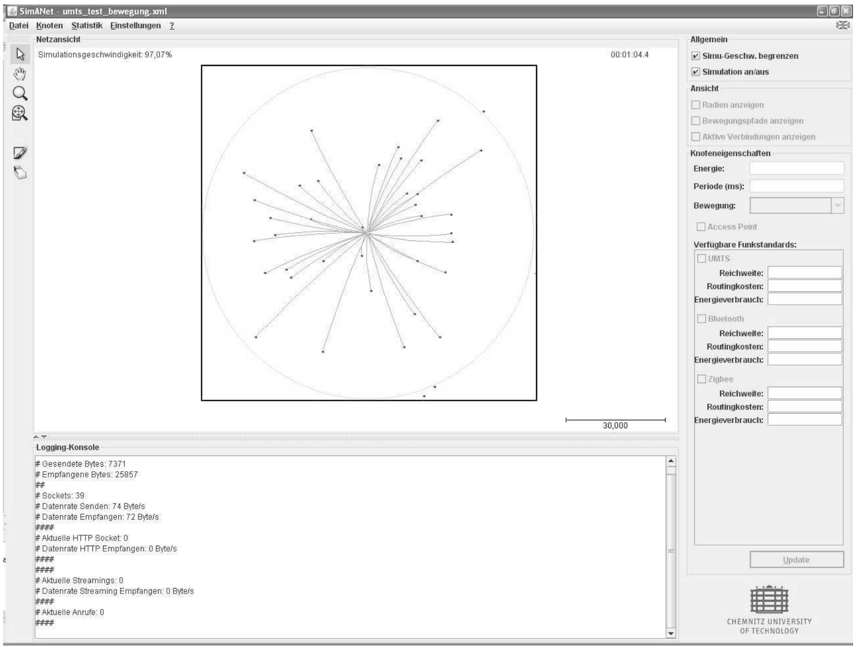
(b) Relatives Kontrolldatenaufkommen (Download), Anteil an Last von Dienst 1

Abbildung 7.6.: Analyse Kontrolldatenaufkommen

Szenarien kann das Kontrolldatenaufkommen vernachlässigt werden, da es im Vergleich zu technologischen und konzeptionellen Schwankungen im Ist-/ Soll-Lastverhältnis klein ist.

7.1.4. Dynamische Netze

Eine der elementaren Anforderungen an das vorgestellte Validierungskonzept ist die Handhabung dynamischer und heterogener Klientenmengen. Die oben aufgeführten Messungen der einzelnen Dienste spiegeln jedoch nur ein absolut statisches Netz wieder, was insbesondere dem Beispiel eines Mobilfunknetzwerkes kaum gerecht wird. Im folgenden werden deshalb Ergebnisse präsentiert, in denen die Klienten-Struktur einer Dynamik während der Laufzeit unterliegt.



☑ Simu-Geschw. begrenzen

☑ Simulation an/aus

```
# Gesendete Bytes: 7371
# Empfangene Bytes: 25857
##
# Sockets: 39
# Datenrate Senden: 74 Byte/s
# Datenrate Empfangen: 72 Byte/s
####
# Aktuelle HTTP Socket: 0
# Datenrate HTTP Empfangen: 0 Byte/s
####
####
# Aktuelle Streamings: 0
# Datenrate Streaming Empfangen: 0 Byte/s
####
# Aktuelle Anrufe: 0
####
```

Abbildung 7.7.: Netzwerkkonfiguration in SimANet für ein dynamisches Klientennetz

Einfach zu bewerkstelligen und auch entsprechend nachvollziehbar ist es, durch den Simulator die Verfügbarkeit der Klienten zu verändern. Im Simulator wird dazu ein Netzwerk nachgestellt, wie es in Abbildung 7.7 dargestellt ist. Das Quadrat stellt einen Raum dar, dessen Kanten mechanische Barrieren sind und in dessen Mitte sich ein Zugangspunkt befindet. Aufgrund der Charakteristik des Zugangspunktes deckt dieser die Ecken des Raumes nicht ab, so daß Klienten, die sich außerhalb des Empfangsradius befinden, keine Dienste nutzen können.

Die Dynamik wird durch Simulation von Bewegungen der einzelnen Klienten in die Messreihen gebracht. SimANet unterstützt hierzu verschiedene Bewegungsmodelle, die den Knoten zugeordnet werden können. Im Falle der folgenden Messungen wird das Markovsche Bewegungsmodell verwendet. Dieses erlaubt für bewegte Knoten Richtungsänderungen innerhalb bestimmter Grenzen und kann so ein teils realistisches Bewegungsmuster von Personen nachbilden.

Die Anzahl der Klienten wird auf 40 erhöht, um im Laufe der Simulation beständig genügend Klienten zur Absicherung des Szenarios zur Verfügung zu haben. Zu Beginn der Simulation befinden sich alle Klienten im Kommunikationsradius des Zugangspunktes. Das Szenario wird geändert, so daß über die gesamte Simulationslaufzeit eine konstante Last für den jeweiligen Dienst erzeugt wird.

Untersucht werden, getrennt für alle drei Dienste, jeweils zwei Abläufe. Zunächst wird eine Referenzmessung durchgeführt, in der sich die Knoten nicht bewegen. Nach fünf Minuten Simulationszeit werden manuell 15 zufällig ausgewählte Klienten aus dem Empfangsbereich des Zugangspunktes entfernt. Die Ergebnisse sind in den jeweils ersten Diagrammen (a) in Abbildung 7.8 bis 7.10 dargestellt.

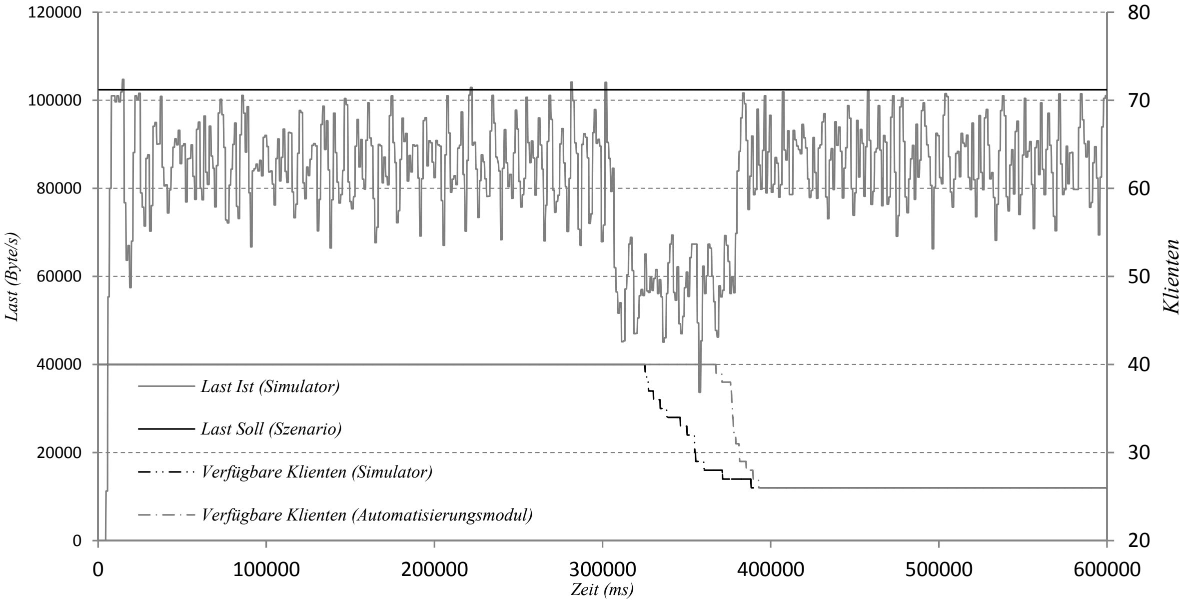
Neben der Soll- und Ist-Last ist jeweils die Zahl der verbundenen Klienten aufgeführt. Hier wird unterschieden, wie viele Klienten im Simulator tatsächlich in Reichweite des Zugangspunktes sind und der Anzahl der Klienten, die im Modell des Automatisierungssystems verfügbar sind. Im Simulator wird dazu der Zustand des Testklienten abgefragt. Nachdem 40 Sekunden lang kein Ping vom Server empfangen wurde, ändert dieser den Zustand auf verbindungslos.

Um die Verfügbarkeit von Testklienten zu ermitteln, pingt das Testservermodul alle 30 Sekunden die Klienten an. Wird nach 30 Sekunden keine Bestätigung empfangen, wird der Klient aus der Liste der verfügbaren Klienten entfernt. Da eine Kommunikation zwischen ihnen durch Verlust des Kommunikationsnetzes nicht mehr möglich ist, entsteht zwischen dem Verbindungsverlust des Klienten im Simulator und der Erkennung dieser Änderung im Automatisierungsmodul eine Latenz von bis zu 40 Sekunden.

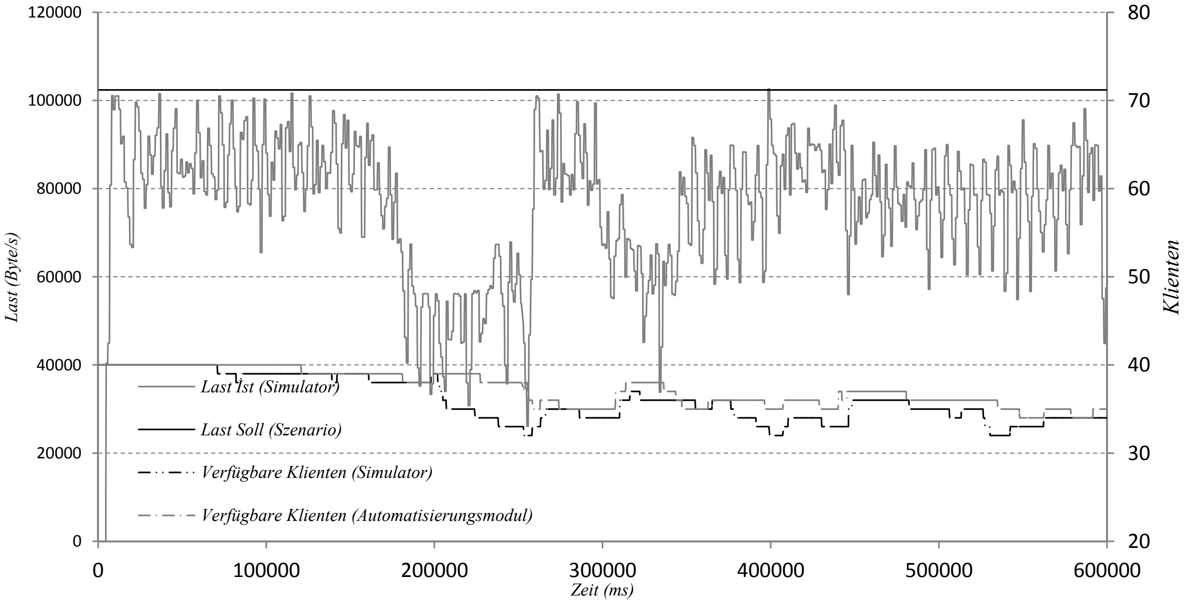
Das daraus resultierende Verhalten ist in den Diagrammen (a) der Abbildungen 7.8 bis 7.10 zu erkennen. Nachdem bei ca. 300 Sekunden Simulationszeit 15 der Klienten entfernt wurden, fällt zunächst die generierte Last des zugehörigen Dienstes. Im Laufe der Zeit ändern die Testklienten ihren Zustand, was jedoch vom Testservermodul noch nicht registriert wird. Zu diesem Zeitpunkt erhält folglich das Testservermodul keinerlei Aktualisierungsmitteilungen der getrennten Klienten, so daß eine Anpassung der Ist-Lasten nicht stattfindet. Erst nach Erreichen entsprechender Timeouts wird durch erneute Testpartitionierung die Lastdifferenz auf die verfügbaren Klienten verteilt.

Die Korrektur des Lasteinbruchs nach Entfernen der Klienten dauert bei allen Diensten ca. 70 bis 80 Sekunden. Bei der Messreihe für Dienst 3, die in Abbildung 7.10(a) dargestellt ist, fällt der massive Abfall der Ist-Last zum Zeitpunkt von rund 380 Sekunden auf. Dieser entsteht durch eine zufällige Überlappung der bereits vorab bestehenden Differenz in Folge der getrennten Klienten und einiger beendeter Anrufe in Folge der abgelaufenen Anrufdauer.

Der kurzfristige Verlust derartig vieler Klienten wird praktisch selten vorkommen. Eine kontinuierliche Fluktuation weniger Klienten ist weitaus wahrscheinlicher. So können Mobiltelefone von den Nutzern in Funklöcher getragen werden oder schalten durch leere Akkumulatoren ab. Dieser Sachverhalt wird durch Nutzung des Bewegungsmodells analysiert.

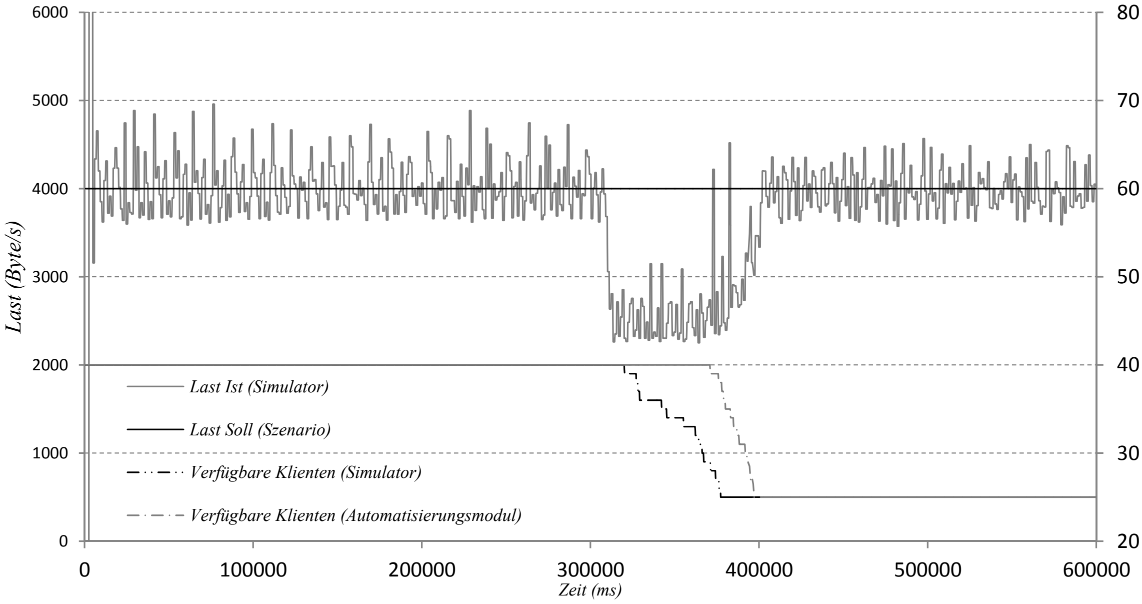


(a) Dienst 1, Referenz

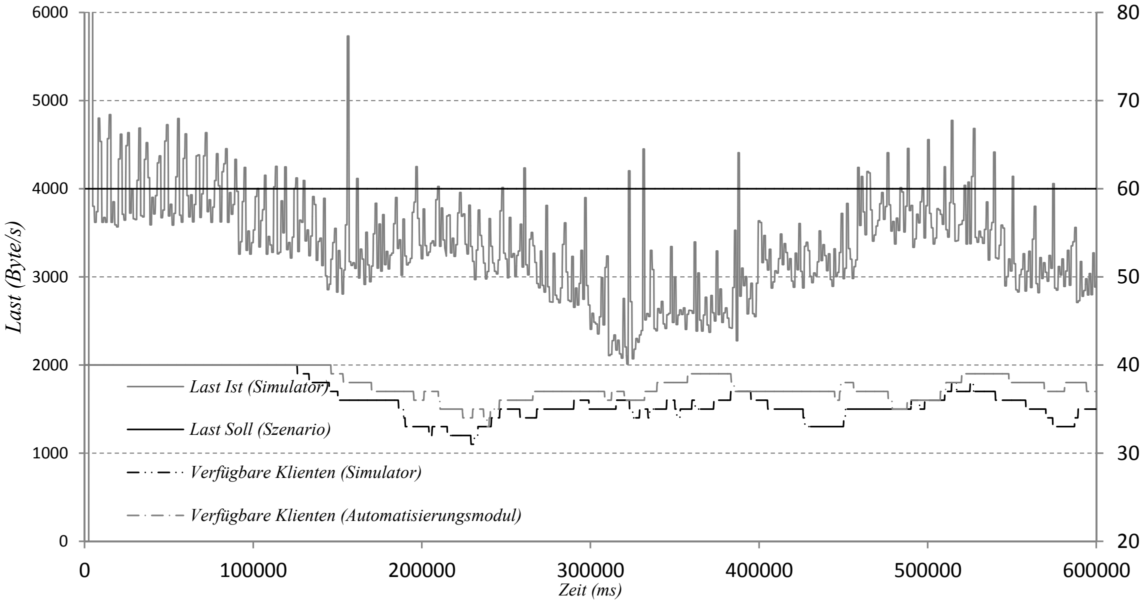


(b) Dienst 1, Dynamisch

Abbildung 7.8.: Vergleich Soll-/ Ist-Last für Dienst 1 bei dynamischen Klientnetzwerken

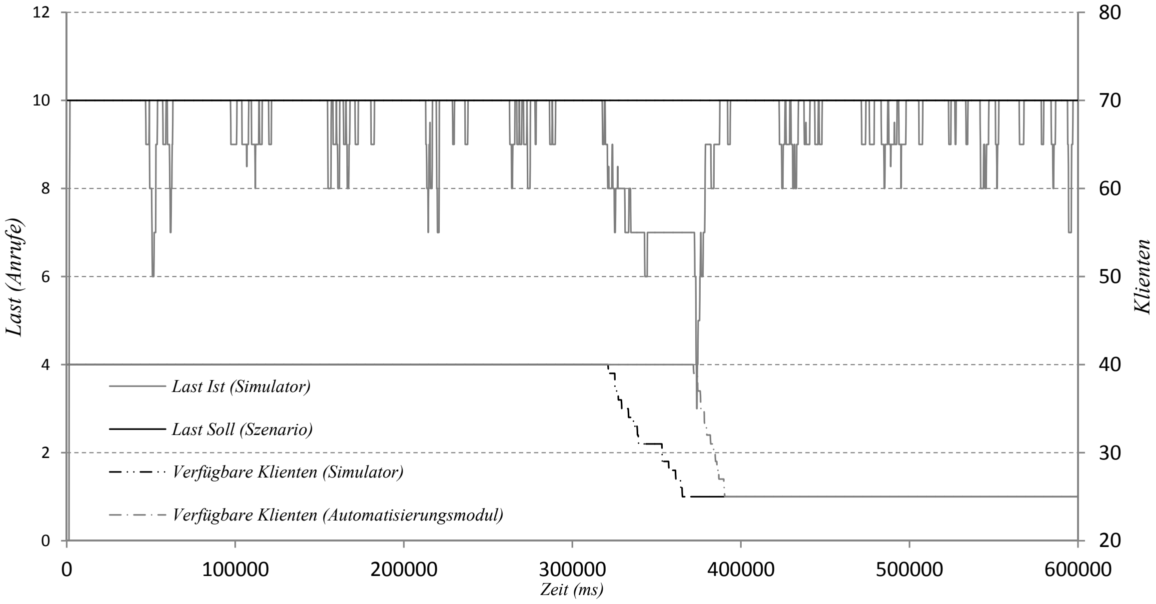


(a) Dienst 2, Referenz

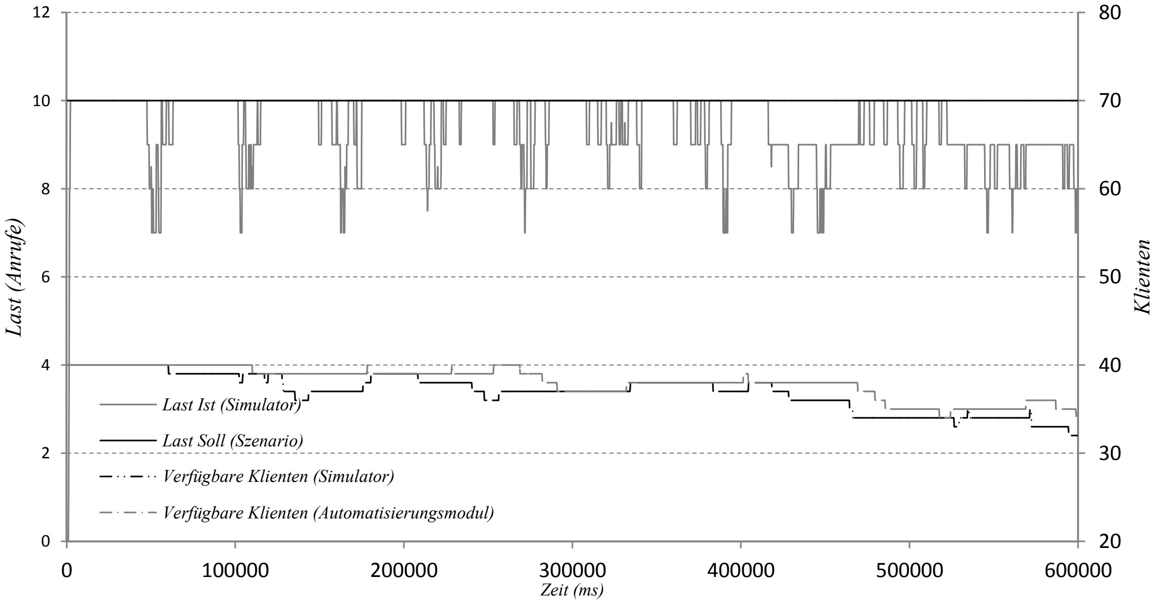


(b) Dienst 2, Dynamisch

Abbildung 7.9.: Vergleich Soll-/ Ist-Last für Dienst 2 bei dynamischen Klientnetzwerken



(a) Dienst 3, Referenz



(b) Dienst 3, Dynamisch

Abbildung 7.10.: Vergleich Soll-/ Ist-Last für Dienst 3 bei dynamischen Klientnetzwerken

Die Messungen werden auch hier getrennt für jeden Dienst durchgeführt. Zu Beginn der Messungen befinden sich alle Klienten im Empfangsbereich des Zugangspunktes. Die Ergebnisse sind in den Diagrammen (b) der Abbildungen 7.8 bis 7.10 dargestellt. Auffällig sind die deutlichen Unterschiede in den Auswirkungen der Dynamik zwischen den einzelnen Diensten. Während es bei Dienst 1 bei ca. 200 Sekunden fast zu einer Halbierung der Ist-Last kommt, sind für Dienst 3 über den gesamten Verlauf kaum Abweichungen zur Referenz zu erkennen. Ursache hierfür ist das zufällige Bewegungsverhalten der Knoten und die variierende Relevanz der Knoten für den jeweiligen Dienst. Insbesondere für Dienst 3 ist die Fluktuation recht gering und kontinuierlich. Im Gegensatz dazu sind die Fluktuationen bei Dienst 1 insbesondere zu Beginn groß und kurzfristig. Wenn dann wegen der Latenzen die Differenz zwischen den im Simulator und im Automatisisierungssystem verbundenen Klienten groß wird und die Differenz zufällig auch noch durch Klienten entsteht, die gerade Last für den Dienst generieren, entstehen in Summe entsprechend große Lasteinbrüche.

Auffällig und durchaus überraschend ist, daß auch eine Fluktuation mit wenigen Klienten zu ähnlich hohen Lasteinbrüchen führen können, wie in der Referenzmessung. Eine Verringerung der Latenzen bzw. Optimierung der Erkennung von Verbindungsverlusten sollte ein vielversprechender Ansatz zur Verbesserung sein.

7.2. Qualitative Bewertung

Während im zurückliegenden Abschnitt eine Betrachtung absoluter Werte, die durch Messungen erlangt wurden, im Mittelpunkt stand, schließen sich nun Diskussionen weiterer Eigenschaften an, die schwer gemessen werden können oder sich implizit aus dem Konzept ergeben.

7.2.1. Ist-Last Abschätzung

Eines der wichtigsten Probleme stellt die Ermittlung der tatsächlich erzeugten Last dar. Diese ist für die Qualität des Konzeptes, also der Annäherung der Ist-Last an die in den Szenarien vorgegebene Soll-Last, maßgeblich. Im Gegensatz zur Simulation kann diese global für das gesamte SUT nur schwer bzw. nicht ermittelt werden. Die Herausforderung liegt folglich darin, daß das interne Modell des Automatisierungsmoduls, auf dem die Lasten der einzelnen Klienten berechnet werden, möglichst genau die realen Lasten eines jeden Klienten abbildet.

Das Automatisierungsmodul hinterlegt den Wert der aktuellen Last eines Klienten i für den Dienst s zum Zeitpunkt t in der Variable ${}^{t}l'_{s,i}$ und errechnet im Falle notwendiger Änderungen die Laständerung ${}^{t}l^{+}_{s,i}$ (siehe Gleichung 5.9). Die Frage ist nun, wie sich aus diesen Größen die tatsächliche Ist-Last des Klienten ableiten läßt und ob diese bis zur nächsten Änderung $t+1$ tatsächlich konstant bleibt.

Die einfachste Möglichkeit ist, die aktuelle Last eines Klienten direkt aus der berechneten Testpartitionierung abzuleiten. Damit ergäbe sich die aktuelle Last zum Zeitpunkt $t + 1$ wie in der Gleichung 7.6 angegeben.

$$^{t+1}l'_{s,i} \;=\; ^{t+1}l_{s,i} = {}^{t}l'_{s,i} + {}^{t}l^{+}_{s,i} \tag{7.6}$$

Dies ist nur möglich, wenn durch die entsprechende Umsetzung auch sichergestellt werden kann, daß der Testklient diese Last auch zuverlässig erzeugt. Im Falle des simulierten Beispiels ist das bei Dienst 2, also dem Streaming-Dienst, möglich. Es ist offensichtlich, daß diese Variante der Ist-Lastabschätzung die präziseste ist und keine Aufwände oder Zeitverzögerungen verursacht.

Für den Dienst 3 trifft die Annahme der technischen Umsetzbarkeit aber nur noch bedingt zu. Ein gestarteter Anruf wird nach einer vordefinierten Zeit beendet, so daß die aktuelle Last $^{t+1}l'_{s,i}$ des betreffenden Klienten durch ein externes Ereignis angepaßt werden muß. Umgesetzt wird dies durch Berichte, die Klienten bei der Änderung eines Testzustands an das Testservermodul senden. Dieses wiederum leitet die Informationen an das Testautomatisierungsmodul weiter, wo die aktuelle Last entsprechend asynchron angepaßt wird. Da die Übertragung der Berichte mit Latenzen behaftet ist, gibt es in diesem Falle zwischen der Änderung der Ist-Last im SUT und der Aktualisierung im Automatisierungsmodul Verzögerungen.

Für Dienst 1 hingegen ist die Annahme der technischen Umsetzbarkeit nicht mehr gegeben. Durch einen gestarteten HTTP-Download wird im Download ein Datenvolumen produziert. Die Höhe hängt jedoch in erster Linie von der Qualität der drahtlosen Verbindung ab, also von Parametern der Bitübertragungsschicht. Diese kann kaum beeinflußt werden. Die tatsächliche Ist-Last ergibt sich also nur unter Berücksichtigung eines Korrekturparameters μ, wie es in Gleichung 7.7 angegeben ist.

$$^{t+1}l'_{s,i} \;=\; {}^{t}l'_{s,i} + {}^{t}\mu_{s,i} \cdot {}^{t}l^{+}_{s,i} \tag{7.7}$$

Dieser Parameter μ ist natürlich abhängig vom Dienst und dem Klienten und kann auch über die Zeit variieren. Wie der Wert bestimmt wird, hängt stark von der Charakteristik des Dienstes, der darauf definierten Last und der technischen Umsetzung ab. Denkbar sind sowohl statistische Verfahren wie auch Messungen und Berichte durch die Testklienten.

Im Falle der Simulation sind die virtuellen TCP-Sockets für Dienst 1 auf eine Bandbreite von 10 kByte/s festgesetzt. Demzufolge gilt in diesem einfachen Falle $^{t}\mu_{s,1} = 1$ für alle Klienten s zu jedem Zeitpunkt t. Auch die Aktualisierung der aktuellen Last durch Klientenberichte wird benötigt, um das Ende eines Downloads an das Automatisierungsmodul zu melden.

In Summe können alle genannten Varianten zur Berechnung der aktuellen Last $^{t+1}l'_{s,i}$ auf Gleichung 7.7 vereinfacht werden und entsprechende Sonderfälle zur asynchronen Aktualisierung oder direkten Einstellung der Last durch entsprechende Definition von $^{t}\mu_{s,i}$ umgesetzt werden.

Verliert ein Testklient die Verbindung zum Testserver, bspw. durch ein fehlendes Mobilfunknetzwerk, wird dies vom Testservermodul auf die Einschränkungsmatrix $\mathcal{G}$ (siehe Gleichung 5.4) abgebildet und muß daher nicht für die Ist-Lastabschätzung betrachtet werden.

Mit den drei Diensten des simulierten Beispiels sind auch drei Varianten für die Ist-Lastabschätzung abgedeckt. Es ist durchaus möglich, daß andere SPS und daraus folgende Implementierungen auch weiterführende Methoden oder Modelle bedingen. Ähnliche Probleme existieren auch im Bereich der Lastverteilung für parallele Rechnersysteme. Bspw. werden in [59] (S. 51) zwei Kriterien zur Ermittlung der Ist-Situation herausgestellt: Aktualität und Exaktheit. Während ersteres die Frage betrachtet, wie häufig die Zustände überhaupt aus dem zu überwachenden System ermittelt werden bzw. ermittelt werden können, hinterfragt das Kriterium Exaktheit, ob die Zustände überhaupt präzise bestimmt werden können bzw. in Folge der Aktualität nicht zu jedem Zeitpunkt hinreichend präzise sind (Vgl. auch [83]).

Besonders bei der Anwendung des Validierungskonzeptes für heterogene Systeme kommt noch ein drittes Kriterium hinzu: die Ressourcen die benötigt werden, um die Kriterien von Aktualität und Exaktheit umzusetzen. Die oben vorgestellten Varianten zur Abschätzung der Ist-Last stellen, zumindest für das Mobilfunkbeispiel, einen vertretbaren Kompromiß aus allen drei Kriterien dar.

7.2.2. Abdeckung

Das vorgestellte Konzept dient der Validierung von dienstbereitstellenden Systemen. Die im Kapitel 6 eingeführte Implementierung des Mobilfunkbeispiels realisiert einen Teil der zu Beginn gestellten Anforderungen (Vgl. Abschnitt 4.3).

Bereits in Ausbaustufe 1 sind die Strukturen der Testbench geschaffen worden, die den Zugriff der Testklienten auf die Dienste des SPS erlauben. Das Testservermodul verwaltet eine variierende Menge von Testklienten, deren Zustände sowie deren laufende und abgeschlossene elementaren Testaufträge. Über eine graphische Nutzeroberfläche ist die Darstellung aller Testklienten möglich. Es können elementare Testaufträge erstellt und an die entsprechenden Klienten gesendet werden. Über Skripte können statische Aufträge automatisiert wiederholt werden. Die Oberfläche zeigt auch die Berichte von abgeschlossenen Testaufträgen an.

Mit diesem Stand ist es folglich möglich, ferngesteuert Aktivitäten auf den Klienten zu starten. Die Aktivitäten selbst müssen als Logik auf den Klienten hinterlegt sein. Die Verwaltung einer großen Zahl an Klienten ist jedoch praktisch kaum möglich, da die zur Validierung des SUT notwendigen elementaren Testaufträge manuell erstellt und die einlaufenden Berichte entsprechend ausgewertet werden müssen. Zur Ausführung von Testszenarien müßte der Nutzer jede Änderung von Soll- oder Ist-Last manuell behandeln.

In Folge wurde mit Ausbaustufe 2 zumindest prototypisch das Testautomatisierungssytem eingeführt, das die automatische Ausführung der Testszenarien erlaubt. Änderungen im Verhältnis zwischen Soll- und Ist-Last werden durch Testpartitionierung automatisch korrigiert. Die instanzierten Modelle sind in der Lage, zeitliche Dynamik und heterogene Ressourcen der einzelnen Klienten zu beschreiben und damit auch in den automatischen Prozessen berücksichtigt zu werden. Mit dieser Funktion ist die Verwaltung einer großen Zahl an Klienten und auch Diensten überhaupt erst möglich, da während der Ausführung der Testszenarien kein oder kaum manueller Eingriff notwendig ist.

In jedem Fall gilt, daß die Bewertung des korrekten Verhaltens von Diensten explizit in der Implementierung des Testklienten vorgenommen wird. Eine Dienstanfrage mit einer gewissen Last kann erfolgreich ausgeführt werden - oder eben nicht. Die Kategorisierung obliegt dem Testklienten und wird durch entsprechende Rückmeldung an das Testservermodul gespeichert. Ggf. können auch Gütefaktoren ermittelt und gespeichert werden.

In Summe konnten damit alle funktionalen Anforderungen, die in Abschnitt 4.3 aufgeführt sind, im Konzept berücksichtigt und in großen Teilen in der prototypischen Implementierung des Mobilfunkbeispiels umgesetzt werden. Es führen jedoch einige Details der Implementierungen zu Einschränkungen.

Mit dem Stand dieser Arbeit ist die Einschränkungsmatrix $\mathcal{G}$, die in Gleichung 5.4 definiert wurde, nicht durchgängig umgesetzt und kann folglich auch nicht genutzt werden. Die Verfügbarkeit von Klienten wird nur über interne Datenstrukturen des Testserver- und des Automatisierungsmoduls verwaltet und kann während der Ausführung von Testszenarien nicht beeinflußt werden. Desweiteren fehlt in der aktuellen Variante noch ein Auswertungsmodul samt graphischer Oberfläche zur Überwachung und Definition aller Modellparameter.

7.2.3. Übertragbarkeit und Grenzen

Ein Großteil der bisherigen Betrachtungen konzentrieren sich auf das Mobilfunkbeispiel und das zur Generierung der Meßergebnisse verwendete Simulatorsystem. Im folgenden wird diskutiert, wie und mit welchen Einschränkungen die Ergebnisse und das Konzept auf weitere SPS übertragen werden können. Dazu ist zunächst zu klären, was mit dem Konzept validiert werden kann und wie die Ergebnisse helfen können, Probleme in den SUTs zu bewerten.

In aller Regel werden durch die Kapselung des SUT und die Betrachtung der Dienste sowie der darauf definierten Lasten rein funktionale Eigenschaften des SUT ausgeblendet. Ein klassischer Vergleich von Eingaben und generierten Ausgaben mit zu erwartenden Ausgaben, wie er bei den Tests auf verschiedenen Ebenen des V-Modells angedacht ist, ist nicht möglich.

Vielmehr ist das Konzept einsetzbar, um langläufige Belastungen von Systemen durch heterogene und fluktuierende Strukturen von Klienten nachzustellen bzw. unter kontrollierten Bedingungen überhaupt ausführen zu können. Aufgrund der verteilte, parallelen Wirkung der Klienten sind deshalb vor allem folgende Mängel in den zu testenden Systemen auffindbar:

- Ressourcenmangel (bspw. Netzwerkkapazitäten, Rechenleistung)
- Ressourcenkonflikte / Abhängigkeiten (zwischen Diensten)

- Parallelisierungseffekte (bspw. durch parallele Dienstzugriffe)
- Laufzeiteffekte (bspw. nicht freigegebene Ressourcen)

Ein Großteil dieser Mängel sind auf Ressourcen oder Parallelisierung zurückzuführen. Die Verschiedenartigkeit führt jedoch dazu, daß sich jeder Mangel durch ein individuelles aber schwer zu verallgemeinerndes Fehlerbild zeigt. Es liegt demzufolge in der Erfahrung der Testingenieure bzw. Entwickler, aus Auffälligkeiten in der Testszenarienausführung auf entsprechende Ursachen zu schlußfolgern. In jedem Falle ist es dafür hilfreich, Zugriff auf Laufzeitinformationen des SUT selbst zu haben und zumindest zeitliche Relationen zwischen Validierungsergebnissen und Systemzuständen ziehen zu können.

Effekte, die die Ausführung der Testszenarien beeinflussen jedoch ihre Ursache nicht in Fehlern des SUT oder der Testklienten haben, müssen für eine sinnvolle Ursachenforschung ausgeschlossen oder zumindest bekannt sein. Naheliegend sind bspw. externe Quellen, wie Server, die vom Testgenerator als Kommunikationsendpunkte gewählt werden, aber u.U. aufgrund mangelnder Ressourcen nicht in der Lage sind, die Lasten zu erbringen. Auch abgebrochene Tests durch ausfallende Klienten, bspw. durch Funklöcher oder leere Batterien, müssen als solche erkannt werden. Möglich wird dies bspw. durch entsprechendes Logging der Zustände.

Eine interessante Erfahrung ergab sich im Laufe der Implementierung der Testbench. Bei der Umsetzung von Ausbaustufe 2 und der Ankopplung an die Simulation wurden nur durch die Möglichkeit, per Simulator eine große Zahl an Klienten bereitzustellen, einige Fehler im Testservermodul gefunden. Diese waren alle auf Parallelisierungseffekte zurückzuführen und sind bei Tests im realen Mobilfunknetzwerk auf Grund der geringen Zahl an Klienten nicht gefunden worden.

Details des Konzeptes bzw. der eingeführten Modelle führen jedoch auch zu Einschränkungen. Da der Fokus auf der Validierung der dienstbereitstellenden Systeme liegt, ist die Nutzung von Diensten zwischen Klienten schwer modellierbar. Technisch ist es möglich, zwei Klienten i und j in einen Testauftrag zu integrieren. Denkbar wäre bspw., daß i den Klienten j anruft. Jedoch sind die Auswirkungen auf die aktuelle Last der beiden beteiligten Klienten schwer modellierbar, da eine Implikation der Last in Klient i zu Klient j bei einer Kommunikation beider im Modell nicht vorgesehen ist.

Damit fallen einige Dienste, bspw. Peer-To-Peer basierte Angebote, aber auch Technologien, bspw. Ad-Hoc Netzwerke, aus der Betrachtung für das Validierungskonzept heraus - zumindest, bis das Modell und der Testpartitionierungsalgorithmus entsprechend angepaßt ist.

Allerdings beschränkt sich die Nutzung nicht auf reine Klienten-Server-Beziehungen. Auch serverseitig initiierte Dienste lassen sich etwas trickreich auf das gegenwärtige Modell abbilden. Hierzu ist für die Testklienten ein „inverser“ Dienst mit der zugehörigen Last zu definieren und das externe System zur Initiierung der Dienstanfragen über den Testgenerator zu integrieren.

Sollen bspw. eingehende Anrufe auf einem Klienten umgesetzt werden, ist dort eine entsprechende Implementierung im Testklienten vorzunehmen, die „Anrufbereitschaft“ herstellt.

Im Automatisierungmodul sind entsprechend die Kostenfunktionen und Parameter für diese Dienst-Klient-Kombination zu hinterlegen. Soll ein Testklient einen Anruf empfangen, muß der Testgenerator neben dem Testklienten auch das System, welches den Anruf initiiert, kontaktieren und einen Anruf der entsprechenden Nummer auslösen.

Neben all diesen funktionalen Betrachtungen sind einige Sicherheits- und Akzeptanzfragen bisher weitestgehend ausgeklammert worden. Allerdings müssen solche Fragen organisatorisch und technisch berücksichtigt werden - vor allem für den Fall, daß die Klienten im SPS sowohl als Testklienten in der Testbench als auch als Nutzergeräte durch Personen verwendet werden. Die technischen Möglichkeiten der Mobilfunk-Testklienten entsprechen bspw. in etwa der eines Computervirus. Gelingt es Externen, Kontrolle über die Klienten zu erlangen, können Attacken auf das SPS oder den Klienten selbst gestartet werden. Entsprechend müssen Sicherheitsmaßnahmen, auf welcher Ebene auch immer, eingeführt werden, die dies verhindern. In Ausbaustufe 1 werden Nachrichten zwischen Testservermodul und Testklienten mit einer Signatur versehen, so daß zumindest ein Mindestmaß an Sicherheit besteht.

Parallel müssen die Nutzer der Klienten motiviert werden, den Testklient zu akzeptieren. Hierzu sind durch organisatorische oder auch technische Mittel insbesondere Bedenken bezüglich Sicherheit und Datenschutz auszuräumen. Ggf. kann, je nach Einsatzgebiet, auch dienstlich die Nutzung durchgesetzt werden.

Unabhängig von all diesen Überlegungen gilt es, Nutzen und Aufwand in Relation zu stellen. Für jedes SUT sind massive Anpassungen der Testbench notwendig. Insbesondere die Testklienten müssen an die jeweiligen Dienste und Klientenplattform angepaßt werden. Ggf. muß auch im Testservermodul der Zugriff auf das Kommunikationssystem zu den Testklienten adaptiert werden. Die Definition der Modellparameter, wie Kostenfunktionen, Einschränkungen etc., ist ebenso aufwändig. In Folge ist der Einsatz nur für Szenarien gerechtfertigt, die diesen Aufwand aus welchen Gründen auch immer rechtfertigen. Dies wird häufig für Tests mit sehr langer Laufzeit der Fall sein, in denen sich Testszenarien über mehrere Tage oder Wochen ziehen. Hier schlagen auch die im vorangegangenen Abschnitt beschriebenen Verzögerungen unter dynamischen Bedingungen nicht derart auffällig zu buche, wie in den kurzfristigen Szenarien, die für die Messungen verwendet wurden.

7.3. Zusammenfassung

Auf Basis des Mobilfunkbeispiels und der Implementierung, die im vorherigen Kapitel vorgestellt wurde, sind in diesem Kapitel Evaluationen und Messungen aufgeführt sowie verschiedene Eigenschaften des Validierungskonzeptes diskutiert worden.

Die Messungen wurden sowohl unter statischen wie auch unter dynamischen Bedingungen durchgeführt. In beiden Fällen stand das Verhältnis zwischen vorgegebener Soll-Last und tatsächlich generierter Ist-Last im Fokus der Analysen. Für drei Dienste, die sich in der technologischen Umsetzung der Dienstnutzung stark unterscheiden, wurden Messungen über einen relativ kurzen und einen sehr langen Zeitraum durchgeführt. Die Messungen zeigen, daß die generierten Ist-Lasten über die gesamte Zeit im ungünstigsten Falle 20% unter den Soll-Lasten liegen, im Normalfall die Abweichungen jedoch nicht mehr als 10% betragen.

Im dynamischen Fall, wenn also Klienten während der Ausführung der Validierung ausfallen, konnte gezeigt werden, daß sobald das Testautomatisierungsmodul die Abweichungen erkennt, auch korrekt und ohne nennenswerte Latenzen nachregelt. Einzig die technologieabhängige Frage, wann ein Klient als ausgefallen detektiert wird, entscheidet über die Dauer der Ist-/ Soll-Lastabweichungen.

Mit den Ergebnissen, die in diesem Kapitel vorgestellt wurden, konnte gezeigt werden, daß das Validierungskonzept technisch umsetzbar ist und für das Beispiel des Mobilfunknetzwerkes gute Ergebnisse liefert.

8. Zusammenfassung

Mit den folgenden Ausführungen werden die Kernthemen der Arbeit herausgestellt und ein Ausblick auf nötige bzw. sinnvolle Weiterentwicklungen gegeben.

Prinzipiell teilt sich die Arbeit in drei Bereiche. Zum einen wurde die Konzeption des Validierungsvorganges erarbeitet. Dies beeinhaltet die Definition eines Modells zur Beschreibung des Systems und der zur Validierung notwendigen Tests. Zum zweiten wurde eine allgemeine Testbencharchitektur konzipiert, mit deren Hilfe die automatisierte Ausführung der Validierung möglich ist. Kernkomponente ist ein Algorithmus zur Lösung des Testpartitionierungsproblems. Die Testbench wurde schließlich für ein Beispiel zum Test einer Mobilfunkinfrastruktur implementiert und evaluiert.

8.1. Validierungskonzept

Im Fokus der Arbeit steht ein Validierungskonzept für eine Klasse komplexer, heterogener Systeme. Diese Klasse umfaßt die Systeme, deren funktionales Verhalten sich durch bereitgestellte Dienste beschreiben läßt. Diese Dienste werden von Klienten in Form von Anfragen und Antworten genutzt. Diese Nutzung ist abstrakt durch eine Last quantifizierbar.

Auf Basis praktischer industrieller Erfahrungen wird die Hypothese aufgestellt, daß es ein zielführender Ansatz zur Validierung ist, wenn eine zeitlich veränderliche Last durch eine veränderliche Menge von Klienten an den Diensten des zu testenden Systems erzeugt wird.

Auf den Klienten werden zur Ausführung s.g. Testklienten implementiert, die in der Lage sind, die Dienste des zu testenden Systems mit bestimmten Lasten zu nutzen. Die funktional korrekte - also erfolgreiche - Nutzung wird explizit von der implementierten Logik des Testklienten ermittelt und ggf. um relevante Güteparameter ergänzt. Die Umsetzung der Testklienten kann als Applikation oder Skript erfolgen und wird in erster Linie durch die Technologie des Klienten bestimmt.

Die Annahme einer heterogenen Klientenarchitektur ist elementarer Bestandteil der Anforderungen. Klienten können sich in ihren eigenen technischen Möglichkeiten, in Ressourcen oder in Netzwerkkapazitäten stark unterscheiden. Auch eine beständige Verfügbarkeit zur Nutzung von Diensten während der Validierungslaufzeit kann nicht vorausgesetzt werden.

Die Gesamtlast für die Dienste des zu testenden Systems ergibt sich aus den Einzellasten der Klienten. Entsprechend müssen die verfügbaren Testklienten - unter Umständen eine große Zahl davon - in der Arbeit von einem zentralen System koordiniert werden. Hierfür muß das zentrale System die Eigenschaften der Klienten kennen. Ein entsprechendes Modell von Kostenfunktionen, Grenzlasten und Verfügbarkeiten ist folglich neben den Testszenarien zu erstellen.

Eine Automatisierung der Ausführung ist in Anbetracht der Vielzahl von Klienten und deren möglicher Dynamik unumgänglich. Das zentrale System erhält dazu Instanzen der erstellten Modelle sowie der Testszenarien als Eingabe und verteilt die zu erzeugenden Lasten auf die verfügbaren Klienten. Bei Änderungen zur Laufzeit werden entsprechende Adaptionen vorgenommen. Desweiteren empfängt das zentrale System die Berichte der Klienten über Erfolg und Güteparameter einzelner Dienstnutzungsaktionen. Diese werden ausgewertet und gespeichert.

Das Konzept sieht auch eine automatisierte Auswertung sowie eine Aufbereitung für die Tester vor. Im Rahmen dieser Arbeit wurde diese Funktion jedoch nicht komplett umgesetzt sondern nur Informationen dargestellt, die für die Evaluierung notwendig sind.

8.2. Testpartitionierung und Testbench

Die automatisierte Ausführung der Validation bedingt die Zerlegung der im Szenario geforderten Lasten eines jeden Dienstes in eine Menge kleinerer Lasten sowie die Zuordnung dieser zu den verfügbaren Klienten. Zerlegung und Zuordnung sind nicht unabhängig voneinander, da auch die Zuordnung die Größe der Einzellasten beeinflußt. Die algorithmische Lösung dieses Testpartitionierungsproblems ist folglich entscheidend für die Funktionsfähigkeit des Konzeptes.

Durch die gegebenen Modelle handelt es sich beim Testpartitionierungsproblem um ein Optimierungsproblem, dessen Zielfunktion nichtlinear und unstetig ist. Wegen der Interpretierbarkeit als Last und der maschinengerechten Verarbeitung sind die Optimierungsvariablen desweiteren diskret definiert, so daß auch die Testpartitionierung als kombinatorisches Optimierungsproblem gesehen werden kann.

Auf Basis bestehender, heuristischer Ansätze wurde ein iterativer Optimierungsalgorithmus erarbeitet und vorgestellt. Die Grundidee besteht in der Bestimmung und Verteilung einer durchschnittlichen Laständerung auf allen verfügbaren Klienten. Durch paarweises Schieben dieser Änderung wird versucht, die durch Kostenfunktionen beschriebenen Kosten zu optimieren. Da es aufgrund von Einschränkungen bei den Testklienten zu Abweichungen zwischen der zugeordneten und zuzuordnenden Last kommen kann, wird in diesem Falle der Algorithmus rekursiv erneut gerufen.

Der Algorithmus hat eine polynomiale Laufzeit, die von der Anzahl der Klienten und der Anzahl der Dienste abhängt. Da es für die Anzahl der Iterationen und Rekursionen Grenzwerte gibt, kann die Laufzeit auch begrenzt werden - zumeist natürlich auf Kosten der Qualität der Ergebnisse.

Parallel zum Problem der Testpartitionierung müssen für eine Automatisierung auch technische Strukturen geschaffen werden. Hierfür wurde eine verteilte Testbench konzipiert und umgesetzt. Diese besteht aus drei Komponenten:

- Testautomatisierungsmodul
  Errechnet mit dem Testpartitionierungsalgorithmus die Aufträge für die verfügbaren Klienten und parametrisiert diese.
- Testservermodul
  Verwaltet Verbindungen und Zustände der Testklienten, leitet Testaufträge und -ergebnisse weiter.
- Testklienten
  Empfangen Testaufträge vom Testservermodul und führen diese aus.

Die Modularisierung dient der leichteren Wiederverwendbarkeit der Komponenten für andersartige, zu testende Systeme. Die Testklienten sind üblicherweise an das System und die zu testenden Dienste anzupassen, wohingegen das Automatisierungsmodul auf abstrakten Modellen arbeitet, die kaum Anpassungen benötigen. Zwischen allen Komponenten muß Kommunikation über geeignete Netze möglich sein.

Für das Anwendungsbeispiel eines zu validierenden Mobilfunknetzwerkes ist die Testbench, samt Testpartitionierungsalgorithmus, implementiert worden. Das Testserver- und das Automatisierungsmodul kommunizieren über TCP/IP-Netzwerke. Die Testklienten sind sowohl auf Desktopsystemen wie auch auf Mobiltelefonen lauffähig und angepaßt, so daß auch in diesem Beispiel die heterogene Klientenarchitektur zu finden ist. Die Kommunikation zwischen Testklienten und Testservermodul wird über die Datenverbindung des eigentlich zu testenden Mobilfunknetzes bewerkstelligt.

8.3. Auswertung

Der Testpartitionierungsalgorithmus und die Testbench wurden hinsichtlich der Funktionsfähigkeit und einiger Kernparameter evaluiert.

Zur besseren Einschätzung der Ergebnisse wurde der heuristische Algorithmus mit dem Matlab-Solver fmincon, der für nichtlineare Optimierungsprobleme gedacht ist, verglichen. In nahezu allen Testreihen lieferte die Heuristik bessere Ergebnisse in erheblich kürzerer Zeit und unter deutlich geringerem Ressourcenbedarf. Ersteres ist vor allem darauf zurückzuführen, daß fmincon Probleme mit der möglichen Unstetigkeit der Zielfunktion hat.

Messungen wurden bis hin zu mehreren tausend Klienten und damit auch mehreren tausend Optimierungsvariablen durchgeführt. Während *fmincon* bei rund 1000 Klienten wegen mangelnden Arbeitsspeichers abbricht, konnte mittels der Heuristik stets eine Lösung gefunden werden, die die Nebenbedingungen erfüllen. Allerdings wachsen die benötigten Rechenzeiten (ohne relevante Begrenzung der Iterations- und Rekursionsanzahl) massiv in den zweistelligen Minutenbereich. Mit der existierenden Implementierung und dem handelsüblichen Desktoprechner zur Ausführung ist die Behandlung von rund 500 Klienten bei Berechnungszeiten von unter 10 Sekunden realistisch.

Auch die Testbench wurde in zwei Ausbaustufen umgesetzt. In Stufe 1 wurden ausschließlich das Servermodul und die Testklienten für verschiedene Architekturen realisiert. Für kurze Zeit Bestand die Möglichkeit, dieses in einem realen Mobilfunknetz zu testen, wobei Testaufträge an Klienten manuell erstellt wurden.

In Ausbaustufe 2 wurde das Automatisierungsmodul unter Verwendung der implementierten Heuristik ergänzt. Zur Evaluierung dieser mußte auf eine Simulation ausgewichen werden. Die Klienten und das Mobilfunknetzwerk wurden im Netzwerksimulator SimANet nachgebildet. Anhand verschiedenartiger Dienste wurde überprüft, wie genau die Ist-Last aller Klienten der Soll-Last der Szenarien folgt. Im ungünstigsten, betrachteten Fall betrugen die Abweichungen, gemessen über die Laufzeit des Szenarios, ca. 20%. In den übrigen Messungen lagen die Abweichungen deutlich unter 10% und damit in einem überzeugenden Rahmen.

Auch die Auswirkungen dynamischer Änderungen während der Validierungsausführung wurden überprüft. Es wurde deutlich, daß die Dauer der Abweichungen, die durch den Ausfall von Klienten entsteht, in erster Linie von der Detektionsdauer der Störung abhängt. Dieser Parameter ist vorrangig technologie- bzw. architekturabhängig.

8.4. Erfahrungen

Insbesondere im Laufe der Evaluierungen konnten erste Erfahrungen gesammelt werden, die zum Teil eine Bewertung der Hypothese 1 zulassen (siehe Abschnitt 4.1).

Bei Betrachtung des Gesamtkonzeptes fällt auf, daß die Validierung durch eine Art von Leistungstests ausgeführt wird. Genaugenommen kann das auch direkt aus der Hypothese, die in Abschnitt 4.1 eingeführt wurde, gefolgert werden. Und es ist anzunehmen, daß für die Klasse der SPS diese Form der Validierung auch überaus praxisrelevant ist. Rein funktionales Testen kann unter Laborbedingungen durchgeführt werden, wohingegen sich Effekte durch Parallelisierung, Verzögerungen oder in Folge von Ressourcenkonflikten erst im Falle starker und paralleler Belastung zeigen. Genau diesen Fall deckt das Konzept ab.

Bereit im Laufe der Entwicklung des Testservermoduls, bei dem es sich definitionsgemäß auch um ein SPS handelt, zeigte sich die Funktionsfähigkeit dieses Ansatzes. Hier wurden rein funktionale Tests unter entsprechend kontrollierten Laborbedingungen durchgeführt. Erst mit der Kopplung an den Simulator bestand die Möglichkeit, parallel zahlreiche Klientenverbindungen zum Testservermodul aufzubauen sowie Dynamik in diese Struktur zu bringen. Allein dadurch gelang es bereits, zahlreiche weitere Fehler aufzufinden, von denen viele auf falsch synchronisierte Datenstrukturzugriffe zurückzuführen waren.

Kritisch ist jedoch die Frage zu bewerten, wo genau die Grenzen des zu testenden Systems liegen. Im Falle des Mobilfunkbeispiels werden Telefone zum Zugriff auf die Dienste verwendet. Der Testklient nutzt dabei Systembibliotheken und diese wiederum die Hardware des jeweiligen Telefons. Entsprechend sind bereits diese Komponenten Bestandteil des ausgeführten Tests. Mit dieser Tatsache muß bewußt umgegangen werden. Einerseits kann die Integration der Kliententechnologie in die Validierung durchaus wünschenswert sein, da dadurch auch diese Systeme einem regelmäßigen Test unterzogen werden. Sollte es jedoch nicht gewünscht sein, muß darauf geachtet werden, daß die Kliententechnologie möglichst hochwertig bzw. ausgereift ist, um nicht falsche Schlüsse auf das Verhalten des eigentlichen SPS zu implizieren.

Im allgemeinen ist zu beachten, daß der Aufwand zur Erstellung einer lauffähigen Testbench verhältnismäßig hoch ist. Auch wenn das Testautomatisierungs- und das Testservermodul weitestgehend unverändert übernommen werden können, ist die Adaption der Testklienten an andere SPS unumgänglich. Die Heterogenität der Klienten kann zwangsläufig dazu führen, daß verschiedene Implementierungen notwendig sind. Entsprechend muß abgewogen werden, inwieweit der Nutzen den Aufwand rechtfertigt. Sinnvoll ist vor allem der Einsatz in Systemen mit zyklischen Weiterentwicklungen, da hier die Änderungen überschaubar sind.

8.5. Ausblick

Das Validierungskonzept wurde im Rahmen dieser Arbeit vorgestellt, partiell prototypisch implementiert und durch die Überprüfung von Kernparametern evaluiert. Neben der Implementierung weiterer Anforderungen der Modelle und der Praxis sollten insbesondere die Evaluationen fortgesetzt werden. Hierzu muß der Wechsel zu realen zu testenden Systemen vorgenommen werden. Im Falle des simulierten Mobilfunkbeispiels traten bei der Verwendung mehrerer Dutzend Klienten bereits zahlreiche Seiteneffekte auf, die durch die hohe Auslastung des Simulationssystems verursacht wurden.

Diese Schritte dienen der Finalisierung des vorgestellten Konzeptes. Aber auch einige Weiterentwicklungen sind reizvoll. Insbesondere das bisher nicht weiter betrachtete Auswertungsmodul bietet einige Möglichkeiten, die Validierung zu verbessern bzw. das Konzept auch stärker in Richtung Systemtest zu verschieben. Wünschenswert wäre die Beeinflußung des laufenden Testvorganges durch die eintreffenden Ergebnisse. Können anhand bestimmter Muster Vermutungen auf Ursachen gefolgert werden, wäre es denkbar, das Kostenmodell des Automatisierungsmoduls derart anzupassen, daß die vermutete Ursache stärker im Gesamtsystem provoziert wird. Durch diesen dann schon fast automatischen (nicht automatisierten) Test könnte die Diagnose möglicher Ursachen ohne großen manuellen Einsatz vereinfacht werden.

Werden die Klienten als Teil des zu testenden Systems verstanden, ist auch eine Dienstnutzung zwischen Klienten - ohne Einbeziehung des zentralen Systems - denkbar. Da diese Klient-Klient-Relationen auf das derzeitige zentralisierte Modell nicht abgebildet werden können, sind entsprechende Änderungen notwendig.

Obwohl das Konzept und das Beispiel als automatisierte Validierungssysteme eingeführt und konzipiert wurden, sind weitere Anwendungsfelder, ggf. mit leichten Anpassungen, denkbar. Beispielsweise kann das Validierungskonzept am Mobilfunkbeispiel an bestehende Ansätze zur Ermittlung der Key Performance Parameter gekoppelt werden (siehe Abschnitt 3.4). Damit wäre die Definition dezentraler, kundennäherer KPI denkbar. Der Prozeß der Ermittlung könnte verstetigt werden, indem für eine Menge an Kunden Anreize zur parallelen Nutzung der Mobilfunkgeräte als Testklienten geschaffen werden.

Denkbar wäre auch eine Anwendung außerhalb der Validierung bzw. des Systemtests. Wegen der ähnlichen Architektur und weitestgehend identischer Modelle ist eine Nutzung der hier vorgestellten Bench, insbesondere des Partitionierungsalgorithmus, zum Lastausgleich in verteilten Rechensystemen denkbar. Allerdings widerstrebt der zentralisierte Ansatz dieser Arbeit den derzeitigen Tendenzen der verteilten Rechendienste - gemeinhin als Cloud Computing bezeichnet.

Durch die Abstrahierung der Modelle von Dienst und Nutzung ist auch eine applikative Anwendung der Grundkonstrukte nötig. Derzeit gibt es bspw. zahlreiche Ansätze im Bereich der klinischen Psychologie, Mobiltelefone zur Überwachung, Konditionierung und Ablenkung gefährdeter Patientengruppen zu nutzen. Hauptgrund sind die gestiegenen technischen Möglichkeiten der Telefone sowie die Verfügbarkeit benötigter Datennetze.

A. Nichtlineare Kostenfunktionen

Die Überprüfung der Heuristik auf Basis nichtlinearer und unstetiger Kostenfunktionen in Kapitel 5.3.3.2 erfolgt mit den hier angegebenen Funktionen A.1 bis A.11.

Die jeweilige Funktion $c_{i,j}(x_1, ..., x_n)$ definiert die Kosten für die Nutzung von Dienst j durch den Klienten i bei den jeweiligen absoluten Lasten x aller n Dienste. Die Kosten für Dienst 3 sind auf allen Klienten identisch.

Ergänzend sind in den Gleichungen A.12 bis A.17 die maximalen Absolutlasten angegeben, die die jeweiligen Klienten für den jeweiligen Dienst erzeugen können.

Für die Evaluierung werden die Klienten entsprechend ihrer laufenden Nummer per Modulo-Operation auf die fünf definierten Klientengruppen $c_{1,j}$ bis $c_{5,j}$ abgebildet.

$$c_{1,1}(x_1) = \begin{cases} 0 & , x_1 = 0 \\ x_1^4 & , x_1 > 0 \end{cases} \tag{A.1}$$

$$c_{2,1}(x_1) = \begin{cases} 0 & , x_1 = 0 \\ (0.5 \cdot x_1)^2 + 10 & , x_1 > 0 \end{cases} \tag{A.2}$$

$$c_{3,1}(x_1, x_2) = \begin{cases} 0 & , x_1 = 0 \\ 0 & , x_2 < 20 \land 20 < x_1 < 60 \\ x_1 + 100 & , x_2 < 20 \land (x_1 \leq 20 \lor x_1 \geq 60) \\ x_1^2 + 100 & , x_2 \geq 20 \end{cases} \tag{A.3}$$

$$c_{4,1}(x_1) = \begin{cases} 0 & , x_1 = 0 \\ 50 \cdot (x_1 mod 10)^2 & , (x_1 \mod 10) < 3 \\ x_1^2 + 50 & , (x_1 \mod 10) \geq 3 \end{cases} \tag{A.4}$$

$$c_{5,1}(x_1) = \begin{cases} 0 & , x_1 = 0 \\ 400 & , x_1 < 20 \\ 1000 & , 20 \leq x_1 < 40 \\ 500 & , x_1 \geq 40 \end{cases} \tag{A.5}$$

$$c_{1,2}(x_2) = \begin{cases} 0 & , x_2 = 0 \\ x_2^4 & , x_2 > 0 \end{cases} \tag{A.6}$$

$$c_{2,2}(x_2) = \begin{cases} 0 & , x_2 = 0 \\ (x_2 - 20)^2 + 50 & , 0 < x_2 < 50 \\ 900 & , x_2 \geq 50 \end{cases} \tag{A.7}$$

$$c_{3,2}(x_1, x_2) = \begin{cases} 0 & , x_2 = 0 \\ 50 & , x_1 < 200 \wedge 50 < x_2 < 60 \\ 5000 & , x_1 < 200 \wedge (x_2 \leq 50 \vee x_2 \geq 60) \\ 10000 & , x_1 \geq 20 \end{cases} \tag{A.8}$$

$$c_{4,2}(x_1, x_2) = \begin{cases} 0 & , x_2 = 0 \\ x_1 \cdot x_2 & , x_2 > 0 \end{cases} \tag{A.9}$$

$$c_{5,2}(x_2) = \begin{cases} 0 & , x_2 = 0 \\ 400 & , x_2 < 20 \\ 1000 & , 20 \leq x_2 < 40 \\ 500 & , x_2 \geq 40 \end{cases} \tag{A.10}$$

$$c_{i,3}(x_1, x_2, x_3) = \begin{cases} 0 & , x_3 = 0 \\ (x_1 + x_2) \cdot x_3 & , x_3 \geq 0 \end{cases} \tag{A.11}$$

$$a_{1,1} = a_{1,2} = 100 \tag{A.12}$$

$$a_{2,1} = a_{2,2} = 50 \tag{A.13}$$

$$a_{3,1} = a_{3,2} = 20 \tag{A.14}$$

$$a_{4,1} = a_{4,2} = 50 \tag{A.15}$$

$$a_{5,1} = a_{5,2} = 50 \tag{A.16}$$

$$a_{i,3} = 1 \tag{A.17}$$

B. Wertediskretisierung

Für den Matlab-Solver fmincon, der zur Evaluierung im Abschnitt 5.3 verwendet wird, sind die Optimierungsvariablen vom Typ $double$ und werden intern als Fließkommazahlen mit 64-Bit Auflösung behandelt. Entsprechend kann die gefundene Optimierungslösung (entspricht Laständerungen) auch aus nicht ganzzahligen Werten bestehen.

Dies steht im Widerspruch zur Definition der Last, die in Gleichung 4.6 angegeben ist. Aus Gründen der Interpretierbarkeit und auch der einfachen Verarbeitung in Eingebetteten Systemen (Klienten) wird die Last als natürliche Zahl definiert.

Zur Vergleichbarkeit der Ergebnisse von Matlab und der Heuristik ist deshalb die Diskretisierung der Ergebnisse von fmincon notwendig.

Denkbar sind verschiedene Ansätze. Der naheliegendste ist, mittels zusätzlichen Nebenbedingungen die Optimierung per fmincon derart einzuschränken, daß nur ganzzahlige Werte der Variablen zu gültigen Lösungen führen. Durchläufe dieser Art führten jedoch dazu, daß fmincon in den meisten Fällen nicht in der Lage war, eine gültige Lösung zu finden und die Optimierung nach Erreichen von Iterationsgrenzen abbrach.

Das nachträgliche Korrigieren der Ergebnisse erwies sich daher als erfolgsversprechender. Hierbei müssen zwei Kriterien berücksichtigt werden. Zum einen gilt es, die Rundungsabweichungen einzelner Variablen über sämtliche Variablen auszugleichen, so daß in Summe die tatsächliche Soll-Laständerung, die von Variablen repräsentiert wird, auch erreicht wird. Zum zweiten gilt es, die Rundungen so vorzunehmen, daß die Auswirkungen auf die entstehenden Kosten möglichst gering sind. Es gilt zu beachten, daß durch Unstetigkeiten in den Kostenfunktionen bereits minimale Änderungen durch Rundung zu großen Änderungen in den Kosten führen können.

Für das Evaluierungssystem, welches in Abschnitt 5.3.2 vorgestellt wird, findet im Block „Auswertung“ mittels des unten angegebenen Algorithmus 4 die Diskretisierung der Ergebnisse von fmincon statt. Eingabe für den Algorithmus bildet neben den berechneten Laständerungen der einzelnen Klienten auch die Soll-Last eines jeden Dienstes ($targetOffset$) sowie die nach der mathematischen Rundung aller Einzelwerte erreichte Ist-Last ($actualOffset$).

Algorithmus 4 Wertediskretisierung

```
procedure LOADDISCRETISATION($targetOffset[n], actualOffset[n]$)
    for all Dienste $j$ do
        $diff[j] \leftarrow round(targetOffset[j] - actualOffset[j])$
        while $diff[j] \geq 1$ do
            $min \leftarrow$ Integer.MAX_VALUE
            $client \leftarrow$ NULL
            for all Klienten $i$ do
                if ${}^{t}l'_{i,j} + {}^{t}l^{+}_{i,j} + 1 \leq a_{i,j}$ then
                    if $c_{i,j}(t, (..., {}^{t}l'_{i,j} + {}^{t}l^{+}_{i,j} + 1, ...)) - c_{i,j}(t, (..., {}^{t}l'_{i,j} + {}^{t}l^{+}_{i,j}, ...)) < min$ then
                        $min \leftarrow c_{i,j}(t, (..., {}^{t}l'_{i,j} + {}^{t}l^{+}_{i,j} + 1, ...))$
                        $client \leftarrow i$
                    end if
                end if
            end for
            if $client <> $ NULL then
                ${}^{t}l^{+}_{i,j} \leftarrow {}^{t}l^{+}_{i,j} + 1$
                $diff[j] \leftarrow diff[j] - 1$
            else
                $break$
            end if
        end while
        while $diff[j] \leq -1$ do
            $min \leftarrow$ Integer.MAX_VALUE
            $client \leftarrow$ NULL
            for all Klienten $i$ do
                if ${}^{t}l'_{i,j} + {}^{t}l^{+}_{i,j} \geq 1$ then
                    if $c_{i,j}(t, (..., {}^{t}l'_{i,j} + {}^{t}l^{+}_{i,j} - 1, ...)) - c_{i,j}(t, (..., {}^{t}l'_{i,j} + {}^{t}l^{+}_{i,j}, ...)) < min$ then
                        $min \leftarrow c_{i,j}(t, (..., {}^{t}l'_{i,j} + {}^{t}l^{+}_{i,j} - 1, ...))$
                        $client \leftarrow i$
                    end if
                end if
            end for
            if $client <> $ NULL then
                ${}^{t}l^{+}_{i,j} \leftarrow {}^{t}l^{+}_{i,j} - 1$
                $diff[j] \leftarrow diff[j] + 1$
            else
                $break$
            end if
        end while
    end for
end procedure
```

Literaturverzeichnis

[1] IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE Std 610.12-1990*

[2] *Binary XML Content Format Specification*. Version 1.3. Wireless Application Protocol Forum, 2001 `http://www.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-192-wbxml-20010725-a.pdf`

[3] BUNDESNETZAGENTUR FÜR ELEKTRIZITÄT, GAS, TELEKOMMUNIKATION, POST UND EISENBAHNEN: Jahresbericht 2010. Version: 2010. `http://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/BNetzA/Presse/Berichte/2011/Jahresbericht2010pdf.pdf?__blob=publicationFile`. 2010. – Forschungsbericht

[4] THE MATHWORKS, INC.: Optimization Toolbox User's Guide. Version: 2012. `http://www.mathworks.com/help/pdf_doc/optim/optim_tb.pdf`. 2012 (R2012b). – Forschungsbericht

[5] ANDREICA, M. ; TAPUS, N. ; IOSUP, Alexandru ; EPEMA, Dick ; DUMITRESCU, Catalin ; RAICU, Ioan ; FOSTER, Ian ; RIPEANU, M.: Towards ServMark, an Architecture for Testing Grids / Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence. Version: November 2006. `http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0062.pdf`. 2006 (TR-0062). – Forschungsbericht

[6] ANDREICA, Mugurel I. ; TAPUS, Nicolae ; DUMITRESCU, Catalin ; IOSUP, Alexandru ; EPEMA, Dick ; RAICU, Ioan ; FOSTER, Ian ; RIPEANU, Matei: Towards ServMark, an Architecture for Testing Grid Services. In: *Technical University of Delft - Technical Report* (2006), Juli

[7] ANDREWS, Dorothy M. ; BENSON, Jeoffrey P.: An automated program testing methodology and its implementation. In: *Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA : IEEE Press, 1981 (ICSE '81). – ISBN 0–89791–146–6, S. 254–261

[8] ANGERMANN, A.: *MATLAB- Simulink- Stateflow: Grundlagen, Toolboxen, Beispiele*. Oldenbourg, 2007. – ISBN 9783486582727

[9] AVRITZER, Alberto ; WEYUKER, Elaine J.: Generating test suites for software load testing. In: *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA : ACM, 1994 (ISSTA '94). – ISBN 0–89791–683–2, S. 44–57

[10] BADKE-SCHAUB, P. ; HOFINGER, G. ; LAUCHE, K.: *Human Factors: Psychologie sicheren Handelns in Risikobranchen.* Springer, 2011. – ISBN 9783642198854

[11] BANET, F.J. ; GÄRTNER, A. ; TESSMAR, G.: *UMTS.* Hüthig, 2004 (Hüthig Telekommunikation). – ISBN 9783826650345

[12] BARCO, Raquel ; WILLE, Volker ; DÍEZ, Luis: System for automated diagnosis in cellular networks based on performance indicators. In: *European Transactions on Telecommunications* 16 (2005), Nr. 5, S. 399–409. – ISSN 1541–8251

[13] BARTHOLOMEW-BIGGS, M.C.: *Nonlinear Optimization with Engineering Applications.* Springer, 2008 (Springer Optimization and Its Applications). – ISBN 9780387787220

[14] BEIZER, Boris: *Black-box testing - techniques for functional testing of software and systems.* Wiley, 1995. – I–XXV, 1–294 S. – ISBN 978–0–471–12094–0

[15] BENSON, Hande Y. ; SEN, Arun ; SHANNO, David F. ; VANDERBEI, Robert J.: Interior-Point Algorithms, Penalty Methods and Equilibrium Problems. In: *Comput. Optim. Appl.* 34 (2006), Juni, Nr. 2, S. 155–182. – ISSN 0926–6003

[16] BOEHM, B. W.: Guidelines for Verifying and Validating Software Requirements and Design Specifications. In: SAMET, P. A. (Hrsg.): *Euro IFIP 79*, North Holland, 1979, S. 711–719

[17] BROUWER, F. ; BRUIN, I. de ; SILVA, J.C. ; SOUTO, N. ; CERCAS, F. ; CORREIA, A.: Usage of link-level performance indicators for HSDPA network-level simulations in E-UMTS. In: *Spread Spectrum Techniques and Applications, 2004 IEEE Eighth International Symposium on*, 2004, S. 844 – 848

[18] BYRD, Richard H. ; HRIBAR, Mary E. ; NOCEDAL, Jorge: An Interior Point Algorithm for Large Scale Nonlinear Programming. In: *SIAM Journal on Optimization* 9 (1997), S. 877–900

[19] CASPAR, Mirko ; VODEL, Matthias ; HARDT, Wolfram: Automated Test Distribution Framework for Service Providing Systems. In: *Proceedings of the 4th National Conference on Computing and Information Technology (NCCIT2008)).* Mahasarakham, Thailand : King Mongkut's University of Technology North Bangkok (KMUTNB), May 2008. – ISBN 978–974–19–3296–2, S. 58–62

[20] CASPAR, Mirko ; VODEL, Matthias ; HARDT, Wolfram: System Level Test of Service-based Systems by Automated and Dynamic Load Partitioning and Distribution. In: *Proceedings of the 10th International Conference on Innovative Internet Community Systems (I2CS2010).* Bangkok, Thailand : ACM Press, June 2010. – ISBN 978–3–88579–259–8

[21] CERVIN, Anton ; HENRIKSSON, Dan ; OHLIN, Martin: *TrueTime 2.0 beta - Reference Manual.* Department of Automatic Control, Lund University, Sweden, 2010 `http://www3.control.lth.se/truetime/report-2.0-beta5.pdf`

[22] CHING, W.K. ; NG, M.K.: *Markov Chains: Models, Algorithms and Applications*. Springer, 2010 (International Series in Operations Research & Management Science). – ISBN 9781441939869

[23] CLAUS, V. ; SCHWILL, A.: *Duden - Informatik A - Z: Fachlexikon für Studium, Ausbildung und Beruf*. Dudenverlag, 2006. – ISBN 9783411052349

[24] DALIBOR, S.: *Erstellung von Testplänen für verteilte Systeme durch stochastische Modellierung*. Inst. für Informatik, Univ. Erlangen-Nürnberg, 2001 (Arbeitsberichte des Instituts für Informatik)

[25] DESPREZ, F. ; GETOV, V. ; PRIOL, T. ; YAHYAPOUR, R.: *Grids, P2P and Services Computing*. Springer, 2010. – ISBN 9781441967930

[26] DONY, C. ; KNUDSEN, J.L. ; ROMANOVSKY, A. ; TRIPATHI, A.: *Advanced Topics in Exception Handling Techniques*. Springer, 2006 (Lecture Notes in Computer Science). – ISBN 9783540374435

[27] DUMITRESCU, Catalin ; RAICU, Ioan ; RIPEANU, Matei ; FOSTER, Ian: DiPerF: An Automated DIstributed PERformance Testing Framework. In: *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA : IEEE Computer Society, 2004 (GRID '04). – ISBN 0–7695–2256–4, S. 289–296

[28] DUSTDAR, Schahram ; HASLINGER, Stephan: Testing of Service-Oriented Architectures - A Practical Approach. In: *Net.ObjectDays*, 2004, S. 97–109

[29] DUSTIN, E. ; RASHKA, J. ; PAUL, J.: *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, 1999. – ISBN 9780201432879

[30] ELSTER, K.H.: *Nichtlineare Optimierung*. H. Deutsch, 1978 (Mathematik für Ingenieure). – ISBN 9783871444319

[31] ERL, Thomas: *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2005. – ISBN 0131858580

[32] FEWSTER, Mark ; GRAHAM, Dorothy: *Software test automation: effective use of test execution tools*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1999. – ISBN 0–201–33140–3

[33] FISCHER, P. ; HOFER, P.: *Lexikon der Informatik*. Springer, 2007. – ISBN 9783540725497

[34] FRANZ, K.: *Handbuch zum Testen von Web-Applikationen: Testverfahren, Werkzeuge, Praxistipps*. Springer, 2007 (Xpert. press Series). – ISBN 9783540245391

[35] FRÜHAUF, K. ; LUDEWIG, J. ; SANDMAYR, H.: *Software-Prüfung: Eine Anleitung zum Test und zur Inspektion*. Vdf Hochschulverlag AG, 2007 (vdf Lehrbuch). – ISBN 9783728130594

[36] GAISBAUER, Sebastian ; KIRSCHNICK, Johannes ; EDWARDS, Nigel ; ROLIA, Jerry: VATS: Virtualized-Aware Automated Test Service. In: *QEST*, 2008, S. 93–102

[37] GARCÍA, Ana B. ; ALVAREZ-CAMPANA, Manuel ; VÉZQUEZ, Enique ; GUENON, Guillermo ; BERROCAL, Julio: ATM transport between UMTS base stations and controllers: supporting topology and dimensioning decisions. In: *PIMRC*, IEEE, 2004, S. 2176–2180

[38] GAREY, Michael R. ; JOHNSON, David S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman & Co., 1979. – ISBN 0716710447

[39] GIRLICH, E. ; KOVALEV, M.M.: *Nichtlineare diskrete Optimierung*. Akademie-Verlag, 1981 (Mathematical research)

[40] GRECHENIG, T. ; BERNHART, M.: *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Studium, 2009 (Pearson Studium). – ISBN 9783868940077

[41] GROSSO, W.: *Java RMI*. O'Reilly Media, 2001 (Java Series). – ISBN 9781565924529

[42] HADDAD, E.: Load Distribution Optimization in Heterogeneous Multiple Processor Systems. In: *Heterogeneous Processing, 1993. WHP 93. Proceedings. Workshop on*, 1993. – ISSN 1066–1220, S. 42 –47

[43] HAMIDZADEH, Babak ; LILJA, David J. ; ATIF, Yacine: Dynamic Scheduling Techniques for Heterogeneous Computing Systems. In: *CONCURRENCY: PRACTICE AND EXPERIENCE* 7 (1995), S. 633–652

[44] HARDT, Wolfram: *HW-SW-Codesign auf Basis von C-Programmen unter Performanz-Gesichtspunkten*. Shaker, 1996 (Berichte aus der Informatik). – ISBN 9783826520518

[45] HARDT, Wolfram: *Integration von Verzögerungszeit-Invarianz in den Entwurf eingebetteter Systeme*. Shaker, 2001 (C-LAB publication). – ISBN 9783826582516

[46] HEINKEL, U. ; INGENIEURE, Verein D.: *Formale Spezifikation und Validierung digitaler Schaltungsbeschreibungen mit Zeitdiagrammen*. VDI-Verlag, 1999 (Fortschritt-Berichte VDI). – ISBN 9783183290208

[47] HOCHBAUM, Dorit S. (Hrsg.): *Approximation algorithms for NP-hard problems*. Boston, MA, USA : PWS Publishing Co., 1997. – ISBN 0–534–94968–1

[48] HUNT, A. ; THOMAS, D.: *Unit-Tests mit JUnit*. Hanser, 2004 (Pragmatisch Programmieren). – ISBN 9783446404694

[49] ISSARIYAKUL, T. ; HOSSAIN, E.: *Introduction to Network Simulator NS2*. Springer, 2008. – ISBN 9780387717593

[50] KELLERER, H. ; PFERSCHY, U. ; PISINGER, D.: *Knapsack problems*. Springer, 2004. – ISBN 9783540402862

[51] KLIMKE, Andreas: How to access Matlab from Java, IANS report 2003/005 / University of Stuttgart. Version: 2003. `http://preprints.ians.uni-stuttgart.de`. 2003. – Forschungsbericht

[52] KORTE, B. ; VYGEN, J.: *Kombinatorische Optimierung: Theorie und Algorithmen*. Springer Berlin Heidelberg, 2008. – ISBN 978–3–540–76919–4

[53] KRISHNAMURTHY, Diwakar ; ROLIA, Jerome A. ; MAJUMDAR, Shikharesh: A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems. In: *IEEE Trans. Softw. Eng.* 32 (2006), November, S. 868–882. – ISSN 0098–5589

[54] KRUSE, R. ; BORGELT, C. ; KLAWONN, F. ; MOEWES, C. ; RUSS, G. ; STEINBRECHER, M.: *Computational Intelligence: Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*. Vieweg+Teubner Verlag, 2011 (Computational Intelligence). – ISBN 9783834812759

[55] LAIHO, J. ; RAIVIO, K. ; LEHTIMAKI, P. ; HATONEN, K. ; SIMULA, O.: Advanced analysis methods for 3G cellular networks. In: *Wireless Communications, IEEE Transactions on* 4 (2005), may, Nr. 3, S. 930 – 942. – ISSN 1536–1276

[56] LE LANN, G.: An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. In: *Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on*, 1997, S. 339 –346

[57] LEHNER, Franz: *Mobile und drahtlose Informationssysteme*. Berlin [u.a.] : Springer, 2003. – ISBN 3540439811

[58] LI, Ying ; LI, Minglu ; YU, Jiadi: Web Services Testing, the Methodology, and the Implementation of the Automation-Testing Tool. In: LI, Minglu (Hrsg.) ; SUN, Xian-He (Hrsg.) ; DENG, Qian-ni (Hrsg.) ; NI, Jun (Hrsg.): *Grid and Cooperative Computing* Bd. 3032. Springer Berlin / Heidelberg, 2004. – ISBN 978–3–540–21988–0, S. 940–947

[59] LUDWIG, Thomas: *Automatische Lastverwaltung für Parallelrechner*. BI-Wiss.-Verl., 1993. – ISBN 978–3411165018

[60] MACKENZIE, Matthew ; LASKEY, Ken ; MCCABE, Francis ; BROWN, Peter F. ; METZ, Rebekah ; HAMILTON, Booz A.: Reference Model for Service Oriented Architecture 1.0. In: *Architecture* 12 (2006), Nr. July, S. 1–31

[61] MEI, Lijun ; ZHANG, Zhenyu ; CHAN, W. K. ; TSE, T. H.: Test case prioritization for regression testing of service-oriented business applications. In: *Proceedings of the 18th international conference on World wide web*. New York, NY, USA : ACM, 2009 (WWW '09). – ISBN 978–1–60558–487–4, S. 901–910

[62] MYERS, Glenford J. ; SANDLER, Corey: *The Art of Software Testing*. John Wiley & Sons, 2004. – ISBN 0471469122

[63] NAHIR, Amir ; SHILOACH, Yossi ; ZIV, Avi: Using linear programming techniques for scheduling-based random test-case generation. In: *Proceedings of the 2nd international Haifa verification conference on Hardware and software, verification and testing*. Berlin, Heidelberg : Springer-Verlag, 2007 (HVC'06). – ISBN 978–3–540–70888–9, S. 16–33

[64] NAUR, Peter (Hrsg.) ; RANDELL, Brian (Hrsg.): *Proceedings, NATO Conference on Software Engineering*. Garmisch, Germany, 1968

[65] ORSTAD, Bengt M. ; REIZER, Erling: *End-to-end Key Performance Indicators in Cellular Networks*, Agder University College, Grimstad, Norway, Diplomarbeit, May 2006

[66] OSMAN, A. ; AMMAR, H.: Dynamic Load Balancing Strategies for Parallel Computers. In: *Sci. Ann. Cuza Univ.* 11 (2002), S. 110–120

[67] PRIESE, L. ; WIMMEL, H.: *Theoretische Informatik: Petri-Netze*. Springer, 2003 (Springer-Lehrbuch). – ISBN 9783540442899

[68] RAMMIG, Franz J.: *Systematischer Entwurf digitaler Systeme - von der Systeme- bis zur Gatter-Ebene*. Teubner, 1989 (Leitfäden und Monographien der Informatik). – 1–353 S. – ISBN 978–3–519–02265–7

[69] RANKIN, C.: The software testing automation framework. In: *IBM Syst. J.* 41 (2002), Januar, Nr. 1, S. 126–139. – ISSN 0018–8670

[70] REORDA, M.S. ; PENG, Z. ; VIOLANTE, M.: *System-level Test and Validation of Hardware/Software Systems*. Springer, 2005 (Springer Series in Advanced Microelectronics). – ISBN 9781852338992

[71] ROSSBERG, Christian ; FROSS, André ; FROSS, Daniel ; HEINKEL, Ulrich: Beacon frame based network simulation using TrueTime network simulator. In: *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. Brussels, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010 (SIMUTools '10). – ISBN 978–963–9799–87–5, S. 47:1–47:2

[72] SAUTER, M.: *Grundkurs Mobile Kommunikationssysteme: UMTS, HSDPA und LTE, GSM, GPRS und Wireless LAN*. Vieweg+Teubner Verlag, 2010. – ISBN 9783834814074

[73] SAWALL, Achim: *Mobilfunkbetreiber gibt Android die Schuld*. Golem.de - IT-News für Profis, (Veröffentlicht: 29.01.2012) `http://www.golem.de/1201/89395.html`. – (abgerufen am 31.01.2012)

[74] SCHIEFERDECKER, Ina ; GROSSMANN, Jürgen: Testing hybrid control systems with TTCN-3: an overview on continuous TTCN-3. In: *STTT* 10 (2008), Nr. 4, S. 383–400

[75] SCHIEFERDECKER, Ina ; STEPIEN, Bernard: Automated Testing of XML/SOAP based Web Services. In: *KiVS*, 2003, S. 43–54

[76] SCHMATZ, Klaus-Dieter: *Java 2 Micro Edition - Entwicklung mobiler Anwendungen mit CLDC und MIDP*. dpunkt.verlag, 2004. – I–XIV, 1–332 S. – ISBN 978–3–89864–271–2

[77] SCHNEIDER, Kurt: *Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. Dpunkt, 2007

[78] SCHWEIZER, W.: *Matlab kompakt*. Oldenbourg, 2006. – ISBN 9783486580822

[79] SHIRAZI, B.A. ; HURSON, A.R. ; KAVI, K.M.: *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Society Press, 1995 (Systems Series). – ISBN 9780818665875

[80] SINGH, Rajat P.: Managing Software Testing Automation Framework with Rational Quality Manager / IBM. 2012. – Forschungsbericht

[81] SPILLNER, Andreas ; LINZ, Tilo: *Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester, Foundation Level nach ISTQB-Standard (3. Aufl.)*. dpunkt.verlag, 2005. – I–XX, 1–276 S. – ISBN 978–3–89864–358–0

[82] SPORER, Mathias: *Konsistenzerhaltende Techniken für generierbare Wissensbasen zum Entwurf eingebetteter Systeme*, Diss., 2007. – 1–351 S.

[83] STANKOVIC, J.A.: An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling. In: *IEEE Transactions on Computers* 34 (1985), S. 117–130. – ISSN 0018–9340

[84] SUHL, Leena ; MELLOULI, Taieb: *Optimierungssysteme: Modelle, Verfahren, Software, Anwendungen (Springer-Lehrbuch)*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2007. – ISBN 3540261192

[85] TAO, L. ; NARAHARI, B. ; ZHAO, Y.C.: Heuristics for Mapping Parallel Computations to Parallel Architectures. In: *Heterogeneous Processing, 1993. WHP 93. Proceedings. Workshop on*, 1993. – ISSN 1066–1220, S. 36 –41

[86] TEICH, Jürgen: *Digitale Hardware/Software-Systeme - Synthese und Optimierung*. Springer, 1997. – I–XVII, 1–514 S. – ISBN 978–3–540–62433–2

[87] THALLER, Georg E.: *Verifikation und Validation - Software-Test für Studenten und Praktiker*. Vieweg, 1994 (Vieweg Lehrbuch Informatik). – I–X, 1–319 S. – ISBN 978–3–528–05442–7

[88] THOMÉ, Bernhard (Hrsg.): *Systems engineering: principles and practice of computer-based systems engineering*. Chichester, UK : John Wiley and Sons Ltd., 1993. – ISBN 0–471–93552–2

[89] UTTENDORFER, Thilo A.: *Design und Implementierung eines verteilten Regressionstest-Frameworks*, Fachhochschule Furtwangen, Diplomarbeit, 2005

[90] VANDERBEI, Robert J. ; SHANNO, David F.: An Interior-Point Algorithm for Nonconvex Nonlinear Programming. In: *Computational Optimization and Applications* 13 (1999), April, Nr. 1, S. 231–252

[91] VARGA, András ; HORNIG, Rudolf: An overview of the OMNeT++ simulation environment. In: *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. Brussels, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008 (Simutools '08). – ISBN 978–963–9799–20–2, S. 60:1–60:10

[92] VODEL, Matthias: *Wissenschaftliche Schriftenreihe Eingebettete Selbstorganisierende Systeme*. Bd. 9: *Funkstandardübergreifende Kommunikation in Mobilen Ad Hoc Netzwerken*. Universitätsverlag Chemnitz, 2010. – ISBN 978–3–941003–18–7

[93] VODEL, Matthias ; HARDT, Wolfram: Data Aggregation in Resource-Limited Wireless Communication Environments - Differences Between Theory and Praxis. In: *Proceedings of the International Conference on Control, Automation and Information Sciences*. Ho Chi Minh City, Vietnam : IEEE Computer Society, November 2012

[94] VODEL, Matthias ; SAUPPE, Matthias ; CASPAR, Mirko ; HARDT, Wolfram: SimANet - A Large Scalable, Distributed Simulation Framework for Ambient Networks. In: *Recent Advances in Information Technology and Security - Journal of Communications (EI Compendex)* 3 (2008), December, Nr. 7, S. 11–19. – ISSN: 1796-2021

[95] VODEL, Matthias ; SAUPPE, Matthias ; HARDT, Wolfram: Parallel High-Performance Applications with MPI2Java - A Capable Java Interface for MPI 2.0 Libraries. In: *Proceedings of the Asia-Pacific Conference on Communications (APCC2010)*. Auckland / New Zealand : IEEE Computer Society, October 2010. – ISBN 978–1–4244–8127–9

[96] WILES, Anthony: ETSI Testing Activities and the Use of TTCN-3. In: *Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*. London, UK, UK : Springer-Verlag, 2001 (SDL '01). – ISBN 3–540–42281–1, S. 123–128

[97] WISE, J.A. ; HOPKIN, V.D. ; STAGER, P.: *Verification and Validation of Complex Systems: Human Factors Issues*. Springer, 1993 (NATO ASI series: Computer and systems sciences). – ISBN 9783540565741

[98] ZHANG, Jia: A Mobile Agent-Based Tool Supporting Web Services Testing. In: *Wirel. Pers. Commun.* 56 (2011), Januar, Nr. 1, S. 147–172. – ISSN 0929–6212

Thesen

Auf Grundlage der vorliegenden Arbeit werden die folgenden Thesen aufgestellt:

1. Die Validierung Dienstbereitstellender Systeme durch die automatisierte Erzeugung von Lasten ergänzt den Systementwurfsprozeß und trägt zur Verbesserung der Qualität der entwickelten Systeme bei.
2. Die technologie- und implementierungsunabhängige Beschreibung der Validierungsanforderungen und -rahmenbedingungen mittels mathematischer Formalismen erlaubt die Definition abstrakter Testziele und stellt die Basis für die automatisierte Ausführung der Validierung dar.
3. Das Testpartitionierungsproblem für die automatisierte Ausführung der Validierung läßt sich mit einer geeigneten Heuristik in Polynomialzeit (suboptimal) lösen.
4. Durch die zentralisierte Kontrolle der Validierungsausführung können auch ressourcenbeschränkte Endgeräte als Klienten eingebunden werden.
5. Die Testbencharchitektur und die effiziente Heuristik erlauben einen dynamischen Ausgleich von Ressourcenmangel und Störungen im Klientennetzwerk während der Validierungsausführung.
6. Während der automatisierten Ausführung der Validierung kann eine gute Annäherung der Ist-Last an die Soll-Last erreicht werden.
7. Die Abweichung der generierten Ist-Last von der geforderten Soll-Last ist maßgeblich abhängig von der klientenseitigen Technologie zur Erzeugung der Last.
8. Durch die geeignete Modularisierung ist die entwickelte Testbench weitestgehend unabhängig vom zu testenden System.