Bedarfsgerechte Auswahl der Dienstqualität für Mobile Cloud-unterstützte Anwendungen

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von
Marvin Ferber
aus Werdau

1. Gutachter : Prof. Dr. Thomas Rauber
2. Gutachter : Prof. Dr. Stefan Jablonski

Tag der Einreichung : 27. 05. 2014
Tag des Kolloquiums : 06. 08. 2014

Danksagung Ich danke meinem Doktorvater Prof. Dr. Thomas Rauber für die langjährige Unterstützung und die Möglichkeit meinen eigenen wissenschaftlichen Stil zu entwickeln und meinen Forschungsinteressen nachzugehen. Ich danke meinen Kollegen am Lehrstuhl für Angewandte Informatik 2 der Universität Bayreuth für fruchtbare Diskussionen und Anregungen, aber auch für berechtigte Kritik. Außerdem möchte ich mich auch bei allen anderen Wegbegleitern bedanken, mit deren Hilfe Forschungsideen sich in Implementierungen und abschließend in Publikationen transformiert haben. Hierfür möchte ich insbesondere Dr. Sascha Hunold und Mario Henrique Cruz Torres danken. Weiterhin möchte ich auch den Studenten danken, die im Rahmen von HiWi-Tätigkeiten und Abschlussarbeiten Ergebnisse erzielten, die zum Entstehen dieser Arbeit beigetragen haben. Ich danke zudem meinen Eltern Ute und Mathias sowie meinem Bruder Henning ganz herzlich, da sie mich von Anfang an motiviert und unterstützt haben, damit ich diese Arbeit beginnen und abschließen kann. Danken möchte ich ebenso Bettina für ihr geübtes Auge bei der Suche nach Schreibfehlern. Erwähnt werden sollen auch die unzähligen Freunde, die mich in den letzten Jahren nach Kräften unterstützt haben. Vielen Dank!

Besonderer Dank gilt meiner Frau, ohne deren umfangreiches „Hintergrundengagement“ und deren Motivationsarbeit diese Arbeit wohl nicht so entstanden wäre. Danke Anne!

Kurzfassung

Mobilgeräte wie Smartphones, Tablets und kleine Laptops finden immer größere Verbreitung. Dadurch wächst auch der Bedarf an mobil genutzten Applikationen. Aufgrund der zugunsten der Batterielaufzeit eingeschränkten Ressourcen können jedoch berechnungsintensive Anwendungen auf diesen Mobilgeräten nur schwer realisiert werden. Eine mögliche Lösung ist die Anwendung von Cloud-Ressourcen zur zeitlich begrenzten Unterstützung solcher Applikationen. Cloud Computing hat sich in letzter Zeit immer mehr etabliert und meint unter anderem das Mieten von Rechnern auf feingranularer zeitlicher Basis nach dem Selbstbedienungsprinzip (Infrastructure as a Service (IaaS)). Darauf aufbauende Plattform-Dienste (Platform as a Service (PaaS)) führen Programmcode der Applikationen (Software as a Service (SaaS)) aus. Bei SaaS handelt es sich um eine Art Software, die zeitlich flexibel via Internet genutzt wird. In der Literatur vorgestellte Ansätze zur Anwendung von Cloud Computing zur Unterstützung mobiler Applikationen lassen jedoch häufig die für Cloud-Ressourcen anfallenden Kosten außer Acht und beachten die teils stark schwankende mobile Netzwerkperformance nicht. Ziel dieser Arbeit ist darum die Erforschung von Techniken zur bedarfsgerechten Bereitstellung von Compute-Ressourcen für mobil genutzte Anwendungen unter Beachtung situationsbezogener Eigenschaften. Es wird eine Middleware für Java vorgestellt, die auf der PaaS-Schicht angesiedelt ist und die die Cloud-unterstützte Ausführung mobiler Anwendungen via Code-Offloading ermöglicht. Dabei wird vormals lokal ausgeführter Programmcode zur Laufzeit zu einem Server übertragen und anschließend dort ausgeführt. Im Gegensatz zu anderen Middlewares ist es hier möglich, die Qualität der zu verwendenden Cloud-Ressource zur Laufzeit auszuwählen. Dazu ist es notwendig, die verfügbare Netzwerkperformance und die erwartete Ausführungszeit der mobilen Applikation zu prognostizieren. Zunächst wird die Netzwerkperformance von 3G-Mobilfunk und WLAN untersucht. Danach wird ein Pipelining-basiertes Kommunikationsprotokoll erarbeitet, welches die Gesamtperformance und die Vorhersagbarkeit der erreichbaren Datenübertragungsrate verbessert. Zur Laufzeitprognose werden durch Regressionsanalyse Applikationsprofile erstellt, die auf Monitoring-Daten basieren, die die Middleware für jede Anwendung sammelt. Über die API der Middleware kann dann in Abhängigkeit situationsbezogener Parameter (Netzwerkperformance und Eingabedaten) eine bedarfsgerechte Auswahl an Cloud-Ressourcen präsentiert werden. Da nicht für jede Anwendung Applikationsprofile erstellt werden können, wird eine Kategorisierung mobiler Anwendungen in taugliche und untaugliche Applikationstypen erarbeitet. Da die Middleware auf IaaS-Ressourcen eines Cloud-Anbieters basiert, werden Lastbalancierungsstrategien analysiert, die eine optimale Auslastung der Ressourcen bei gleichzeitig konstanter Performance für alle Nutzer garantieren. Hierfür werden Simulationen zum Verhalten der Strategien auf einer IaaS-Cloud durchgeführt. Die Evaluation der prototypischen Implementierung der Middleware und insbesondere der Güte der vorhergesagten Performance wird anhand realer Applikationen durchgeführt. Dazu wird die Raytracing-Funktionalität zur Erstellung fotorealistischer Bilder einer 3D-Hausdesignsoftware und eine Applikation zur fortlaufenden Kamerabildverarbeitung auf Smartphones genutzt. Die erzielten Ergebnisse unter Benutzung von WLAN und 3G-Netzwerken zeigen, dass eine sinnvolle Auswahl passender Ressourcentypen zur Laufzeit nicht nur möglich, sondern auch wichtig ist, um für den Nutzer unnötige Kosten aufgrund unzureichend ausgelasteter Cloud-Ressourcen zu vermeiden.

Abstract

Mobile devices such as Smartphones and Tablets are becoming more and more popular. As a result, the need for mobile applications also grows. Because of the limited resources of such battery-powered devices, compute-intensive applications can hardly be executed on them. One solution to overcome this restriction is the time-limited use of cloud resources to assist mobile applications on demand. Cloud Computing has emerged in the past years. It includes the rental of compute resources on a fine-grained timely basis using self-service. This is known as Infrastructure-as-a-Service (IaaS). On top of this, a Platform-as-a-Service (PaaS) is set up that hosts the application code of Software-as-a-Service (SaaS). SaaS is a kind of software that is used via Internet and that is billed on a timely flexible model, e.g., the pay-per-use model. The use of Cloud Computing to assist mobile applications has been proposed in the literature already. However, the cost for cloud usage and the influence of the current mobile network performance are often not discussed. Consequently, this work aims at the provision of cloud resources to assist mobile applications in accordance with the current Quality of Service (QoS) demands of the user. To achieve this goal, the performance of 3G and WLAN network connections is studied in a first step. Based on the results, a novel communication protocol is developed, which uses pipelining. This protocol achieves a better overall communication performance and a better predictability of the throughput in contrast to widely used protocols such as HTTP. Furthermore, a middleware for Java is presented, which supports the execution of cloud-assisted mobile applications. The middleware is located at the PaaS layer. Code-Offloading is used to execute formerly locally executed program code on a remote server. This requires to transfer and install program code to the server at runtime. The middleware manages a pool of cloud resources from an IaaS provider. Thus, load-balancing and strategies for reuse are analysed in order to maximize resource usage while assuring a constant performance for each user. This analysis is done via simulation, which is based on parameters from an existing cloud environment. It is possible to select the quality of the underlying resources for a cloud-assisted mobile application at runtime. To facilitate the decision, a runtime prediction of the expected performance is done for available cloud resources. The prediction is based on monitoring data collected by the middleware for each application. A profile is created for each application using regression analysis. A common API is used to calculate cloud options at runtime in accordance with the current network performance and current input data. Applications are categorized in order to identify suitable applications that can be equipped with such a runtime prediction. The quality of the prediction is evaluated using real applications. A software for ray tracing of photo-realistic images of a 3D home designer and a software for streaming image processing of camera pictures on an Android device are used. Results show that a reasonable selection of resource types can be presented at runtime using 3G and WLAN mobile network configurations. Moreover, it is necessary to review the available options carefully in order to avoid monetary overhead for the user, which may be caused by inefficiently utilized cloud resources.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

3G	Mobilfunkstandard der 3. Generation
3GPP	3rd Generation Partnership Project
4G	Mobilfunkstandard der 4. Generation
ALU	Arithmetic Logic Unit
AMI	Amazon Machine Image
AP	Accesspoint (WLAN)
API	Application Programming Interface
AR	Augmented Reality
ARM	Acorn Risc Machine (Computerprozessor)
AWS	Amazon Web Services
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSMA/CA	Carrier Sense Multiple Access / Collision Avoidance
DSL	Digital Subscriber Line
EC2	Elastic Compute Cloud
ECU	Elastic Compute Unit
FDD	Frequency Division Duplex
FPU	Floating Point Unit
FPS	Frames per Second
GC	Garbage Collector, Garbage Collection
GSM	Global System for Mobile Communications
HSDPA	High Speed Downlink Packet Access
HSPA	High Speed Packet Access
HSUPA	High Speed Uplink Packet Access
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IDL	Interface Definition Language
I/O	Input / Output
JIT-Compiler	Just-in-Time-Compiler
JMS	Java Message Service
JDK	Java Development Kit

JRE	Java Runtime Environment
JSON	Javascript Object Notation
JVM	Java Virtual Machine
LTE	Long Term Evolution
MAC	Media Access Control
MAC	Message Authentication Code
MCC	Mobile Cloud Computing
MSS	Maximum Segment Size
MTU	Maximum Transfer Unit
OS	Operating System
PaaS	Platform-as-a-Service
QoS	Quality of Service
REST	Representational State Transfer
RFC	Request for Comments
RMI	Remote Method Invocation (Java)
RPC	Remote Procedure Call
RTT	Round Trip Time
SaaS	Software-as-a-Service
SASL	Simple Authentication and Security Layer
SLA	Service-Level Agreement
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SSH	Secure Shell
SSL/TLS	Secure Socket Layer / Transport Layer Security
TDD	Time Division Duplex
TCP/IP	Transmission Control Protocol / Internet Protocol
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
VM	Virtuelle Maschine
WAR	Web Application Archive
WCDMA	Wideband Code Division Multiple Access
WLAN	Wireless Local Area Network
WSDL	Web Service Description Language
XML	Extensible Markup Language

1 Einleitung

Die Verwendung mobiler Endgeräte zur Nutzung von Internet-Angeboten hat in den letzten Jahren stark zugenommen und es wird erwartet, dass dieser Trend anhält[1]. Zu mobilen Endgeräten gehören unter anderem Internet-fähige Mobiltelefone, sogenannte Smartphones, deren Absatz sich von 2010 zu 2011 um rund 61% auf über 490 Mio. verkaufte Geräte erhöht hat [Int12]. Seit 2010 werden außerdem Tablet-PCs mit großem Touchscreen immer beliebter. Auch kleine Laptops mit Tastatur und ohne Touchscreen, wie etwa Netbooks, können zu den mobilen Geräten gezählt werden. Diese mobilen Endgeräte erlauben durch mobile Datenübertragung unterwegs die Nutzung von vormals stationär genutzten Programmen, wie etwa Webbrowser. Zusätzlich bieten diese Geräte auch Software mit ausschließlich mobilem Einsatzzweck, etwa Fußgängernavigation, an. Schlüsseltechnologien der mobilen Kommunikation sind in diesem Zusammenhang die Mobilfunkstandards der dritten Generation (3G) wie UMTS sowie WLAN Standards für den räumlich begrenzten Einsatz, beispielsweise im Büro oder zu Hause. Im 4. Quartal 2011 gab es bereits über 1 Mrd. Mobilfunkteilnehmer in 3G Netzen weltweit [MW12]. Die zunehmend mobile Nutzung von Anwendungen stellt jedoch neue Herausforderungen an die Entwicklung von Software. Dies betrifft unter anderem die Anpassung an eine berührungssensitive Touchoberfläche und die Berücksichtigung des ausschließlich kabellosen Netzwerkzugangs. Da mobile Geräte im Gegensatz zu stationären PCs nur begrenzte Speicher- und Verarbeitungsfähigkeiten haben, ist die Verwendung von entfernten Funktionen via Internet eine Alternative [KL10]. Ziel ist die Optimierung mobiler Ressourcen wie Akkulaufzeit und Speichernutzung sowie die schnellere Ausführung ressourcenintensiver Applikationen. Hierbei haben sich vor allem Dienste zur Online-Speicherung von Daten, wie etwa *DropBox* [2] oder *Microsoft SkyDrive* [3], etabliert [Sos10]. Aber auch Funktionen wie Büroprogramme und Online-Spiele werden immer stärker genutzt. Im Folgenden werden diese Art Applikationen unter dem Begriff *Mobile Server-unterstützte Anwendungen* zusammengefasst. *Mobile Server-unterstützte Anwendungen* werden da genutzt, wo mobile Kommunikationsgeräte genutzt werden, beispielsweise zu Hause, im Büro oder im Urlaub.

Mobile Endgeräte und stationäre PCs oder Laptops unterscheiden sich in vielen Eigenschaften. Tabelle 1.1 fasst gängige Eigenschaften verschiedener mobiler Endgeräte zusammen. Die Mehrheit der Mobilgeräte nutzt heutzutage stromsparende Prozessoren auf ARM-Basis [For12]. ARM-Prozessoren sind im Bereich eingebetteter Systeme sehr beliebt und daher

[1] Der Anteil mobil aufgerufener Webseiten ist seit Anfang 2009 stetig gestiegen und erreichte im Dezember 2013 ca. 22%. Dabei ist der Anstieg im Vergleich zum Vorjahresmonat mit ca. 58% erheblich. (siehe Onlineressource `http://gs.statcounter.com/` (abgerufen am 24.3.2014))

[2] *„Dropbox ist ein kostenloser Service, mit dem Sie Ihre Fotos, Dokumente und Videos überall dabeihaben und leicht mit anderen teilen können. Das Unternehmen wurde 2007 von den beiden MIT-Studenten Drew Houston und Arash Ferdowsi gegründet, die genug davon hatten, sich Dateien per E-Mail zuzusenden, nur damit sie auf mehreren Computern damit arbeiten konnten.“* (Onlineressource `https://www.dropbox.com/about` (abgerufen am 1.11.2013))

[3] *„SkyDrive ist ein kostenloser Onlinespeicher für Dateien, auf den Sie von beliebigen Geräten aus zugreifen können. [...] Mit SkyDrive können Sie von überall auf Ihre Fotos, Dokumente und anderen wichtigen Dateien zugreifen.“* (Onlineressource `http://windows.microsoft.com/de-at/skydrive/download`(abgerufen am 1.11.2013))

Tabelle 1.1: Kategorisierung und Eigenschaften mobiler Endgeräte*

	Smartphone	Tablet	Netbook	Laptop/Desktop
Displaygröße	3-5”	7-10”	10-13”	14-24”
Displayauflösung	800x480	1024x600	1366x768	1920x1080
CPU	1 Kern ARM	2 Kerne ARM	2 Kerne x86	4 Kerne x86
RAM	512 MB	1 GB	2 GB	4 GB
Persistenter Speicher	2 GB	8 GB	320 GB	500 GB
Betriebssysteme	Android/iOS/Windows Phone/etc.		Linux/MacOS/Windows/etc.	
Eingabemethode	Touchscreen		Tastatur und Mouse/Touchpad/Trackpoint	

* = Häufigste Werte ermittelt aus den auf www.geizhals.at am 8. März 2013 in der Europäischen Union gelisteten Produkten in der jeweiligen Kategorie.

eher von geringem Energieverbrauch als von hoher Verarbeitungsgeschwindigkeit geprägt.

> „*There are a number of physical features that have driven the ARM processor design. First, portable embedded systems require some form of battery power. The ARM processor has been specifically designed to be small to reduce power consumption and extend battery operation-essential for applications such as mobile phones and personal digital assistants (PDAs).*“[SSW04, S.5]

Außerdem haben mobile Geräte oft eine Fülle von integrierten Sensoren wie beispielsweise Kamera, GPS, Höhenmesser oder Beschleunigungssensor, was für Laptops und stationäre PCs unüblich ist. Stationäre PCs haben neben einer höheren Verarbeitungsgeschwindigkeit oft auch eine großzügigere Ausstattung mit Arbeitsspeicher und persistentem Speicher (Festplatte, etc.). Bezüglich Display-Auflösung haben die Mobilgeräte stark aufgeholt, wobei viele Smartphones und Tablets aufgrund der geringen Baugröße dennoch eher kleinere Auflösungen haben als die meisten stationären PCs und Laptops. Um die Eigenschaften mobiler Geräte besser zu unterstützen, wurden spezielle Betriebssysteme für diese Geräte entwickelt, beispielsweise *iOS* und *Android* [BK11]. Kann man auf einem Laptop üblicherweise verschiedene Betriebssysteme und Anwendungen beliebiger Art installieren, so ist bei Mobilgeräten oftmals mit dem Kauf des Geräts die darin verwendete Software-Plattform festgelegt. Dies äußert sich einerseits darin, dass der Entwickler durch einfache Schnittstellen unterstützt wird und dem Anwender eine reichhaltige Anwendungsauswahl präsentiert werden kann. Andererseits ist die Interoperabilität niedriger, man spricht von *Vendor Lock-in*[4]. Da solche mobilen Geräte die Nutzung des Internets zum Datenaustausch essenziell vorsehen, werden sogenannte App-Stores verwendet, um Applikationen direkt von dort herunterzuladen und zu installieren. Andere Arten der Installation, etwa via CD oder Diskette, sind hier nicht mehr vorgesehen.

Verschiedene Internet-Dienste, die besonders mit der starken Verbreitung mobiler Endgeräte an Bedeutung gewonnen haben, wie etwa *Youtube* oder *Dropbox*, profitieren von einer

[4] „*Vendor lock-in, or just lock-in, is the situation in which customers are dependent on a single manufacturer or supplier for some product (i.e., a good or service), or products, and cannot move to another vendor without substantial costs and/or inconvenience.*“ (Onlineressource `http://www.linfo.org/vendor_lockin.html`(abgerufen am 1.11.2013))

neuen Art der Bereitstellung von Server-Ressourcen, die nach dem *pay-per-use* Modell für jedermann im Internet nutzbar ist. Dieses Konzept erlaubt es beispielsweise, Server-Maschinen auf Stundenbasis zu mieten und zu bezahlen. Man spricht bei diesem Konzept von *Cloud Computing* oder *Utility Computing*. *Cloud Computing* Technologien haben den Server-Markt gegenüber traditionellen Rechenzentren stark flexibilisiert. In traditionellen Rechenzentren können Server auf Monatsbasis gemietet werden. Man kann dort üblicherweise vorkonfektionierte Betriebssysteme der Rechenzentrumsbetreiber nutzen oder ein angepasstes System selbst installieren. Cloud-Betreiber bieten heute Rechnerressourcen, Plattformen und Applikationen auf feingranularer Basis zur Miete an. Dies gilt vor allem für die Konfigurationsmöglichkeiten und die feingranulare Abrechnung. Beliebige Betriebssysteme können binnen weniger Minuten auf konfigurierbarer Serverhardware aufgesetzt und in Betrieb genommen werden. Diese flexible und schnelle Konfigurierbarkeit und Nutzbarkeit von Ressourcen hat ganz neue Arten von Applikationen hervorgebracht, die ohne solche Cloud-Technologie nur schwer realisierbar wären. Hierzu zählen beispielsweise Multicast-Videoanwendungen oder soziale Netzwerke, die immer mehr Multimedia-Inhalte verknüpfen und aufbereiten. Dabei versuchen Cloud-Anbieter, durch bessere Auslastung der Hardware die Kosten zu reduzieren.

Eine Schlüsseltechnologie in diesem Zusammenhang ist die Virtualisierung von Ressourcen, welche es erlaubt, mehrere virtuelle Betriebssysteme auf einer physikalischen Maschine auszuführen. Dadurch kann beispielsweise ein System mit 4 CPU-Kernen mit einem leichtgewichtigen Hostbetriebssystem ausgestattet werden, auf dem dann zwei Instanzen virtueller Maschinen mit je 2 CPU-Kernen und einer Partition des verfügbaren Arbeitsspeichers ausgeführt werden. Die virtuellen Maschinen können dabei verschiedene andere Betriebssysteme beherbergen. Durch Virtualisierung erhöht sich auch die Energieffizienz der Serverlandschaft, was zu einem weiteren positiven Effekt von *Cloud Computing* führt. Aufgrund dieser positiven Eigenschaften versuchen Entwickler, Cloud-Technologie gewinnbringend in ihre Applikationen und Dienste einzubauen.

Kombiniert man *Cloud Computing* mit der Nutzung von mobilen Endgeräten, so spricht man von *Mobile Cloud Computing* (MCC). Solche Anwendungen lagern berechnungsintensive Teile/Module der Applikation auf entfernte Server aus. Dies wird häufig als Client/Server Partitionierung bezeichnet. Dadurch können Limitierungen bezüglich Hauptspeicherbedarf, CPU-Leistung oder Speicherkapazität mobiler Endgeräte umgangen werden. Aber auch die Steigerung der Akkulaufzeit und die Verbesserung der Zuverlässigkeit mobiler Anwendungen sind Ziele des MCC [DLNW11]. Diese Arbeit untersucht, wie rechenintensive Anwendungen durch Cloud-Unterstützung für mobile Endgeräte besser nutzbar gemacht werden können. Dazu wird der Begriff *Mobile Cloud-unterstützte Anwendungen* eingeführt und deren Eigenschaften und Modellierungsmöglichkeiten werden untersucht. Die Erstellung dieser Arbeit ist dabei ein Prozess, der über ca. drei Jahre andauerte. Während der Bearbeitung der einzelnen Fragestellungen in den Jahren 2011 bis 2014 wurden von Unternehmen und Forschergruppen teils neuere Technologien und Ergebnisse bekannt, die in dieser Arbeit nicht mehr umfänglich berücksichtigt werden konnten. Auf einige dieser Ergebnisse und deren Bedeutung für die vorliegende Arbeit geht das Abschlusskapitel ein. Zunächst werden im nachfolgenden Kapitel wichtige Begriffe erläutert und anschließend die Problemstellung und Struktur der Arbeit vorgestellt.

1.1 Begriffliche Grundlagen

Dieser Abschnitt führt einige grundlegende Begriff ein, die für diese Arbeit bedeutsam sind. In dieser Arbeit sind wichtige Eigennamen, Firmennamen und Technologien *kursiv* dargestellt. Abkürzungen werden bei ihrer ersten Verwendung eingeführt und gegebenenfalls kurz erläutert. Sie können zudem im Abkürzungsverzeichnis nachgeschlagen werden.

1.1.1 Internet und Technologien

Das Internet, welches aus verschiedenen Forschungsnetzwerken wie dem ARPANET hervorging, basiert auf einer Reihe von Netzwerkprotokollen, die in den 1970er Jahren entwickelt wurden [HL08]. Dieser TCP/IP-Protokollstapel besteht aus Protokollen, die in Schichten organisiert sind und aufeinander aufbauen. Wichtige Protokolle sind in diesem Zusammenhang das *Internet Protocol* (IP), welches jedem Netzwerkgerät eine Adresse zuweist und das *Transmission Control Protocol* (TCP), welches zuverlässige Ende-zu-Ende Verbindungen zwischen Netzwerkgeräten aufbaut [KR08]. Weitere Details zu den unteren hardwarenahen Schichten des TCP/IP-Protokollstapels werden in Kapitel 4 im Rahmen der Untersuchung verschiedener kabelloser Netzwerktechnologien betrachtet. Hier sollen zunächst weitere wichtige Protokolle der Anwendungsschicht eingeführt werden. Dazu zählen vorrangig das *Hypertext Transfer Protocol* (HTTP) und die *Hypertext Markup Language* (HTML), welche zusammen Anfang der 1990er Jahre den Beginn des *Web* begründeten. Das zuvor nur zum Austausch von Daten genutzte Internet wurde so um eine Komponente erweitert, die eine einfache Verknüpfung von Texten und Daten ermöglichte. Damit weckte das Internet auch das Interesse von Privatpersonen und Firmen.

Auf Basis des Internets und des *Web* wurden auch neue Architekturen für verteilte Anwendungen vorgestellt. So wurde die noch in den 1990er Jahren favorisierte Komponentenarchitektur, wie sie beispielsweise die *Common Object Request Broker Architecture* (CORBA) implementiert, durch eine lose gekoppelte *Service-oriented Architecture* (SOA) abgelöst. SOA kann wie folgt definiert werden.

> „*A Service-Oriented Architecture (SOA) is a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus. A service consists of a contract, one or more interfaces, and an implementation.*“[KBS05, S.57]

SOA beschreibt also, wie abgeschlossene, wiederverwendbare und selbstbeschreibende Dienste in standardisierter Art und Weise genutzt und kombiniert werden können. *Cloud Computing* ist untrennbar mit dem Dienstbegriff (Service) verbunden. Der Dienstbegriff beschreibt in der Informatik eine Funktionalität, der vor allem die Attribute Abgeschlossenheit, Standardkonformität, Verfügbarkeit via Netzwerk/Internet und daraus resultierend auch eine flexible/leichte Kombinierbarkeit zuzuordnen sind [Pap03, Erl06]. Eine sehr populäre Implementierung von SOA sind *Web Services*, die auf dem Austausch von *eXtended Markup Language* (XML) Nachrichten basieren. Das *Simple Object Access Protocol* (SOAP) wird dabei zum Nachrichtenaustausch genutzt. SOAP verwendet dabei in den meisten Fällen HTTP zur Kommunikation. Größter Kritikpunkt an SOAP *Web Services* ist der hohe Overhead durch die XML-Codierung. Deshalb wurden als eine Art Gegenbewegung zur weithin standardisierten SOAP *Web Service* Landschaft leichtgewichtige *RESTful Web Services* vorgestellt [Fie00].

REST (*Representational State Transfer*) hat als Ziel, die Struktur des Internets bestmöglich für die Kommunikation auszunutzen. Die Kernkonzepte des REST Architekturstils sind Zustandslosigkeit von Client/Server Verbindungen, Anwendbarkeit von (Web-)Caches, einheitliche Schnittstellen und die Anwendungen eines Schichtenmodells, was in diesem Zusammenhang mit transparenten Weiterleitungen gleichzusetzen ist. Da es außer HTTP keine weitere Implementierung von *RESTful Web Services* gibt, wird eine Ressource oder Dienst üblicherweise durch einen *Uniform Ressource Identifier* (URI) adressiert und liefert Daten dann in einer bei der Anfrage zu spezifizierenden Codierung zurück. Hierbei wird neben XML auch oft die wesentlich schlankere *JavaScript Object Notation* (JSON) verwendet. Zustände können über die Verwendung von *Hyperlinks* manipuliert werden.

Fast jede Programmiersprache bringt heute Konstrukte oder Bibliotheken zur Netzwerkkommunikation mit. Dabei gibt es neben Implementierungen für die genannten *Web Service* Technologien auch verschiedene Programmiersprachen-spezifische Implementierungen wie beispielsweise *Java Remote Method Invocation* (RMI) oder *C# .NET Remoting*. Diese Technologien sind zwar häufig nicht Plattform-übergreifend nutzbar, sind dafür aber aus der jeweiligen Sprache mit geringem Aufwand benutzbar, was zur Verkürzung der Entwicklungszeit beitragen kann.

1.1.2 Cloud Computing

Der Begriff *Cloud Computing* wird in der Wissenschaft kontrovers diskutiert [VRMCL08], was zu der Notwendigkeit führt, hier zunächst die für diese Arbeit relevanten Termini einzuführen und deren Bedeutung festzulegen. Der Begriff kann folgendermaßen zusammengefasst werden:

> „*Thus, Cloud Computing is the sum of SaaS and Utility Computing, but does not include Private Clouds.*“[AFG+09, S.1]

Diese Definition erscheint zunächst unbefriedigend, da sie weitere spezifische Termini enthält, die einer Erklärung bedürfen. Der Begriff *Utility Computing* meint die allgemeine Verfügbarkeit von Computerressourcen, wie dies etwa für Strom und Wasser der Fall ist, und deren Abrechnung nach tatsächlichem Verbrauch [Rap04]. In diesem Zusammenhang wird von *pay-per-use* und *pay-as-you-go* gesprochen. Dies entspricht der Nutzung von Ressourcen eines traditionellen Rechenzentrums, nur mit einer sehr kurzen Vertragslaufzeit, beispielsweise mit einer Abrechnung im Stundentakt. Eine solche kurzlebige Art der Bereitstellung und Abrechnung von Computerressourcen, wie persistenter Speicher oder Compute-Ressourcen, ist mit realer Hardware praktisch kaum zu realisieren. Allein die Konfiguration eines Rechners und die Installation der Software würden wohl schon Stunden in Anspruch nehmen. Stattdessen werden virtualisierte Ressourcen verwendet, die sich schnell und flexibel bereitstellen lassen. Virtualisierung von Maschinen gilt als Schlüsseltechnologie für *Utility Computing*.

Weiterhin wird in [AFG+09] ausgeführt, dass *Clouds* eben diese Menge von teils virtualisierten Ressourcen sind, die Anbieter im Rahmen des Begriffs *Utility Computing* bereitstellen. Stellt ein Anbieter seine *Cloud* öffentlich zur Verfügung, wie etwa *Amazon Web Services* (AWS)[5], dann spricht man von einer *Public Cloud*. Es gibt jedoch auch *Clouds*, die nur intern,

[5] *„Amazon Web Services bietet einen kompletten Satz von Infrastruktur- und Anwendungsservices, mit denen Sie fast alles von unternehmensweiten Anwendungen und Big Data-Projekten (große Datenmengen) bis hin zu gesellschaftlichen Spielen und mobilen Anwendungen in der Cloud ausführen können.“* (Onlineressource `https://aws.amazon.com/de/` (abgerufen am 1.11.2013))

etwa in großen Banken, genutzt werden und nur unternehmensintern zur Verfügung stehen. Dabei sprich man von *Private Clouds*.

Das US-amerikanische Standardisierungsinstitut NIST (*National Institute of Standards and Technology*) liefert folgende Definition von *Cloud Computing*, die ebenfalls oft zitiert wird:

> „*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.*“[MG11, S.2]

Dieses Dokument definiert die Charakteristiken *On-demand self-service*, *Broad network access*, *Resource pooling*, *Rapid elasticity* und *Measured service*. Dabei geht diese Definition über [AFG+09] hinaus, da sie explizit die Messbarkeit verschiedener Laufzeitparameter der Cloud-Dienste zu Optimierungszwecken fordert. Außerdem definiert [MG11] zusätzlich zu den *Private* und *Public Clouds* eine Kombination aus beiden, die *Hybrid Clouds*, und eine *Community Cloud*, welche eine Mischform aus *Public* und *Private Cloud* ist. In [MG11] wird jedoch nicht explizit das *pay-per-use* Prinzip zur Abrechnung gefordert.

In dieser Arbeit wird die Anwendung des Begriffs *Cloud Computing* auf *Public Clouds* eingeschränkt. Dies setzt auch das obligatorische Vorhandensein eines *pay-per-use* Abrechnungsmodells voraus. Darüber hinaus sind aber auch auf diese eingeschränkte Variante die in [MG11] genannten fünf Charakteristiken anwendbar. Im Bereich *Cloud Computing* stellen Dienste nicht nur Software-Funktionalität dar, wie etwa Wettervorhersage oder Suche im Branchenbuch, sondern auch virtuelle Hardware oder Ausführungsumgebungen für benutzerdefinierten Programmcode. Hierbei spricht man von verschiedenen Schichten, die ihre Dienste aufeinander aufbauen können. Eine Visualisierung des Schichtenmodells ist in Abbildung 1.1 zu sehen. In der genannten und weiterer Literatur wird häufig übereinstimmend zwischen den drei nachfolgend beschrieben Schichten unterschieden, die auch in dieser Arbeit zu Grunde gelegt werden.

- Infrastrukutur als Dienst (Infrastructure as a Service - IaaS) bietet dabei grundlegende Dienste wie virtuelle Maschinen, Netzwerk oder Festplattenspeicher an.
- Darauf aufbauend bieten Dienste der Plattform-Schicht (Platform as a Service - PaaS) die Möglichkeit, Programmcode direkt auf vorgefertigten Maschinen oder Servern auszuführen ohne dabei die Maschinen selbst installieren oder warten zu müssen. Hierbei werden dann häufig nur noch Programme einer ausgewählten Programmiersprache unterstützt. Im Gegenzug übernimmt die Plattform-Schicht das Verteilen der Applikation auf Ressourcen der darunter liegenden Schicht und außerdem die Lastbalancierung und das Anpassen der Anzahl von benötigten Ressourcen an die jeweilige Situation (Skalierung).
- Schließlich kann auf Basis der darunter liegenden Schichten als oberste Schicht Software als Dienst (Software as a Service - SaaS) angeboten werden. Dabei handelt es sich oft um Software, die via Webinterface im Browser genutzt wird.

Eine dem *Cloud Computing* sehr ähnliche Technologie ist das *Grid Computing*. Dieser Begriff stammt aus dem wissenschaftlichen Rechnen und beschreibt Bestrebungen, moderne

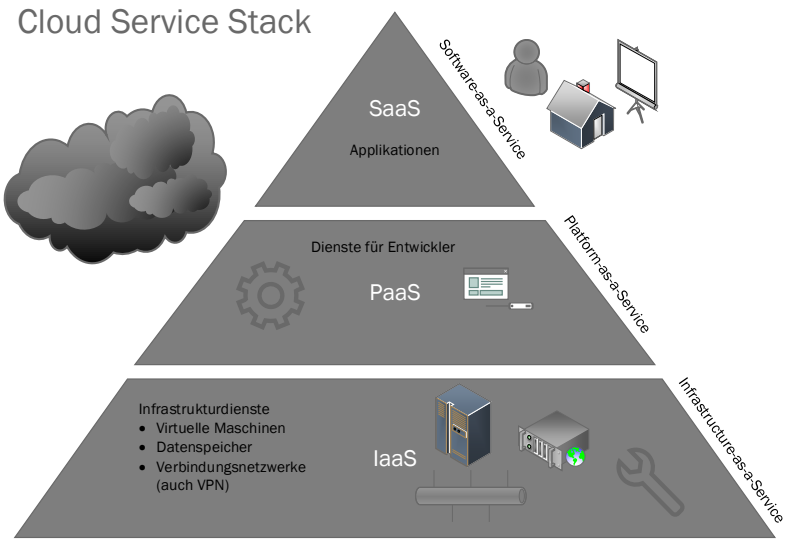


Abbildung 1.1: Schichteneinteilung von Cloud-Diensten. (vgl. Onlineressource `http://commons.wikimedia.org/wiki/File:Cloud_Services.gif` (abgerufen am 1.11.2013))

Hochleistungsrechner derart zu vernetzen, dass Berechnungen auf diesem zusammengeschalteten Netzwerk (genannt *Grid*) noch schneller erfolgen können. Ein *Computational Grid* ist eine im Vergleich zu einem Hochleistungsrechner lose gekoppelte Hardware- und Software-Infrastruktur, die einen konsistenten Zugriff auf die Kapazitäten von geografisch verteilten Hochleistungsrechnern ermöglicht [FK04]. Durch Standardisierungsbemühungen haben sich allgemein anerkannte Protokolle für den Zugriff auf entfernte (*Grid*-)Ressourcen etabliert, beispielsweise das *Web Service Ressource Framework* (WSRF) [Ban06]. Eine häufig verwendete Implementierung dieser *Grid*-Technologie stellt das *Globus Toolkit Version 4* dar [Fos05], welches von verschiedenen internationalen Partnern aus dem unternehmerischen und aus dem universitären Bereich stetig weiterentwickelt wird. *Grid* und *Cloud Computing* sind sich in einigen Punkten sehr ähnlich. Beide bieten einen standardisierten und zuverlässigen Zugang zu Computerressourcen. Auch im Bereich *Grid Computing* wurden Probleme wie Allokation und Management von Ressourcen, Optimierung durch Virtualisierung und ein einheitlicher Zugriff auf alle Teile des *Grids* bereits thematisiert.

Cloud Computing kann sowohl als Weiterentwicklung des traditionellen Rechenzentrumkonzepts verstanden werden, als auch als Erweiterung der *Grid*-Technologie [FZRL08]. Gegenüber einem traditionellen Rechenzentrum wurde vor allem die Flexibilität stark erhöht. Dies trifft nicht nur auf Kundenseite mit dem *pay-per-use* Modell zu. Auch auf Anbieterseite kann der Energieverbrauch durch Ressourcen-Konsolidierung und weniger Idle-Standby der Server optimiert werden. Zusätzlich sind nun in einem Rechenzentrum viel mehr Anwendungen möglich, was zu einer Vergrößerung von Rechenzentren und somit im Allgemeinen auch zu einer Steigerung der Effizienz führt. *Grid* und *Cloud* unterscheiden sich vor allem durch die anvisierte Nutzergruppe. Während *Grid*-Umgebungen überwiegend von Universitäten und anderen Forschungseinrichtungen betrieben werden und auch vornehmlich von diesen genutzt werden, stehen *Clouds* insbesondere auch einem privaten und unternehmerischen Publikum

zur Verfügung. Musste man sich früher für einen Zugang zu einem Großrechner bewerben und ein interessantes Forschungsvorhaben präsentieren, so reicht heute eine Kreditkarte aus, um sich einen Rechner für die Dauer eines Experiments zusammenzustellen. *Cloud Computing* richtet sich demnach insbesondere an den kommerziellen Sektor. Durch die damit verbundenen marktwirtschaftlichen Anforderungen stellt sich gerade beim *Cloud Computing* auch besonders die Frage nach einer effizienten Ressourcenausnutzung, um finanziellen Overhead zu vermeiden. Im Vergleich zu nicht virtualisierten Clustern, *Grid*-Infrastrukturen und Servern, die man bisher nutzte, um entfernte Ressourcen zur Verfügung zu stellen, bietet *Cloud Computing* folgende Vorteile:

- feingranulare Partitionierung der verfügbaren Hardwareressourcen,
- schnelle Konfiguration neuer Dienste durch Virtualisierung,
- einfachere Lastbalancierung durch Verschiebung der Lastbalancierung von der Applikationsschicht auf die Infrastrukturschicht,
- bessere Skalierungsmöglichkeiten durch schnelles Starten und Beenden von Diensten und dadurch auch effizientere Ausnutzung der verfügbaren Hardware.

1.1.3 Mobile Cloud Computing

Mobile Computing wird bereits seit ca. 20 Jahren aktiv beforscht [FZ94, Sat96]. Beim *Mobile Computing* geht man heute wie damals davon aus, dass man einen mobilen, batteriebetriebenen Computer benutzt, der in seiner Leistungsfähigkeit gegenüber einem Festrechner stark eingeschränkt ist. Dies ist unter anderem dem geringen Gewicht und der begrenzten elektrischen Energie geschuldet, die ein solches Gerät verfügbar hat [PKKB09]. Schon vor vielen Jahren wurden Techniken der verteilten Ausführung von mobilen Anwendungen vorgeschlagen mit dem Ziel, den Funktionsumfang von mobilen Anwendungen zu erhöhen. Hierbei spielten Techniken der Client/Server Programmierung oder auch abstraktere Konzepte wie *Verteilte Objekte* (*Remote Object*) eine Rolle. Technologien wie CORBA und vor allem auch Internettechnologien wie HTTP, HTML und andere trugen maßgeblich dazu bei, dass Qualität und Funktionsumfang *Mobiler Server-unterstützter Anwendungen* stetig stiegen.

Mobile Cloud Computing versucht nun durch die Eigenschaft der ständigen Verfügbarkeit von Rechen- und Speicherressourcen, auch für mobile Endgeräte Dienste anzubieten. Das *Mobile Cloud Computing Forum* fasst den Begriff wie folgt zusammen:

> „*Mobile Cloud Computing at its simplest, refers to an infrastructure where both the data storage and the data processing happen outside of the mobile device. Mobile cloud applications move the computing power and data storage away from mobile phones and into the cloud, bringing applications and mobile computing to not just smartphone users but a much broader range of mobile subscribers.*“Onlineressource `http://www.2exhibitions.com/conference.asp?show=Mobile_Cloud_Computing_Forum`(abgerufen am 1.11.2013)

Es geht also darum, für mobile Applikationen externe Ressourcen nutzbar zu machen. Dabei soll diese Technik für viele Nutzer gleichzeitg zur Verfügung stehen. Hierbei trifft man oft auf Begriffe wie Applikations-Partitionierung [CM10] oder Code-Offloading [KL10, GTF06]. Applikations-Partitionierung beschäftigt sich mit der Auftrennung von Applikationen in mobil

ausgeführte Teile und serverseitig ausgeführte Teile. Da dies oft von der gerade verfügbaren Netzwerkperformance abhängig ist, geschieht diese Partitionierung oft zur Laufzeit. Wenn ein Client Applikationscode zur Laufzeit auf dem Server installieren und verwenden kann, nennt man diesen Vorgang Code-Offloading. Dies hat den Vorteil, dass beliebiger Programmcode installiert werden kann und somit die Applikation auf dem Server nicht vorher installiert werden muss.

Anwendungen für *Mobile Cloud Computing* sind im Bereich *Mobile Commerce*, *Mobile Learning*, *Mobile Healthcare* oder *Mobile Gaming* zu finden [DLNW11]. Beliebt sind auch die eingangs erwähnten Anwendungen zum Abspeichern von Daten in der Cloud, wie etwa *Dropbox* oder *Skydrive*.

1.1.4 Quality of Service

Die Dienstqualität (*Quality of Service* (QoS)) beschreibt nicht-funktionale Eigenschaften eines Dienstes. Dabei findet man in der Literatur häufig die Kriterien Preis, Antwortzeit, Reputation, Zuverlässigkeit und Verfügbarkeit, die die nicht-funktionalen Eigenschaften beschreiben [ZBD+03]. Diese Kriterien werden benutzt, um bei gleichen funktionalen Eigenschaften eine Auswahl zwischen verschiedenen Diensten treffen zu können. Die genaue Spezifikation der Kriterien hängt dabei auch vom Anwendungsfall ab. Um ein genaues Verständnis und entsprechende Werte dieser Kriterien beispielsweise im Fall einer bilateralen Vereinbarung zu manifestieren, werden häufig sogenannte *Service-Level-Agreements* (SLA) zwischen Dienstanbieter (Provider) und Dienstnutzer (Consumer) ausgehandelt.

Der *Preis* beschreibt die Kosten, die durch Inanspruchnahme eines Dienstes entstehen. Hierbei wird im Cloud-Umfeld üblicherweise das *pay-per-use* Modell zu Grunde gelegt. Die *Antwortzeit* beschreibt die Bearbeitungsdauer einer Anfrage durch einen Dienst. Die Bearbeitungsdauer ist oft auch von einer gegebenen Eingabegröße abhängig. Die *Reputation* beschreibt die heute oft übliche Bewertung eines Dienstes durch die Endnutzer. Sie entspricht einer Produktbewertung und umfasst deshalb nicht nur die Bewertung nicht-funktionaler Eigenschaften. Dieses Kriterium wird in dieser Arbeit nicht weiter betrachtet. Die *Zuverlässigkeit* beschreibt die Wahrscheinlichkeit, dass eine Anfrage in einem gegebenen Zeitfenster korrekt beantwortet wird. Die Zuverlässigkeit hängt stark von der Anzahl der Anfragen pro Zeiteinheit und den in diesem Zeitraum zur Verfügung stehenden Ressourcen ab. Aber auch die Effizienz eines Dienstes bezüglich der Ausnutzung des Ressourcenpools spielt eine große Rolle. Die *Verfügbarkeit* eines Dienstes beschreibt die Wahrscheinlichkeit, dass der Dienst verfügbar ist. Üblicherweise wird diese als Verhältnis aus Dauer der Erreichbarkeit und einem Zeitinterval der Gesamtdauer definiert. Die *Verfügbarkeit* ist von sehr hardwarenahen Faktoren und auch den Netzwerkkomponenten abhängig und kann etwa durch redundante Rechenzentren verbessert werde. Der Softwareentwickler hat häufig nur begrenzten Einfluss auf die *Verfügbarkeit* eines Dienstes, den er zur Verfügung stellt. Hierzu handelt der Softwareentwickler einen Vertrag mit dem Rechenzentrumsanbieter als Hoster seiner Dienste aus.

1.1.5 Mobile Kommunikationsnetze

Die hier betrachteten kabellosen Kommunikationsnetze lassen sich in Flächennetze und Punktnetze unterscheiden. Flächennetze sind die allgemeinen Mobilfunknetze, die an nahezu jedem Punkt verfügbar sind. Im Fokus dieser Arbeit steht vor allem das UMTS-Netz in Deutschland. Mobilfunknetze sind in Funkzellen unterteilt, die mindestens durch eine Antenne repräsen-

tiert sind. Als Teilnehmer in solch einer Funkzelle ist man stets mit mindestens einer Antenne verbunden. In der Literatur zu mobilen Kommunikationsnetzen wird ein mobiles Endgerät auch oft als Handset oder mobile Equipment bezeichnet [Sau13]. Aus Kompatibilitätsgründen werden ältere Netzwerke bei Einführung einer neuen Kommunikationstechnologie nicht sofort abgeschaltet, auch wenn diese neue Antennenkonfigurationen benötigen. So existieren in Deutschland beispielsweise drei verschiedene Mobilfunktechnologien nebeneinander. Das ältere GSM-Netz, welches ursprünglich nicht für den Austausch von Daten vorgesehen war, wurde schon fast überall durch das auf Datenkommunikation optimierte UMTS-Netz erweitert. Zusätzlich erfolgt in Deutschland momentan der Aufbau des LTE-Netzes, welches gegenüber UMTS eine sehr stark beschleunigte Datenübertragung erlaubt und auch für stationäre Internetanschlüsse via Mobilfunknetz genutzt wird. Das LTE-Netz in Deutschland ist jedoch noch nicht flächendeckend verfügbar. Die Mobilfunknetze unterliegen einer ständigen Weiterentwicklung. So existierten aufbauend auf dem UMTS-Standard Weiterentwicklungen namens HSPA und HSPA+, die durch verschiedene Optimierungen eine höhere mittlere Bandbreite und geringere Netzwerklatenz pro Teilnehmer ermöglichen. In Deutschland sind maximale Bandbreiten zwischen 7,2 und ca. 21 Mbit/s je nach Anbieter verfügbar [PH13]. Im Vergleich zu kabelgebundenen Kommunikationsnetzen schwanken die Bandbreite und die Latenz je nach Nutzeranzahl und Verbindungsqualität.

Diese Schwankungen treten auch bei WLAN-Netzwerken auf, die die Gruppe der Punktnetze bilden. WLAN-Netzwerke sind immer nur punktuell verfügbar und bestehen im Allgemeinen aus einer Antenne, bzw. einem Zugangspunkt. Ausnahmen bilden hier größere Installationen, wie etwa in Bahnhöfen oder Universitäten, die aus mehreren Antennen bestehen und oftmals große Teile der Institution oder des Ortes mit WLAN-Zugang versorgen. Aber auch diese sind lokal begrenzt. WLAN wird auch häufig in privaten Haushalten eingesetzt, um einen kabelgebundenen Internetzugang für mehrere Personen/Geräte nutzbar zu machen. WLAN unterliegt ebenso einer ständigen Weiterentwicklung. Aktuell verfügt fast jeder Zugang über eine Bandbreite von 54 Mbit/s. Weit verbreitet sind mittlerweile auch Zugänge mit 300 Mbit/s und mehr, je nach Antennenkonfiguration [Sau13]. WLAN ermöglicht üblicherweise Latenzen im einstelligen Millisekunden-Bereich. Da der hinter dem WLAN liegende Internetzugang oft nicht die volle WLAN-Bandbreite durchleiten kann, begrenzt sich die verfügbare Bandbreite zum Internet ensprechend. Gängige Anschlüsse für private Haushalte liegen hier meist im zweistelligen Mbit-Bereich, jedoch ist die verfügbare Anschlussqualität in Deutschland regional sehr unterschiedlich, sodass mancherorts auch nur weniger als 1 Mbit/s zur Verfügung stehen kann. WLAN und UMTS wurden für diese Arbeit als Kandidaten für die drahtlose Kommunikation ausgewählt, da sie nicht nur eine angemessene Kommunikationsgeschwindigkeit bieten, sondern auch in vielen Geräten verfügbar sind[6].

1.2 Problemstellung und Zielsetzung

Die vorliegende Arbeit nähert sich dem Thema *Mobile Cloud Computing* zunächst aus Sicht des Anwenders, der auf mobilen Endgeräten Applikationen nutzt. Mobile Applikationen kön-

[6] Laut www.geizhals.at waren am 8. März 2013 ca. 40% aller in der Europäischen Union angebotenen Handys ohne Vertrag mit UMTS und WLAN ausgerüstet. Bei den Smartphones (Touchscreen, Auflösung ab 320x240 und GPS) sind es sogar ca. 92%. Notebooks mit WLAN machen über 99% aller Geräte aus. Dagegen sind nur ca. 10% aller Notebooks mit UMTS ausgerüstet. Tablets ohne WLAN gibt es nicht und ca. 28% aller in der EU angebotener Tablets sind mit UMTS ausgerüstet.

nen nach ihrem Einsatzzweck in zwei Gruppen unterteilt werden:

1. Anwender möchten die kabellosen Geräte nutzen, um ihre vormals stationär genutzten Programme auch mobil zu verwenden,

2. Anwender möchten neue Anwendungen nutzen, die die Fähigkeiten der mobilen Sensoren nutzbar machen.

Einsatzzweck 1 leitet sich aus dem aktuellen Trend in den Absatzzahlen von PCs, Notebooks und Mobilgeräten wie Tablets und Smartphones ab. Dieser lässt erkennen, dass zunehmend mehr Mobilgeräte und weniger PCs und Notebooks gekauft werden. Dies legt die Vermutung nahe, dass diese neuartigen Mobilgeräte zunehmend auch als Ersatz für PCs und Notebooks dienen. Einsatzzweck 2 macht sich den Unterschied zu Nutze, den diese Geräte gegenüber stationären PCs und Notebooks haben. Nutzer stellen demnach an mobile Anwendungen die Anforderung, dass diese sich ähnlich bedienen lassen wie stationäre Anwendungen und dass diese auch ähnlich flüssig und schnell arbeiten.

Bei der Realisierung von Anwendungen, die diesen Anforderungen entsprechen, kommt es jedoch zu verschiedenen Konflikten. Die gegenüber stationären PCs veränderte Hardware, beispielsweise geringere CPU Leistung, Hauptspeicher, Display-Auflösung und unterschiedliche Betriebssysteme lassen eine einfache Portierung von Applikationen stationärer PCs oft nicht zu. Für die mobile Nutzung ist eine komplexe Anpassung der Applikation an die jeweiligen Geräte notwendig. Dies führt nicht selten dazu, dass der Funktionsumfang der vormals stationär genutzten Anwendungen auf dem Mobilgerät geringer ausfallen muss. Weiterhin sind neue Applikationen, die die auf Mobilgeräten vorhandenen Sensoren zur Laufzeit auswerten, schwierig zu realisieren, da die Auswertung oft mit komplexen Berechnungen verbunden ist. Diese Berechnungen können aufgrund der geringen Hardware-Ausstattung mobiler Geräte häufig nicht schnell genug ausgeführt werden, um einen nutzbaren Mehrwert für den Anwender zu generieren.

Manche dieser Probleme lassen sich durch Optimierungen, wie die Nutzung von Prozessorinstruktionserweiterungen des Mobilprozessors, direkt lösen, so etwa die Decodierung von HD-Videoströmen. Andere Anwendungen, wie etwa speicherintensive Bildverarbeitungsroutinen können davon nicht profitieren. Manche Anwendungen können jedoch vom Einsatz von *Mobile Cloud Computing* profitieren. Bei der Bereitstellung einer Infrastruktur für *Mobile Cloud Computing* sind verschiedene Aspekte von Bedeutung. Dabei fokussiert sich diese Arbeit auf die Nutzung von weitgehend öffentlich verfügbaren Technologien, um eine breite Anwendbarkeit der Technologie zu erreichen. Will man für eine große Anzahl von MCC-Nutzern entfernt verfügbare Serverressourcen öffentlich zur Verfügung stellen, so sind nachfolgende Aspekte besonders zu betrachten.

Mobile Netzwerkverbindung

Problematisch ist bei *Mobilen Server-unterstützten Anwendungen* zunächst immer die Bewertung der verfügbaren Netzwerkverbindung. Die zuverlässige Kenntnis über die Datenübertragungsgeschwindigkeit und die Größe der zu übertragenden Daten sind notwendig, um die erwarteten Übertragungszeiten abschätzen zu können. Hierbei ist die verfügbare kabellose Internetanbindung der mobilen Geräte besonders zu beachten, da je nach verwendeter Technik und Empfangsstärke die Geschwindigkeit stark variieren kann. Die Übertragungstechnologie selbst hat ebenfalls Einfluss auf die Übertragungsgeschwindigkeit. Zusätzlich beeinflusst die

Codierung der Daten und die Häufigkeit von Sendeoperationen die Performance der Kommunikation. Diese Parameter haben damit auch Einfluss auf den Speedup, der durch die Auslagerung von Berechnungen auf entfernte Server erreicht werden kann. Je kleiner das Verhältnis aus Datenübertragungszeit und Berechnungszeit ist, desto besser eignet sich ein Programm für die Auslagerung. Es ist demnach notwendig, die potenziell geringe verfügbare Bandbreite durch geeignete Kommunikationsmuster und Codierungen bestmöglich auszunutzen. Dabei können aufgrund beispielsweise der höheren Latenz und teilweise fehlenden Duplexfähigkeit nur bedingt Technologien aus der kabelgebundenen Kommunikation übernommen werden.

Auswahl der Cloud-Ressource

Eine weitere Problemstellung ist die situationsabhängige Auswahl einer geeigneten Cloud-Ressource zur Unterstützung der mobilen Applikation. Da Cloud-Anbieter ihre Ressourcen in verschiedenen Qualitätsstufen anbieten, ist es sinnvoll, eine bedarfsgerechte Auswahl zu treffen, die mit den geforderten QoS Eigenschaften wie beispielsweise Verarbeitungszeit in Einklang steht. Die Auswahl hängt zunächst wieder direkt von den Datenübertragungszeiten ab. Es ist beispielsweise unökonomisch, bei geringer Netzwerkperformance eine schnelle und teure Ressource auszuwählen, die dann nur zu einem geringen Prozentsatz ausgelastet werden kann. Hierbei ist zu beachten, dass verschieden schnelle Ressourcen auch verschiedene Kosten haben. Neben der Verarbeitungsgeschwindigkeit und Auslastung einer Ressource spielt also auch die Kostenoptimierung bei der Auswahl eine Rolle. Um die Verarbeitungsgeschwindigkeit abschätzen zu können, ist es notwendig, das Applikationsverhalten (insbesondere die Skalierbarkeit) für eine bestimmte Eingabe auf den zur Verfügung stehenden Cloud-Ressourcen zu kennen. Insbesondere das Herstellen eines Zusammenhangs zwischen einer Eingabegröße und der Verarbeitungszeit ist nicht immer einfach, da eine Abhängigkeitsfunktion gefunden werden muss, von der nicht bekannt ist, ob sie überhaupt existiert.

Verbesserung der Granularität der Abrechnungsperiode

Eine dritte Problemstellung ist die Verbesserung der Granularität bezüglich der Abrechnungsperiode für den Endkunden. *Mobile Cloud-unterstützte Anwendungen* haben oft zeitlich begrenzte Anforderungen an entfernt verfügbare Ressourcen, die je nach Anwendung und Nutzerverhalten zwischen wenigen Sekunden bis Minuten betragen können. Cloud-Anbieter hingegen rechnen die Nutzung im Stundentakt ab. Hier bedarf es einer geeigneten Wiederverwendungsstrategie, um Cloud-Ressourcen zwischen Nutzern zu teilen und möglichst gut auszulasten. Dabei muss jedoch trotzdem sichergestellt werden, dass jeder Nutzer die Performance, für die er zahlt, auch nutzen kann. Um zudem noch die Abrechnung auf feingranularer Basis zu ermöglichen, bedarf es der Entwicklung eines Cloud-Nutzungs-Zyklus. Da Tests mit vielen Nutzern auf einer realen Cloud-Infrastruktur kaum zu finanzieren sind, muss eine Simulation durchgeführt werden, die mit möglichst real gemessenen Daten arbeitet.

Weitere Nebenproblemstellungen

Eine weitere Herausforderung ist der allgemeine Schutz der übertragenen Daten und auch der Schutz von Fremdzugriffen auf die bereitgestellten Ressourcen. Dies ist notwendig, weil jede Nutzung der Infrastruktur Kosten für den Nutzer verursacht. Die Verschlüsselung der Verbindung sowie geeignete Mechanismen zur Authentifizierung sind deshalb obligatorisch. Dies erfordert eine geeignete Modellierung des Lebenszyklus der Verbindung zwischen Nutzer

und der entfernten Ressourcen. Die Bereitstellung von Applikationen für die zu entwerfende Middleware ist ebenfalls Teil der Problemstellung. Hierbei sind geeignete Monitoring- und Debugging-Möglichkeiten zu schaffen, um die Ausführung von Applikationen analysieren und abrechnen zu können.

Zielsetzung der Arbeit ist demnach die Analyse der oben genannten Aspekte bei der Erstellung und Ausführung von mobilen Applikationen aus dem Bereich *Mobile Cloud Computing* und anschließend der Entwurf und die Implementierung einer Middleware, die die Ergebnisse der Analysen beachtet und die geforderten Eigenschaften umsetzt. Diese zu entwickelnde Middleware hat selbst Dienstcharakter und ist auf dem PaaS-Level anzusiedeln. Dem Nutzer soll jedoch im Gegensatz zu bestehenden PaaS-Systemen die Möglichkeit gegeben werden, auf die Qualitätseigenschaften der darunter liegenden IaaS-Schicht Einfluss zu nehmen. Dies wird benötigt, um rechenintensive *Mobile Cloud-unterstützte Anwendungen* realisieren zu können, die dann in der Lage sind, den in 1.2 genannten Anforderungen gerecht zu werden. Insbesondere sollen dazu nur öffentlich verfügbare Technologien betrachtet werden. Dazu zählen *Public Clouds* und das öffentliche Mobilfunknetz ergänzt durch diverse WLAN *HotSpots*. Diese Einschränkung ist zugleich Herausforderung und soll eine breite Anwendbarkeit der Forschungsergebnisse sicherstellen.

Im Gegensatz zu anderen Arbeiten wird in dieser Arbeit die Energieeinsparung durch *Mobile Cloud Computing* nicht thematisiert, da die Verbesserung des Nutzererlebnisses (englisch *User Experience*) unter Beachtung der Kosten im Vordergrund steht. Dazu ist es oft notwendig, die verfügbaren Ressourcen (Cloud-Ressourcen und mobiles Endgerät) auch auszulasten.

1.3 Aufbau der Arbeit

Diese Arbeit nähert sich der zentralen Fragestellung unter Beachtung verwandter Arbeiten. Dazu wird in Kapitel 2 zunächst ein Überblick über die einschlägige Literatur zu den Themen *Cloud Computing*, *Mobile Computing* und anderen gegeben. Nach der Einordnung der Arbeit in die aktuelle Forschungslandschaft werden anschließend theoretische Grundlagen zu einer Theorie zusammengefügt, die durch eigene empirische Beobachtungen verfeinert und erweitert wird. Das resultierende Modell, welches in Kapitel 3 vorgestellt wird, ist in der Lage, *Mobile Server-unterstützte Anwendungen* abzubilden. Das Modell wird dabei aus Anforderungen und Erfahrungen generiert, die in verwandten und eigenen Arbeiten zu finden sind. Dazu werden Anwendungen und deren Charakteristik, existierende Middlewares und deren Architektur und weitere Technologien aus dem Bereich *Public Computing* analysiert.

Weitere Aspekte, die insbesondere für MCC von Bedeutung sind, werden in speziellen Kapiteln untersucht. Dazu zählt die theoretische und praktische Untersuchung von mobilen Netzwerkverbindungen in Kapitel 4. Die Leistungsfähigkeit von UMTS und WLAN-Verbindungen wird unter Verwendung verschiedener Mobilgeräte betrachtet. Erfasst werden Parameter wie Latenz, Bandbreite und Duplexfähigkeit. Zusätzlich wird der Einfluss von Verschlüsselung, Kommunikationsfrequenz und Datencodierung untersucht. Diese Untersuchung mündet in die Umsetzung einer effizienten Kommunikationsbibliothek, die die Eigenschaften mobiler Netzwerkverbindungen bestmöglich unterstützt und für Anwendungen eine zuverlässige Kommunikationsschnittstelle bereitstellt.

In Kapitel 5 wird untersucht, wie eine Architektur zur Bereitstellung feingranularer und konfigurierbarer Cloud-Ressourcen zur Unterstützung von mobilen Anwendungen aussehen

kann. Dazu zählt auch die Modellierung des Zugriffs auf solche Ressourcen. Um die Eigenschaften und die Leistungsfähigkeit einer solchen Architektur auf Basis öffentlicher Cloud-Anbieter zu untersuchen, wird eine Simulation durchgeführt. Dazu werden zunächst relevante Simulationsparameter und Simulationseingaben ermittelt. Diese sollen direkt aus realen MCC-Anwendungen und realer Infrastruktur abgeleitet sein und müssen dazu entsprechend gemessen werden. Ziel der Simulation ist der Vergleich verschiedener Algorithmen zur Allokation und Lastbalancierung von Cloud-Ressourcen.

Kapitel 6 untersucht den Aspekt der bedarfsgerechten Auswahl von verfügbaren Cloud-Ressourcen zur Ausführungszeit des Programms. Es werden geeignete Parameter ermittelt, die zunächst die aktuelle Situation des Nutzers bezüglich der verfügbaren Netzwerkperformance und der Geschwindigkeit des Mobilgeräts bestimmen. Zusätzlich werden weitere Parameter definiert, die eine Prognose über die zu erwartende Ausführungszeit bei Ausführung auf einer entfernten Cloud-Ressource ermöglichen. Aufbauend auf diesen Parametern wird dann eine Schnittstelle entwickelt, die es ermöglicht, Applikationen mit spezifischen Eigenschaften bereitzustellen und darüber hinaus in Abhängigkeit der Laufzeitparameter eine Auswahl möglicher und sinnvoller Ausführungsoptionen zu präsentieren. Zur Auswahl werden geeignete Algorithmen entwickelt.

Die Ergebnisse der in Kapitel 3–6 durchgeführten Untersuchungen münden dann in Kapitel 7 in den Entwurf einer Middleware, die geeignete Mechanismen zur Kommunikation, Allokation von Ressourcen und deren bedarfsgerechte Auswahl unterstützt. Dazu werden die Middleware-Komponenten und die Programmierschnittstelle (*Application Programming Interface* (API)) der Middleware für *Java* vorgestellt. Dieses Kapitel beschäftigt sich auch mit der Erstellung von Applikationen auf Basis der API und stellt die Möglichkeiten der Laufzeitumgebung bezüglich Debugging und Monitoring von Anwendungen vor.

Zur Demonstration der Anwendbarkeit der Middleware werden in Kapitel 8 Applikationen vorgestellt, die mit Hilfe der Middleware realisiert sind. Dies dient auch dem Zweck der Bewertung der Leistungsfähigkeit der Middleware. Dabei werden synthetische Parameter untersucht, wie etwa die Dauer der Allokation von Ressourcen. Außerdem werden auch Applikationen aus dem Open Source Bereich unter Verwendung der Middleware für mobile Geräte portiert. Die Transformation und Analyse wird ebenfalls beschrieben. Anschließend wird die Leistungsfähigkeit der Auswahlstrategie für Cloud-Ressourcen für die einzelnen Applikationen gemessen. Dazu werden die geschätzten und die real gemessenen Ausführungszeiten verglichen.

Kapitel 9 fasst die Arbeit zusammen und resümiert die wichtigsten Ergebnisse der Untersuchungen. Außerdem werden Anknüpfungs- und Erweiterungspunkte der Arbeit diskutiert. Dieses Kapitel mündet in einer Handlungsempfehlung zur Erstellung von MCC-Anwendungen unten den Einschränkungen wie sie in dieser Arbeit untersucht wurden.

Die für die Untersuchungen in dieser Arbeit erstellten *Java*-Programme wurden überwiegend unter *Linux Mint*[7] entwickelt. Dazu wurde Das *Java Development Kit* (JDK)[8] der

[7] *„The purpose of Linux Mint is to produce a modern, elegant and comfortable operating system which is both powerful and easy to use. Started in 2006, Linux Mint is now the 4th most widely used home operating system behind Microsoft Windows, Apple Mac OS and Canonical's Ubuntu. [...] Based on Debian and Ubuntu, it provides about 30,000 packages and one of the best software managers.“* (Onlineressource `http://www.linuxmint.com/about.php` (abgerufen am 16.4.2014))

[8] *„Mit dem Java Development Kit (JDK) lassen sich Java SE-Applikationen entwickeln. Dem JDK sind Hilfsprogramme beigelegt, die für die Java-Entwicklung nötig sind. Dazu zählen der essenzielle Compiler, aber auch andere Hilfsprogramme, etwa zur Signierung von Java-Archiven oder zum Start einer Management-*

OpenJDK-Implementierung von *Java* verwendet, deren Eigenschaften in Kapitel 6.3.3 genauer betrachtet werden. Als Entwicklungsumgebung kam *Eclipse*[9] zum Einsatz. Für die erstellten *Android*-Programme kam ebenfalls die Entwicklungsumgebung *Eclipse* zum Einsatz, welche für diesen Zweck um das Plugin *Android Development Tools* (ADT) erweitert wurde. Zur Entwicklung von *Android*-Programmen war zusätzlich das *Android Software Development Kit* (SDK)[10] notwendig, welches unter anderem den Compiler für *Android*-Programme enthält.

Konsole. In den Versionen Java 1.2, 1.3 und 1.4 heißt das JDK Java 2 Software Development Kit (J2SDK), kurz SDK, ab Java 5 heißt es wieder JDK.“ (siehe [Ull12, S.78])

[9] *„Eclipse (von englisch eclipse ‚Sonnenfinsternis' ‚Finsternis' ‚Verdunkelung') ist ein quelloffenes Programmierwerkzeug zur Entwicklung von Software verschiedenster Art. Ursprünglich wurde Eclipse als integrierte Entwicklungsumgebung (IDE) für die Programmiersprache Java genutzt, aber mittlerweile wird es wegen seiner Erweiterbarkeit auch für viele andere Entwicklungsaufgaben eingesetzt. Für Eclipse gibt es eine Vielzahl sowohl quelloffener als auch kommerzieller Erweiterungen.“* (Onlineressource `http://de.wikipedia.org/wiki/Eclipse_(IDE)` (abgerufen am 16.5.2014)) Eclipse ist verfügbar unter Onlineressource `http://www.eclipse.org/` (abgerufen am 16.5.2014)

[10] *„The Android SDK provides you the API libraries and developer tools necessary to build, test, and debug apps for Android. If you're a new Android developer, we recommend you download the ADT Bundle to quickly start developing apps. It includes the essential Android SDK components and a version of the Eclipse IDE with built-in ADT (Android Developer Tools) to streamline your Android app development.“* (Onlineressource `http://developer.android.com/sdk/index.html` (abgerufen am 16.5.2014))

2 Verwandte Arbeiten

Im Bereich *Mobile Cloud-unterstützte Anwendungen* gibt es eine Vielzahl von Überschneidungen mit den Bereichen Mobilität, Client/Server, Services Computing, Ressourcenmanagement, und mehr. In diesem Kapitel werden wichtige verwandte Arbeiten vorgestellt. Zunächst erfolgt eine Charakterisierung von Anwendungen aus dem Bereich MCC, anhand derer dann eine bessere Einordnung des behandelten Problems erfolgen kann. Auf Basis der Anwendungscharakteristik werden dann bereits verfügbare und verwandte Technologien vorgestellt und deren Eignung und Beschränkungen für *Mobile Cloud-unterstützte Anwendungen* erörtert. Bezüglich der Problemstellung liegt dabei besonderes Augenmerk auf der Beachtung der speziellen Eigenschaften von mobilen Kommunikationsverbindungen, der Granularität der benötigten Serverunterstützung und der bedarfsgerechten Auswahl der Serverressource.

2.1 MCC-Anwendungen und deren Charakteristik

Aus dem Studium einschlägiger Literatur zum Thema *Mobile (Cloud) Computing* geht hervor, dass vor allem Anwendungen aus den Bereichen Bildverarbeitung, Computerspiele und Internetanwendungen wie E-Mail oder Web eine große Rolle für den Endbenutzer spielen (siehe Tabelle 2.1). Dabei zählen E-Mail und Web üblicherweise nicht zu den berechnungsintensiven Anwendungen und sind somit für unsere Betrachtungen von untergeordnetem Interesse.

2.1.1 Bildverarbeitung

Mobile Bildverarbeitung ist ein Beispiel, wofür man die in mobilen Endgeräten eingebauten Sensoren nutzen kann. Damit können dem Nutzer neuartige Applikationen bereitgestellt werden, die auf einem stationären PC nicht realisierbar wären. Hierbei sind vor allem Applikationen der Mustererkennung, Bildbearbeitung oder *Augmented Reality* (AR) von Bedeutung [CLK+11, CBC+10, KPKB10, MPPS+12]. Diesen Anwendungen ist üblicherweise gemein, dass sie in mindestens eine Richtung größere binäre Datenmengen in Form von Video oder Bilddaten übertragen. Dabei sind die Anwendungen jedoch oft nicht sehr zeitkritisch. Das heißt, dass etwaige größere Latenzzeiten durch die mobile Datenverbindung durch Pufferung (beispielsweise bei Video) ausgeglichen werden können. Eine Ausnahme bilden hier Anwendungen von Augmented Reality. Da hierbei aufgenommene Kamerabilder in Echtzeit mit Zusatzinformationen aufgewertet werden, ist eine schnelle Antwortzeit des Servers Voraussetzung für das flüssige Bedienen einer solchen Anwendung. Anwendungen aus dem Bereich Bildverarbeitung benötigen außerdem eine recht hohe Übertragungsbandbreite, die mindestens 1 Mbit/s betragen sollte, um brauchbare Ergebnisse zu gewährleisten. Auf Serverseite benötigen diese Anwendungen meistens erhöhte Rechenkapazität. Aus [CBC+10] geht beispielsweise hervor, dass ein Gesichtserkennungsalgorithmus Daten im 10-100 kB Bereich zwischen Server und Client transferiert, und dass eine Client/Server-Variante der Gesichtserkennung 2-3 s benötigt. Die Bildverarbeitungsapplikationen können sowohl in 3G und WLAN-Netzwerken benutzt werden. Bildverarbeitungsanwendungen sind meist von kurzer

Nutzungsdauer geprägt. Bei Augmented Reality Applikationen beträgt die Nutzungsdauer bis zu einigen Minuten [KPKB10].

2.1.2 Mobile Gaming

Im Bereich Mobile Gaming ist zwischen zwei grundsätzlichen Richtungen zu unterscheiden. Einerseits wird hier das sogenannte Stream-Gaming oder auch Cloud-Gaming angewendet, bei dem die Spiele nicht mehr lokal auf dem Gerät, sondern auf entfernten Servern ausgeführt werden. Die Bildschirmausgabe wird per Stream übertragen. Andererseits gibt es auch eine Gruppe von lokal ausgeführten Spielen, bei der per Netzwerk verbundene Spieler gegeneinander/miteinander spielen. Im Vergleich zum Mobile Video Streaming, welches durch Pufferung eine schlechte Netzwerkperformance ausgleichen kann, ist beim Cloud-Gaming vor allem die Netzwerklatenz entscheidend, um ein flüssiges Spielerlebnis und eine adäquate Reaktionszeit zu garantieren. Die Anforderungen an die Übertragungsbandbreite können dabei sehr unterschiedlich sein und zwischen wenigen kbit/s und einigen Mbit/s für Cloud-Gaming liegen. Ebenso divergieren die Anforderungen auf Serverseite stark.

Die in [CLK+11, CBC+10, KPKB10] vorgestellten Spiele gehören zur Gruppe der lokal ausgeführten Server-unterstützten Spiele. In [KPKB10] wird beispielsweise das AR-Spiel *PhotoShoot* vorgstellt, bei dem sich zwei Spieler via Smartphonekamera gegenseitig in einem Fadenkreuz anvisieren und virtuell aufeinander schießen können. Die Software erkennt dabei Treffer automatisch. Dabei wird ein Gesichtserkennungsalgorithmus auf einem entfernten Server ausgeführt, da er lokal zu viele Ressourcen und Zeit benötigen würde. Mobile Gaming benötigt die entfernten Server-Ressourcen je nach Anforderung des Benutzers zwischen wenigen Sekunden bis zu mehreren Minuten.

2.1.3 Desktop Anwendungen

Weitere Anwendung von MCC ist die Bereitstellung von existierenden Desktop-Anwendungen für Mobilgeräte. Dabei werden die Desktop-Anwendungen manchmal nicht portiert, sondern entfernt ausgeführt und nur die Bildschirmausgabe und Interaktionen werden übertragen [SBCD09]. Die Applikation kann auch in einen lokalen und einen entfernt ausgeführten Teil partitioniert werden [CIM+11]. Diese Anwendung ähnelt dem Cloud-Gaming, stellt jedoch aufgrund der geringeren Bildschirmaktivität oftmals nicht so hohe Anforderungen an die verfügbare Übertragungsbandbreite. Im Fokus dieser Technologien stehen nicht nur die schnellere Ausführung von Applikationen, sondern oft auch die Bereitstellung von großen Speichermengen, sowohl Arbeitsspeicher als auch persistenter Speicher, was Mobilgeräte nicht leisten können. Die Nutzungsdauer dieser Applikationen liegt im Minutenbereich bis hin zu einigen Stunden. Als Beispiele für mobil genutzte Desktop-Anwendungen sind in der Literatur beispielsweise Büroprogramme [SBCD09] oder ein Virus-Scanner [CIM+11] angegeben.

2.1.4 Zusammenfassung Anwendungscharakteristik

Die betrachteten Applikationen unterscheiden sich in der Menge der übertragenen Daten, der Nutzungsdauer und der benötigten Ressourcen. Für die meisten Anwendungen sind jedoch bidirektionale Kommunikation, kurze Reaktionszeit oder eine relativ hohe Bandbreite sehr nützlich, um den Overhead für die entfernte Ausführung gering zu halten. Hier wird erneut die Diskrepanz zwischen der technologischen Verfügbarkeit und den Anforderungen, die Nutzer

Tabelle 2.1: Eigenschaften verschiedener Frameworks für Mobile (Cloud) Computing

Name	Technologie	Intention	Security	Applikationen	Netzwerk	Evaluation
Cloudlets [SBCD09]	• Dienst wird auf einer VM in der Nähe ausgeführt • Sprachunabhängig	• Realisierung ressourcen-hungriger Anwendungen	• SSH • SASL	• Büroprogramme • Gimp • andere	• WLAN	• Benchmarks zur Bereitstellungszeit der Cloudlets
CloneCloud [CIM+11]	• Application Level VM Clone • Java/Dalvik/.NET • Android	• Beschleunigung • Energieeinsparung	• möglich • nicht implementiert	• Virus-Scanner • Bildersuche • Verhaltens-beobachtung	• WLAN • 3G[3]	• für genannte Apps: Ausführungszeit und Energieaufnahme
Maui [CBC+10]	• Microsoft .NET • Windows Mobile	• Energieeinsparung	• k.A.[1]	• Gesichtserkennung • Arcade Spiel • Sprach-Übersetzer	• WLAN • 3G[3]	• Energieeinsparung und Speedup für Gesichtserkennung, Videospiel und Schachspiel
Mars [CLK+11]	• RPC Mechanismus • Laufzeitsystem zur Optimierung der RPC-Aufruf-Entscheidungen • keine konkrete Technologie genannt, vermutlich Linux nach den Testgeräten zu urteilen	• Beschleunigung • Energieeinsparung	• k.A.[1]	• Gesichtserkennung • AR[2] • AR Videospiel	• WLAN • 3G[3]	• Messungen mit Smartphone und Tablet • WLAN und 3G[3] • Vergleich mit simulierten statischen RPC-Aufrufen
Cuckoo [KPKB10]	• Java • Android • IBIS middleware	• Beschleunigung	• nicht implementiert • zukünftige Arbeit	• Objekterkennung • verteiltes AR[2] Spiel	• WLAN • andere	• Applikationsbench-marks der Objekterkennung mit WLAN
Tasklets [MPPS+12]	• entfernte Ausführung von kleinen abgeschlossenen Berechnungseinheiten (Tasklets)	• Beschleunigung	• k.A.[1]	• Bildeverarbeitung • Spracherkennung	• k.A.[1]	• k.A.[1]

1 (k.A.) = keine Angabe
2 (AR) = Augmented Reality
3 (3G) = Mobilfunknetzwerk der 3. Generation (beispielsweise UMTS und HSPA)

an mobile Applikationen stellen, deutlich. Zudem ist eine Anwendung in ubiquitär verfügbaren Mobilfunknetzwerken nicht immer möglich. Die übertragenen Datenmenge liegen im kB bis MB Bereich. Außerdem ist die Leistungsfähigkeit der benötigten Serverressourcen als moderat einzuschätzen. Somit können prinzipiell auch Cloud-Ressourcen zum Einsatz kommen. Teure Servermaschinen aus dem Höchstleistungsrechnen sind nicht erforderlich. Die Performance der Server sollte jedoch relativ konstant sein. Die Nutzungsdauer der meisten Applikationen liegt im Minutenbereich.

2.2 Realisierung von Mobile (Cloud) Computing Anwendungen

Aus technischer Sicht folgen die verschiedenen Frameworks und Middlewares für MCC-Anwendungen unterschiedlichen Ansätzen. Eine mögliche Form der Unterscheidung kann bezüglich der Art der Auslagerung von berechnungsintensiven Anwendungsteilen vorgenommen werden. Hierbei sind in der Literatur Ansätze zu finden, die ohne Modifizierung des Programm- und Quellcodes des eigentlichen Programms auskommen. Es gibt aber auch Ansätze, die eine explizite Partitionierung des Programms durch den Programmierer vorsehen. Die einzelnen Verfahren sind in Tabelle 2.1 zusammengefasst.

2.2.1 Verfahren ohne explizite Anpassung des Programmcodes

Es gibt die Möglichkeit, eine Applikation direkt „in der Cloud“ auszuführen und nur die Bildschirmausgabe auf das Mobilgerät zu übertragen. Dieses *Cloudlets* genannte Verfahren wird in [SBCD09] vorgeschlagen. Hierbei läuft das Programm selbst komplett auf einer VM und nur die Benutzereingaben und die Bildschirmausgabe werden übertragen. Dies ermöglicht auch sehr komplexe und speicherintensive Programme. Da jedoch relativ viele Daten übertragen werden müssen, ist eine WLAN-Verbindung notwendig. Darüber hinaus ist hierbei eventuell auch der Datenaustausch zwischen Mobilgerät und Server schwierig und müsste über ein separates gemeinsames Laufwerk wie beispielsweise *Dropbox* realisiert werden. Zur Verschlüsselung setzt *Cloudlets SSH*[1] und *SASL*[2] ein.

Bei dem in [CIM+11] vorgestellten *CloneCloud* wird eine Client-Instanz und eine Server-Instanz des gleichen Programms simultan ausgeführt. Dabei werden wie in [CBC+10] nur Programme unterstützt, die in einer *Application Level VM* (beispielsweise *.NET*[3] oder *Java-VM*[4]) ausgeführt werden. Die in [CIM+11] vorgestellte Implementierung von *CloneCloud*

[1] *„Die Secure Shell (SSH) bietet eine sichere Möglichkeit, um sich in einen entfernten Rechner einzuloggen und diesen mittels Befehlszeile zu steuern bzw. Dateien auszutauschen. Dazu stellt der anfragende SSH-Client zuallererst die Identität des entfernten Rechners sicher. Erst danach findet eine Authentifizierung des Benutzers statt. Die Datenübertragung zum entfernten Rechner läuft gesichert (verschlüsselt und integritätsgesichert) ab.“* [BMB+05, S.400ff.]

[2] *„Das Simple Authentication and Security Layer Rahmenwerk (SASL) stellt Mechanismen bereit, die eine einfache Integration von Authentifizierungsverfahren in verbindungsorientierte Kommunikationsprotokolle ermöglichen.“* [BMB+05, S.428ff.]

[3] *„.NET bezeichnet eine von Microsoft entwickelte Software-Plattform zur Entwicklung und Ausführung von Anwendungsprogrammen. .NET besteht aus einer Laufzeitumgebung (Common Language Runtime), in der die Programme ausgeführt werden, sowie einer Sammlung von Klassenbibliotheken, Programmierschnittstellen und Dienstprogrammen (Services). .NET ist auf verschiedenen Plattformen verfügbar und unterstützt die Verwendung einer Vielzahl von Programmiersprachen. .NET-Programme werden zunächst in eine Zwischensprache (Common Intermediate Language) übersetzt, bevor sie von der Laufzeitumgebung ausgeführt werden.“* (Onlineressource `http://de.wikipedia.org/wiki/.NET`(abgerufen am 2.11.2013))

[4] *„Die Java Virtual Machine (abgekürzt Java-VM oder JVM) ist der Teil der Java-Laufzeitumgebung (JRE)*

benutzt *Java* und ist für eine speziell angepasste *Android*-Version für Mobilgeräte verfügbar. Die Server-Instanz des Programms residiert dabei ebenfalls in einer *Application Level VM*. In regelmäßigen Abständen werden zudem deren Zustände synchronisiert. Bei berechnungsintensiven Aufgaben ist die Server-Instanz schneller fertig und synchronisiert den neuen Zustand, bevor die Client-Instanz den neuen Zustand selbst errechnet hätte. Dadurch lassen sich Ausführungszeit und Energieaufnahme des Mobilgeräts verringern. Es ist nicht bekannt, welche Kommunikationstechnologie zur Synchronisation verwendet wurde. Bei dieser Variante, wie auch bei den in [SBCD09] vorgestellten *Cloudlets*, ist eine Server-Instanz dediziert einem einzigen Benutzer zugewiesen. Multiplexing von Anfragen auf eine VM und die Wiederverwendung der VM sind ausgeschlossen. Für jeden Benutzer und jedes Programm wird eine neue VM-Instanz gestartet.

In [CBC+10] wird eine Technik zur dynamischen Partitionierung der Applikation zur Laufzeit, genannt *Maui*, vorgeschlagen. Dazu wird eine Laufzeitanalyse des Programms durchgeführt und potenziell rechenintensive Teile werden erkannt und unter Berücksichtigung der aktuell verfügbaren Netzwerkperformance auf einen Server ausgelagert. Dabei wird der serverseitig auszuführende Quellcode erst zur Laufzeit übertragen. Hierbei spricht man vom Code-Offloading. Vorteil ist hierbei auch, dass die universellen Server verschiedene Anwendungen auch gleichzeitig beherbergen können. Damit die Laufzeitanalyse auch mit vorkompilierten Programmen funktioniert, können nur Programmiersprachen zum Einsatz kommen, die eine *Application Level VM* zur Code-Interpretation einsetzen, da die VM auch Selbstüberwachungsfunktionen (Monitoring) bereitstellt. In [CBC+10] wird .NET verwendet. Für die Datenübertragung wird die spezifische effiziente Bibliotheksimplementierung *.NET Remoting* benutzt. *.NET Remoting* unterstützt, wie auch *Java*, Objektserialisierung. Dabei werden Objekte aus dem Speicher der *Application Level VM* direkt via Netzwerk versendet, sodass der Objektzustand auf der Gegenseite erhalten bleibt und das Objekt dort als Kopie erscheint. Leider macht die zugehörige Literatur keine Angaben über die Sicherheitsmechanismen, die bei der Datenübertragung verwendet wurden.

2.2.2 Verfahren mit expliziter Client/Server Programmierung

Teilt man eine Applikation in eine Server-Komponente und eine Client-Komponente, so spricht man von *remote processing*. Dabei werden Teile der Applikation beispielsweise durch einen entfernten Prozeduraufruf (englisch *Remote Procedure Call* (RPC)) vom Client aufgerufen. Eine Applikation zu partitionieren ist jedoch im Allgemeinen ein schwieriger Vorgang. Das liegt darin begründet, dass einzelne Applikationsteile Abhängigkeiten zueinander haben können, die eine Aufteilung in getrennte Teile erschweren oder verhindern. Oft werden Graphalgorithmen, vor allem zum Lösen des MINCUT Problems [GJ79], benutzt, um diese Abhängigkeitsgraphen effizient zu partitionieren. Solche Algorithmen sind oft NP-vollständig und somit nicht in polynomieller Laufzeit zu lösen. Deshalb wird die Partitionierung oft bereits zur Übersetzungszeit durchgeführt. Diesen Ansatz verfolgen [CLK+11] und [KPKB10].

In [CLK+11] werden Programmfragmente, die entfernt ausgeführt werden sollen, vor dem

für Java-Programme, der für die Ausführung des Java-Bytecodes verantwortlich ist .Der andere Teil der Java-Laufzeitumgebung sind die Java-Klassenbibliotheken. [...] Die Java Virtual Machine ist dazu da, den vom Java-Compiler erzeugten (plattformunabhängigen) Bytecode plattformabhängig auszuführen. Dazu werden die erzeugten Bytecode-Dateien (Dateiendung „.class“) während der Laufzeit in die lokale Maschinensprache übersetzt.“ (Onlineressource `http://de.wikipedia.org/wiki/Java_Virtual_Machine`(abgerufen am 2.11.2013))

Kompilieren durch den Programmierer markiert. Zur Laufzeit wird dann entschieden, welche Programmfragmente entfernt ausgeführt werden, um einen guten Performancegewinn und eine möglichst hohe Energieeinsparung zu erreichen. Dabei wird die Technik des entfernten Prozeduraufrufs angewendet. Es wird davon ausgegangen, dass die via RPC verfügbaren Methoden auf Serverseite bereits vorhanden sind und nicht erst zur Laufzeit dort installiert werden. Das in [CLK+11] vorgestellte *Mars* wurde für Linux-Geräte mit ARM-Prozessor getestet. Über die verwendeten Programmiersprachen und Kommunikationstechnologien ist nichts bekannt.

In [KPKB10] wird *Cuckoo* vorgestellt, ein Offloading Framework für *Android*-Applikationen, welches auf der *IBIS* Middleware basiert [PKKB09]. Vom Programmierer werden Funktionen spezifiziert, die entfernt ausgeführt werden können. *Cuckoo* entscheidet dann zur Laufzeit, ob lokal oder entfernt ausgeführt wird. Dazu besitzt die Middleware auch eine Management-Komponente, die aus einer Menge von verfügbaren Compute-Ressourcen auswählen kann. Die Ressourcen müssen der Middleware jedoch explizit bekannt gemacht werden. Eine zentrale Registry existiert nicht. *Java*-Code kann zur Laufzeit auf einer Compute-Ressource installiert werden. Dafür bietet die Ressource einen speziellen Port an. Zur Kommunikation wird die aus dem wissenschaftlichen Rechnen bekannte effiziente Kommunikationsbibliothek *Smart-Sockets* von *IBIS* benutzt, welche die effiziente Nutzung der verfügbaren Netzwerkbandbreite erleichtert [PKKB09]. Das Verzichten auf Sicherheitsmechanismen wird explizit genannt und deren Integration wird als zukünftige Arbeit angegeben.

2.2.3 Zusammenfassung Realisierung von MCC-Anwendungen

Eine Laufzeitanalyse und Partitionierung zur Laufzeit ermöglichen zwar die verteilte Ausführung von unmodifizierten Programmen, jedoch erfordern sie auch eine entsprechend angepasste Laufzeitumgebung. Dies schränkt vor allem seitens des mobilen Endgerätes die Anwendbarkeit dieses Ansatzes stark ein, da man dort kaum Möglichkeiten hat, die Softwareinstallation anzupassen. Um eine breite Anwendbarkeit sicherzustellen, ist eine vorpartitionierte Variante besser, die durch Bibliotheksunterstützung dann eingeschränkte Entscheidungen zur Laufzeit treffen kann.

Existierende Ansätze zur Auslagerung beschränken sich auf Betrachtungen zum Speedup, zur möglichen Energieeinsparung auf dem mobilen Gerät und auf Möglichkeiten der Entscheidungsfindung, wann ausgelagert werden soll und wann nicht. Üblicherweise wird dabei davon ausgegangen, dass für jeden Nutzer eine dedizierte Servermaschine bereitsteht. Diese Annahme ist jedoch mit vielen Nutzern kaum mehr zu rechtfertigen. Nur in [MPPS+12] wird die Allokation von Ressourcen auf öffentlich verfügbarer Infrastruktur in Betracht gezogen. Das darin vorgeschlagene Konzept der *Tasklets* ist jedoch nur knapp beschrieben. Konkrete Mechanismen und Zielumgebungen sind nicht genannt, weshalb dieses Konzept den beschriebenen Kategorien nicht zugeordnet werden konnte.

Manche der vorgestellten Technologien sind nur für WLAN praktikabel. Andere unterstützen auch 3G oder andere Kommunikationstechnologien (siehe Tabelle 2.1). Manche Verfahren realisieren die entfernte Ausführung des Programmcodes in Abhängigkeit der verfügbaren Netzwerkperformance. Es wird jedoch bei keinem Verfahren darauf eingegangen, wie die Netzwerkperformance bestimmt wurde und wie die Entscheidungsfindung im Detail implementiert ist.

2.3 MCC aus Sicht der Cloud-Infrastruktur

Die von den vorgestellten Technologien genutzten Server-Ressourcen müssen zunächst bereitgestellt werden und den einzelnen Clients zur Laufzeit auch zugewiesen werden. Man spricht von Ressourcenallokation. Hierbei gibt es zentral angelegte Middlewares, die einen einzelnen Front-Server benutzen, um Ressourcen für die Clients zu allokieren. Es gibt jedoch auch dezentrale Verfahren, bei denen die Clients selbst versuchen, Ressourcen in ihrer Umgebung zu allokieren.

2.3.1 MCC-Architekturen

In der Literatur finden sich von abstrakten Beschreibungen ganzer Paradigmen über Architekturen mit einer konkreten Implementierung bis hin zu speziellen Cloud-basierten Middlewares die verschiedensten Ansätze, um Cloud-basierte Compute-Dienste auf feingranularer Basis bereitzustellen. MCC-Architekturen sind im Allgemeinen zweischichtige Architekturen bestehend aus Client und Compute-Server. Dagegen ist die typische Web-Architektur dreischichtig, bestehend aus Client, Webserver und Datenbank. Die Datenhaltungskomponente spielte im MCC nur eine untergeordnete Rolle, da die zu verarbeitenden Daten ohnehin erst vom Client auf den Compute-Server übertragen werden müssen.

In [FE10] wird *Cloud@Home* vorgestellt, eine neuartige Anwendung des *@Home*-Paradigmas aus dem *Volunteer Computing* auf Technologien des *Cloud Computings*. Ziel dabei ist es, konfigurierbare Compute-Ressourcen auf feingranularer Basis zur Verfügung zu stellen. Dabei werden explizit auch mobile Einsatzzwecke genannt. Beim *Volunteer Computing* können Nutzer eigene ungenutzte Rechenkapazitäten für eine rechenintensive Aufgabe zur Verfügung stellen. Ein sehr bekanntes Projekt aus diesem Bereich ist beispielsweise *SETI@home*[5].

> „*The Cloud@Home motto is: heterogeneous hardware for homogeneous Clouds.*“[FE10, S.581]

Um dies zu ermöglichen, ist *Cloud@Home* in die Schichten *Frontend Layer*, *Virtual Layer* und *Physical Layer* eingeteilt. Dabei vermittelt und verwaltet der *Frontend Layer* Ressourcen aus dem *Virtual Layer*. Zur Verfügung stehen die virtuellen Ressourcen *storage* und *execution*. Das *@Home* bedeutet hier, dass beliebige Maschinen, so auch Desktop-Computer, in den *Physical Layer* eingebunden werden können. Es wird jedoch nur ein abstraktes Konzept vorgestellt, das keine Aussagen darüber macht, ob beliebige VM Images (beispielsweise 32bit und 64bit) auf allen Ressourcen lauffähig sind oder ob die *Resource Engine* für VM Images eine Vorauswahl an Anbietern trifft, auf denen die Images tatsächlich lauffähig sind. Obwohl das System auf *Volunteer Computing* aufsetzt, ist auffällig, dass keine Behandlung von Firewall-Problemen, wie sie im privaten Bereich durch NAT-*Router* oder Betriebssystem-Firewalls häufig auftreten, Bedeutung beigemessen wird.

Zentrale Verfahren zur Ressourcenallokation setzen einen Front-Server oder Broker-Server ein. Dies hat den Nachteil, dass es einen Single Point of Failure gibt, der bei Ausfall das ge-

[5] *„SETI@home ist ein wissenschaftliches Experiment, welches mit dem Internet verbundene Computer für die Suche nach außerirdischer Intelligenz einsetzt. SETI (Search for Extraterrestrial Intelligence) ist ein wissenschaftliches Aufgabenfeld mit dem Ziel außerirdisches Leben zu finden. Eine Herangehensweise, bekannt als radio SETI, nutzt Radioteleskope, um nach schmalbandigen Funksignalen aus dem Weltraum zu suchen.“* (Onlineressource `http://setiathome.berkeley.edu/sah_about.php` (abgerufen am 2.11.2013))

samte System unbenutzbar macht. Ein Ausfall kann beispielsweise durch eine *DoS* Attacke[6] ausgelöst werden. Des Weiteren liegen Cloud-Rechenzentrum und Client eventuell weit entfernt voneinander, was zu hohen Netzwerklatenzen führen kann. Um genau diese hohen Netzwerklatenzen zu vermeiden und um das System robuster gegen Angriffe zu machen, wurden auch dezentrale Technologien erforscht, die eine Compute-Ressource im lokalen Umfeld eines Client benutzen. Dabei befindet sich in [SF05] die Ressource direkt am WLAN *Accesspoint*. Ein sogenannter *home server*, der für den Client immer erreichbar ist, migriert zur Laufzeit eine VM-Instanz auf einen lokal verfügbaren Server. Eine Spracherkennungs- und eine Remote-Desktop-Software konnten durch dieses System deutlich beschleunigt werden. Hierbei kann zwar hohe Netzwerkbandbreite sichergestellt werden, jedoch ist diese Technologie beispielsweise nicht auf 3G Netzwerke anwendbar. Auch Authentifizierung und Abrechnung sind in dezentralen Netzwerken mit hohem Aufwand verbunden. Zudem ist die Verfügbarkeit bestimmter Ressourcen nicht so gut sichergestellt wie bei einem zentralisierten System.

Eine zentralisierte Middleware, die innerhalb einer Cloud-Infrastruktur, beispielsweise *Amazon EC2*, ausgeführt wird, bietet den Vorteil, dass sie sehr gut verfügbare und durch entsprechende horizontale Skalierung genügend Ressourcen vorhalten kann, um Benutzeranfragen auf gleichbleibendem Niveau zu befriedigen. Ein solches System wird in [MPPS+12] vorgeschlagen. Zudem liegen die verfügbaren Ressourcen eines IaaS-Anbieters üblicherweise in definierten Qualitätsstufen vor. Zur Allokation der Ressourcen wird ein Broker eingesetzt, der Ressourcen an anfragende Clients vermittelt. Zudem kann über diesen Broker auch die Authentifizierung und Abrechnung der Nutzung erfolgen. In [MPPS+12] wird *Vision* vorgestellt, eine Ausführungsplattform für Tasklets, die zwischen PaaS- und IaaS-Schicht angesiedelt ist. Die vorgestellte Middleware behandelt ein Tasklet als abgeschlossene Berechnungseinheit, die durch einen sogenannten *Tasklet Trading Service* (TTS) auf eine passende Ressource zugewiesen werden kann. Dieser Trading Service ist ähnlich dem *CORBA Trading Service*, unterscheidet sich von diesem jedoch dadurch, dass der (gehandelte) Dienst generisch ist und beliebige Tasklets ausführen kann, also nur Compute-Ressourcen anbietet. *Vision* ist der in dieser Arbeit vorgestellten Middleware hinsichtlich der angestrebten Ziele ähnlich (siehe Kapitel 1.2). Weitere Informationen zu *Vision* konnten jedoch aus der darüber verfügbaren Literatur nicht beschafft werden. Dies betrifft insbesondere Angaben zu den Anwendungsszenarien und Implementierungstechnologien. Somit konnte auch keine vergleichende Bewertung durchgeführt werden.

2.3.2 Eigenschaften von Diensten auf Cloud-Infrastrukturschicht

IaaS-Anbieter ermöglichen das Mieten von virtuellen Maschinen (VMs) in verschiedenen Konfigurationen. Da die Vorstellung der einzelnen Konfigurationen der am Markt öffentlich verfügbaren Cloud-Anbieter hier zu weit führen würde, sind deren spezifische Eigenschaften in Anhang B zusammengefasst. Nachfolgend werden verschiedene Dienste auf Infrastrukturschicht allgemein vorgestellt.

[6] *„Als Denial of Service (kurz DoS, englisch für: Dienstverweigerung) wird in der digitalen Datenverarbeitung die Nichtverfügbarkeit eines Dienstes bezeichnet, der eigentlich verfügbar sein sollte. Obwohl es verschiedene Gründe für die Nichtverfügbarkeit geben kann, spricht man von DoS in der Regel als die Folge einer Überlastung von Infrastruktursystemen. Dies kann durch unbeabsichtigte Überlastungen verursacht werden oder durch einen mutwilligen Angriff auf einen Server, einen Rechner oder sonstige Komponenten in einem Datennetz.“* (Onlineressource `http://de.wikipedia.org/wiki/Denial_of_Service` (abgerufen am 2.11.2013))

IaaS-Anbieter wie *Amazon EC2*, *Rackspace* und *Elastic Hosts* betreiben dazu Rechenzentren an verschiedenen Standorten weltweit. *Amazon EC2* und *Rackspace* bieten feste Konfigurationen von VM-Typen an. Bei *Elastic Hosts* kann der Kunde hingegen selbst die CPU-Kernanzahl, Arbeitsspeicher und weitere Parameter konfigurieren. Auch die Abrechnungsmodelle unterscheiden sich stark. Die Bezahlung erfolgt üblicherweise nach Verbrauch (*pay-as-you-go*), aber auch *Prepaid*-Modelle werden vereinzelt angeboten. *Amazon EC2* bietet zudem beispielsweise drei verschiedene Mietmodelle an. Die sogenannten *On-Demand*-Instanzen werden pro Stunde gemietet und pro Stunde bezahlt. Dies Variante ist immer garantiert verfügbar und sehr flexibel abzurechnen, führt in der Regel aber auch zu den höchsten Kosten. Benötigt man hingegen einfach nur Rechenzeit, die keinen zeitlichen Beschränkungen unterliegt, so kann man *Spot*-Instanzen mieten, deren Preis sich nach marktwirtschaftlichen Regeln auf Basis von Angebote und Nachfrage in kurzen Abständen ändert. Um eine Instanz zu erhalten, muss man einen Preis bieten. Liegt dieser über dem momentanen Marktpreis für den VM-Typ, so erhält man die Instanz zum aktuellen Marktpreis. *Spot*-Instanzen können jedoch jederzeit unterbrochen werden, beispielsweise wenn der aktuelle Marktpreis das Höchstgebot übersteigt. Sie sind deshalb nur für ausfalltolerante Anwendungen, wie etwa Datenanalyse, geeignet. Außerdem gibt es noch *Reserved*-Instanzen. Diese sind ebenfalls garantiert verfügbar und werden pro Stunde abgerechnet, jedoch wird für die Nutzung ein Vertrag über 1–3 Jahre geschlossen und eine Vorabgebühr gezahlt. Im Gegenzug verbilligen sich dadurch die Nutzungskosten pro Stunde erheblich, sodass diese Variante für häufig genutzte und garantiert verfügbare Anwendungen die beste Lösung darstellt. Jeder IaaS-Anbieter hat eigene Abrechnungsmodelle. Für die jeweilige Applikation muss demnach von Fall zu Fall neu entschieden werden, welcher Anbieter die günstigste Lösung bietet.

Das Betriebssystem nebst Softwarestack der Cloud-Applikation wird auf einem VM-Image gespeichert, welches von der VM-Instanz ausgeführt wird. Jeder Nutzer kann den aktuellen Zustand einer VM als Images speichern, dies wird oft als Snapshot bezeichnet. Alle IaaS-Anbieter stellen Vorgabeimages mit häufig verwendeten Betriebssystemen und Software bereit. Manche Anbieter erlauben auch das Installieren beliebiger Systeme. Alle in Anhang B aufgeführten Anbieter unterstützen Linux-Images. Viele bieten auch Windows-Images an, die oft etwas teurer sind. In der Regel kann ein VM-Image auf Instanzen verschiedenen Typs ausgeführt werden. Dies gilt jedoch nur, solange die darunter liegende Virtualisierungstechnologie die gleiche ist. *Amazon EC2* bietet beispielsweise *paravirtualisierte* Instanzen und *hardwarevirtualisierte* Instanzen an. Bei beiden Varianten wird ein Hypervisor verwendet, um beispielsweise den Zugriff auf die verschiedenen Hauptspeicherpartitionen der VM-Instanzen zu regeln. Der Hypervisor fungiert als eine Art Hardwarevermittler für die Gastbetriebssysteme. Von *Amazon EC2* wird die Virtualisierungstechnologie *Xen*[7] verwendet [LYKZ10]. Bei der Paravirtualisierung ist hierfür eine Anpassung des Betriebssystems nötig. Bei der Hardwarevirtualisierung ist dies nicht nötig, dafür müssen jedoch spezielle Prozessorerweiterungen vorhanden sein. Eine Übersicht über verschiedene Virtualisierungstechnologien ist beispielsweise in [RHFN+12] zu finden. VM-Images in *Amazon EC2* (*Amazon Machine Images* (AMIs)) können jeweils nur auf einer Virtualisierungstechnologie ausgeführt werden. Neuere *hardwarevirtualisierte* Instanzen erlauben den Zugriff auf die Hardware der entsprechenden Maschine.

[7] *Xen* ist ein Hypervisor, eine Software, die den Betrieb mehrerer virtueller Maschinen auf einem physischen Computer erlaubt. *Xen* entstand an der britischen *University of Cambridge*. *Xen* wurde von Beginn an für beste Performance optimiert und konnte dies auch gegenüber anderen Virtualisierungstechnologien unter Beweis stellen. (siehe [BDF+03])

Dies wird beispielsweise beim Zugriff auf die Grafikkarte für GPU-unterstützte Applikationen benötigt.

Für den in diese Arbeit notwendigen Softwarestack mit Verwendung von *Java*, sind einfach ausgestattete Linux-Instanzen und eine darauf installierte *Java*-Laufzeitumgebung ausreichend. *Amazon EC2* stellt für solche Aufgaben eigene VM-Images mit einer Amazon Linux Distribution zur Verfügung. Diese beinhalten nur den nötigsten Softwarestack und schließen bereits die zur Verwaltung notwendigen Kommandozeilenwerkzeuge zum Ansprechen der *Amazon EC2*-API ein.

Außerdem bieten IaaS-Anbieter weitere Dienste an. Beispielsweise können VM-Instanzen zur allgemeinen Verfügbarkeit mit statischen IP-Adressen versehen werden. Weiterhin können virtuelle Lans (VLANs) eingerichtet werden, um Cloud-Ressourcen direkt in ein Unternehmensinternes Netzwerk einzugliedern. Zusätzlich kann eine Firewall konfiguriert werden, um die Cloud-Ressourcen abzuschotten. Abschließend ist es zudem möglich ein Monitoring an die gemieteten Instanzen anzuheften, welches bei bestimmten Lastzuständen Alarm auslöst. Jeder Anbieter stellt diese Zusatzdienste in verschiedener Ausprägung, Qualität und Kosten zur Verfügung. Somit ist auch hier von Fall zu Fall zu entscheiden, welche Eigenschaften benötigt werden und welche Kosten dafür anfallen.

2.3.3 Lastbalancierung und Skalierungsmethoden

Die QoS eines Cloud-Dienstes sollte unabhängig von der Anzahl gleichzeitiger Nutzeranfragen sein. In der Schichteneinteilung der Cloud-Dienste gibt es die Plattform-Dienste (PaaS), die es ermöglichen, Software zu schreiben, die Lastbalancierung und Skalierung der Ressourcen automatisch implementiert. Cloud-Anbieter, die eine PaaS anbieten, skalieren selbstständig die benötigten Ressourcen, die notwendig sind, um die für den Nutzer gehostete Applikation innerhalb der vereinbarten Parameter (SLA) zu betreiben. Dabei werden, wie etwa bei *Googles App Engine*, Gebühren für jede einzelne Anfrage an die Applikation erhoben. Im Vergleich dazu werden bei IaaS-Anbietern im Allgemeinen Gebühren für jede Stunde Nutzungsdauer einer laufenden virtuellen Maschine erhoben, zuzüglich Kosten für Speicher und Datenübertragung via Netzwerk. Das PaaS Modell ist zwar relativ feingranular, bietet aber auch kaum Möglichkeiten der Einflussnahme, wie beispielweise eine situationsangepasste Bearbeitungsgeschwindigkeit.

Das IaaS-Modell ist sehr grobgranular, bietet dafür aber das höchste Maß an Konfigurierbarkeit im Cloud-Schichtmodell. Dazu bietet die IaaS-Schicht die Möglichkeit, die Anzahl der Ressourcen, die serverseitig genutzt werden, an die jeweilige Anfragemenge anzupassen. Bei dieser mengenmäßigen Anpassung spricht man von horizontalem Skalieren. Skalierbarkeit wird wie folgt beschrieben:

> „*Im Allgemeinen bezeichnen wir Systemarchitekturen als skalierbar, wenn sie sich jedem zukünftigen Lastzuwachs anpassen können, sei er erwartet oder nicht.*“[Emm03, S. 22]

Eine SaaS ist also besonders gut skalierbar, wenn sie in der Lage ist, durch Vergrößerung der bereitgestellten Ressourcen einen möglichst proportional größeren Anfragedurchsatz zu erzielen und wenn die Skalierung so funktioniert, dass der mögliche Anfragedurchsatz zu jedem Zeitpunkt in etwa dem tatsächlichen Anfrageverhalten entspricht. Dem gegenüber spricht man von vertikalem Skalieren, wenn man nicht die Anzahl von Ressourcen verändert, sondern deren individuelle Geschwindigkeit. Man kann eine VM dazu beispielsweise auf eine performantere

Hardware migrieren. Da das Anfrageverhalten vorweg nicht bekannt ist, kann eine optimale Ressourcennutzung nicht garantiert werden. Durch Anwendung von Heuristiken wird jedoch stets versucht das Optimum zu erreichen.

Ziel einer horizontalen Skalierung ist es, immer gerade so viele Server wie nötig verfügbar zu haben. Um dies zu gewährleisten, schaltet man auch Maschinen ab, wenn sie nicht mehr benötigt werden. Um eine Entscheidung bezüglich der Aufwärts- und Abwärtsskalierung zu treffen, kann man beispielsweise die mittlere Server-Auslastung messen. Steigt diese über 80%, skaliert man nach oben, fällt sie unter 20%, skaliert man nach unten. Dieses Verfahren ist jedoch recht träge, da es erst reagiert, wenn es oft schon sehr spät ist [CRB11]. Es dauert einige Minuten, bis neue Maschinen gestartet sind. Daher können auch vorausschauende Verfahren angewendet werden, die aus dem aktuellen Anstieg der Server-Auslastung eine Prognose für den Zeitpunkt geben, zu dem die neuen Maschinen schon gestartet sind. Ein solches Verfahren ist in [GGW10] beschrieben. Prognosealgorithmen werden häufig auch mit maschinellem Lernen kombiniert, um nicht nur aus den aktuellen Daten, sondern auch aus vorangegangenen Messungen eine Prognose zu erstellen [RDG11].

Die Auswahl der Lastbalancierungsstrategie richtet sich auch nach der Länge und Komplexität der Anfragen. Ziel der Lastbalancierung ist es, alle Anfragen so auf die verfügbaren Server zu verteilen, dass diese möglichst gleichmäßig ausgelastet sind. Einfache Web-Anfragen können dabei oft ad hoc verteilt werden. Komplexe Anfragen, die zusätzlich eine Datenbankabfrage auslösen, sind schwieriger zu balancieren. Hierbei handelt es sich um Mehr-Schicht-Architekturen, die eine Anfrage in mehrere kleinere Anfragen aufsplitten, welche für sich zusätzlich balanciert werden müssen. Für die hier betrachteten berechnungsintensiven Anwendungen sind einfache Anfragen zu erwarten, die sich jedoch in ihrer Länge unterscheiden können. Solche Anfragen lassen sich oft auch als unabhängige Tasks formulieren. Somit können zur Lastbalancierung auch Algorithmen aus den Bereichen Task-Scheduling und Job-Scheduling zum Einsatz kommen [OK10]. *Cloud Computing* erlaubt dabei aufgrund seiner marktwirtschaftlichen Ausrichtung zusätzliche Optimierungen beispielsweise auf Kosteneffizienz anstatt auf Geschwindigkeit.

In Bezug auf MCC-Anwendungen wird von einzelnen spontan auftretenden Anfragen ausgegangen. Scheduling und Lastbalancierung werden somit *online* durchgeführt. Ein Überblick über verschiedene Techniken für *Online Scheduling* ist beispielsweise in [Leu04] zu finden. Je nach Charakteristik der einzelnen Anfragen sind verschiedene Lastbalancierungsstragien möglich. Haben alle Anfragen etwa die gleiche Bearbeitungsdauer, so wird das Rundlauf-Verfahren (englisch *Round-Robin*) angewendet und die Anfragen werden reihum auf die verfügbaren Server verteilt. Unterscheiden sich die Anfragen hingegen in der Länge, führt das Rundlauf-Verfahren nicht mehr zu einer gleichmäßigen Auslastung. Man teilt neue Anfragen in diesem Fall immer dem Server mit der geringsten Anzahl aktuell aktiver Anfragen zu. Man nennt dieses Verfahren *Shortes Queue First* [CM99]. Auch stochastische Verfahren, die die Anfragen per Zufall mit definierten Wahrscheinlichkeiten verteilen, sind möglich [FHR09].

Unter Verwendung von Infrastrukturdiensten eines IaaS-Anbieters müssen Lastbalancierungs- und Skalierungsmethoden für darauf aufbauende Applikationen stets gemeinsam betrachtet werden. Im Vergleich zu anderen Technologien, wie *Grid* oder einem traditionellen Rechenzentrum, beeinflussen sich beide Methoden wechselseitig. Beispielsweise hängen Skalierungsalgorithmen oft von der Serverauslastung ab, welche wiederum durch die Lastbalancierung beeinflusst wird.

2.3.4 Cloud-QoS und Monitoring

Im Vergleich zu traditionellen Rechenzentren hat sich die Art des Aushandelns und Festschreibens der Dienstqualität zwischen Anbieter und Benutzer stark geändert. Das *self-service* Prinzip im *Cloud Computing* stellt für alle Benutzer gleichermaßen ein Service-Level-Agreement zur Verfügung und ersetzt damit das separate Aushandeln mit einzelnen Benutzern [Sos10]. Was für den Anbieter aus ökonomischer Sicht Vorteile bringen kann, kann den Benutzer aber auch Flexibilität kosten, da er beispielsweise die Verfügbarkeit nicht mehr nach seinen Bedürfnissen aushandeln kann. Der Benutzer muss in diesem Fall nehmen, was ihm der Anbieter anbietet. Darüber hinaus sind bei Verletzung der SLA durch den Anbieter üblicherweise auch keine Strafzahlungen vorgesehen. Außerdem lässt sich ein Monitoring zur Überwachung der SLA im Cloud-Bereich auch schwer implementieren. Hier stellt sich beispielsweise die Frage, ob Maschinen des gleichen Typs auch immer die gleiche Performance aufweisen. Daraus ergibt sich, dass die einzelnen Anbieter auf dem Cloud-Markt auch immer mehr mit ihrer Reputation werben müssen. Kunden werden also nicht nur den Preis und die gebotene Leistung der einzelnen Anbieter vergleichen. Sie werden auch vergleichen, wie stabil die einzelnen Anbieter ihre Leistung anbieten. Dem gegenüber steht jedoch auch die bereits beschriebene Schwierigkeit, einen einmal ausgewählten Anbieter zu verlassen und zu einem anderen Anbieter zu migrieren (*Vendor Lock-in*).

Services sollten vor allem im Cloud-Umfeld einen Service-Lebenszyklus und autonomes Management unterstützen. Der Service-Lebenszyklus ist teil des SLA Managements. In [WBTY11, S. 13ff.] werden dazu verschiedene Stationen eines Lebenszyklus beschrieben:

1. *Desgin and Development* – Entwicklung des Dienstes durch Anbieter,
2. *Service Offering* – Anbieten des Dienstes via Datenblatt, Werbung, etc.,
3. *Service Negotiation* – Aushandeln der Konditionen mit dem Benutzer,
4. *Service Provisioning* – Bereitstellen des Dienstes für den Benutzer,
5. *Service Operations* – Verwendung des Dienstes durch den Benutzer,
6. *Service Decommissioning* – Abschalten des Dienstes durch den Anbieter.

Das Projekt *SLA@SOI* befasst sich mit der Realisierung dieser Anforderungen zur Unterstützung Cloud-basierter Business-Anwendungen. In diesem Zusammenhang wird oftmals auch ein *Service-Broker* eingesetzt, um für den Benutzer einen dessen Anforderungen entsprechenden Service zu vermitteln. Die in [FE10, S. 575ff.] beschriebene und bereits weiter oben erwähnte Technologie *Cloud@Home* vermittelt Compute-Ressourcen an Benutzer in mehreren Schritten. Zunächst wird der Benutzer authentifiziert. Danach übersendet der Nutzer einen Computing Request mit den zugehörigen Anforderungen und der gewünschten Qualität (SLA). Das System beantwortet den Request und allokiert eine passende VM für den Benutzer. Die eigentliche Nutzung der VM erfolgt dann direkt durch Nutzer, ohne dass weitere Dienste der *Cloud@Home* Umgebung in Anspruch genommen werden müssen. Die Literatur macht jedoch keine Angaben, welche Anforderungen und Qualitätsstufen vom Benutzer spezifiziert werden können und auch nicht, wie danach eine geeignete VM gefunden und allokiert wird. Außerdem wird keine Aussage darüber getroffen, wie eine nicht mehr genutzte VM vom Benutzer an das System zurückgegeben wird.

Autonomes Service Management bedeutet im Cloud-Umfeld nicht nur das selbstständige Aushandeln von SLAs zwischen Benutzer und Anbieter unter gegebenen Anforderungen, sondern auch eine Rekonfigurierung im Fehlerfalle oder die Reaktion auf Verletzungen der SLA. Eine große Herausforderung im Cloud-Umfeld ist die Durchsetzung der SLA beziehungsweise deren Überwachung. In [WBTY11, S. 105ff.] wird eine Cloud-kompatible Variante des SLA *Penalty Managements* beschrieben. Zunächst müssen *Monitoring*-Parameter definiert werden, die dann während der *Service Operations* Phase überwacht werden können. In gewissen Abständen werden die gesammelten Daten dann ausgewertet und auf die Verletzung von vorher definierten SLA-Parametern untersucht. Tritt eine solche Verletzung auf, wird sie in einem weiteren Schritt bearbeitet. Als *Monitoring*-Parameter wird oftmals die Service-Antwortzeit benutzt. Das sogenannte *Penalty Management* entscheidet dann, wie zu verfahren ist. Gegebenenfalls sind auch mehrere Verletzungen pro Zeiteinheit nötig, um eine Aktion auszulösen. Eine gängige Lösung besteht darin, den Service, der die Verletzung verursacht hat, auf eine andere Ressource zu migrieren. Dies kann gegebenenfalls auch die Migration zu einem anderen Anbieter bedeuten. Dazu muss der Dienst jedoch bei einem anderen Anbieter auch verfügbar sein.

2.3.5 Zusammenfassung MCC-Infrastruktur

Aus Sicht des MCC ist eine zentrale Middleware, die auf einer IaaS-Umgebung aufgebaut ist, oft die bessere Wahl zur Bereitstellung feingranularer Compute-Ressourcen. Im Vergleich zu einer dezentral organisierten Middleware ist die Qualität der Ressourcen genauer definiert und deren Verfügbarkeit ist in der Regel besser. Der Nachteil der recht hohen Latenzen impliziert jedoch die Einschränkung auf MCC-Anwendungen, die keine schnellen Reaktionszeiten erfordern. Die Vermittlung von Compute-Ressourcen muss durch einen Front-Server erfolgen, wenn eine Zugangskontrolle und Abrechnung der Nutzung der Ressourcen erfolgen soll. Die Verwendung eines Front-Server ist zudem notwendig, um eine Lastbalancierung bei der Allokation der verfügbaren Ressourcen zu gewährleisten. Die dabei favorisierte Variante weist einen Client fest auf eine dedizierte Ressource zu, die der Client danach direkt anspricht. Die dabei auftretende Schwachstelle des System bei Ausfall des Front-Servers kann nur durch redundante Auslegung dieses Servers vermieden werden. Jede Cloud-Ressource sollte zudem ein Monitoring implementieren, um einerseits die Abrechnung der Nutzungsdauer zu realisieren und andererseits auch die Performance der einzelnen Ressourcen zu überwachen. Da die SLAs der verfügbaren öffentlichen IaaS-Anbieter eine Bestrafung von SLA-Verletzungen nicht vorsehen, müsste man diese Bestrafung selbst implementieren. Dazu sollte die MCC-Infrastruktur mehrere IaaS umfassen. Dann kann man im Rahmen einer SLA-Verletzung den Dienst auf einen anderen IaaS-Anbieter migrieren, der beispielsweise aufgrund von erhobenen Messdaten eine konstantere Dienstqualität liefert.

2.4 Fazit

Von den in diesem Kapitel vorgestellten Arbeiten, die unter dem Schlagwort MCC zu finden sind, betrachten nur [MPPS+12, KPKB10] auch die Allokation der benötigten Cloud-Ressourcen. Alle Arbeiten aus der betrachteten Literatur können jedoch lokal verfügbare Serverressourcen nutzen. Die Verwendung von Flächenkommunikationsnetzen (beispielsweise 3G Mobilfunk) ist nicht überall möglich. Manche Ansätze sind nur mit WLAN nutzbar.

Somit ist die breite Anwendbarkeit der vorgestellten Arbeiten nicht sichergestellt. Auch die Abrechnung der Nutzungskosten für Cloud-Ressourcen und die dafür notwendige Integration einer Benutzerverwaltung werden von keinem der vorgestellten Ansätze betrachtet. Außerdem wird die Allokation von entfernt verfügbaren Ressourcen verschiedener Qualität in den betrachteten Arbeiten nicht unterstützt. Im Ergebnis konnte somit keine Arbeit aus der Literatur identifiziert werden, die den Anforderungen dieser Arbeit gerecht wird. Deshalb werden in dieser Arbeit im Nachfolgenden zumeist eigene Techniken erarbeitet, um die gestellten Anforderungen zu erfüllen.

Bezüglich der Anforderung einer breiten Anwendbarkeit von *Mobilen Cloud-unterstützten Anwendungen* unter Verwendung von öffentlicher Cloud-Infrastruktur sind nur Techniken anwendbar, die keine Modifikationen an vorhandenen Geräten oder deren Betriebssystem erfordern. Somit scheiden Ansätze aus, die eine automatische Partitionierung der Software zur Laufzeit implementieren, da dies oft die Modifikation der Ausführungsplattform erfordert. Auch Ansätze der Bildschirmübertragung scheiden aus, da hier ein hoher Datenoverhead auftreten kann, der für öffentlich verfügbare Flächenkommunikationsnetze nicht geeignet ist.

Es wird eine zentral organisierte Middleware favorisiert, die über einen Front-Server den Mobilgeräten zur Verfügung steht. Jedem Endgerät/Benutzer wird eine dedizierte Ressource für Compute-Aufgaben zur Verfügung gestellt, um eine hohe Ausführungsperformance zu erreichen. Zudem wird die Technik des dynamischen Code-Offloadings favorisiert, um Programmcode auf den Compute-Servern zu installieren. Code-Offloading bieten den Vorteil, dass das Mobilprogramm auch ohne Serverunterstützung lauffähig ist, beispielsweise wenn keine Compute-Ressourcen genutzt werden können aufgrund zu geringer verfügbarer Netzwerkperformance. Außerdem sind die Compute-Ressourcen bei der Anwendung von Code-Offloading generisch, was deren Management in einer Cloud-Infrastruktur vereinfacht.

Beim Code-Offloading ist es auf Serverseite notwendig, Programmcode zur Laufzeit kompilieren, laden und ausführen zu können. Um dies zu realisieren, ist es zudem sinnvoll, wenn Mobilgerät und Compute-Server die gleiche Softwareplattform verwenden. Dann kann zur Laufzeit leicht über die lokale oder entfernte Ausführung entschieden werden. Verbreitete Softwareplattformen, die serverseitig und auf Mobilgeräten eingesetzt werden, sind etwa *Java* oder .NET. *Java* ist jedoch dank *Android* auf wesentlich mehr Mobilgeräten verfügbar, weshalb die Wahl der Programmiersprache und Softwareplattform in der vorliegenden Arbeit für die weiteren Untersuchungen und die zu entwickelnde Middleware auf *Java* und *Android* fiel.

3 Modell für Mobile Server-unterstützte Anwendungen

In diesem Kapitel wird unter Beachtung der bereits analysierten Literatur ein Modell vorgestellt, welches Kosten- und Performance-Eigenschaften von *Mobilen Cloud-unterstützten Anwendungen* abbilden kann. Da kein vorhandenes Modell unverändert übernommen werden konnte, wurde für diese Arbeit ein eigenes Modell erarbeitet, in das einzelne Aspekte aus [HBDTD07] und [CLK+11] eingeflossen sind. Dieses Modell wird zunächst allgemein für *Mobile Server-unterstützte Anwendungen* aufgestellt und in den nachfolgenden Kapiteln zur Erfassung verschiedener spezieller Aspekte weiter verfeinert.

Aus der Analyse der vorgestellten Literatur in Kapitel 2 ist bereits argumentiert worden, dass eine MCC-Architektur mit Front-Server favorisiert wird. Dabei realisiert der Front-Server die Authentifizierung des Benutzers und die Allokation der Compute-Ressource. Compute-Ressourcen stammen aus dem Portfolio eines öffentlichen IaaS-Anbieters, liegen also in verschiedener, aber definierter Qualität vor. Der auszuführende Programmcode wird zur Laufzeit mittels Code-Offloading zunächst durch den Client auf der Compute-Ressoure installiert um ihn anschließend aufrufen zu können. Abbildung 3.1 visualisiert die einzelnen Komponenten grafisch und zeigt deren Interaktion.

3.1 Mobile Server-unterstützte Anwendungen

Wie jedes Modell soll das hier vorgestellte die wichtigsten Eigenschaften reflektieren, die zur Analyse von *Mobilen Server-unterstützten Anwendungen* notwendig sind. Zunächst werden mobile Anwendungen und Server-unterstützte Anwendungen betrachtet, anschließend wird das Modell zur Erfassung des Einflusses von Cloud-Technologien erweitert.

3.1.1 Abstrakte generische Anwendung

Ausgehend von einer abstrakten Anwendung werden zunächst die wichtigsten Eigenschaften eines Computerprogramms abgebildet. Ein Programmlauf L beschreibt die Ausführung eines Programms P unter Verwendung der Eingabe E. Die Laufzeitfunktion $lauf(\dots)$ beschreibt die Ausführungszeit Δt_L eines Programmlaufs auf einer gegebenen Maschine/Ressource R.

$$\Delta t_L \quad = \quad lauf(R, L). \tag{3.1}$$

Tabelle 3.1 fasst die Eigenschaften des betrachteten Modells zusammen. Da man die gesamte Eingabe eines Programms zu Beginn oftmals nicht kennt, wird das Programm in Module zerlegt, die jeweils Eingaben akzeptieren, die weniger komplex sind. Als Ergebnis sind für die vollständige Bearbeitung der Eingabe eines Programmmoduls keine weiteren Programmmodule involviert. Ein Programmteil $p_i \in P$ und eine zu p_i passende Eingabe $e_i \in E$ bilden einen Programmteillauf (p_i, e_i). Diese Dekomposition macht es möglich, einige Programmteile mit bekannter Eingabe besser modellieren zu können. Beispielsweise lässt sich ein Programm so in berechnungsintensive Teile und eine grafische Benutzeroberfläche zerlegen. Dadurch lässt

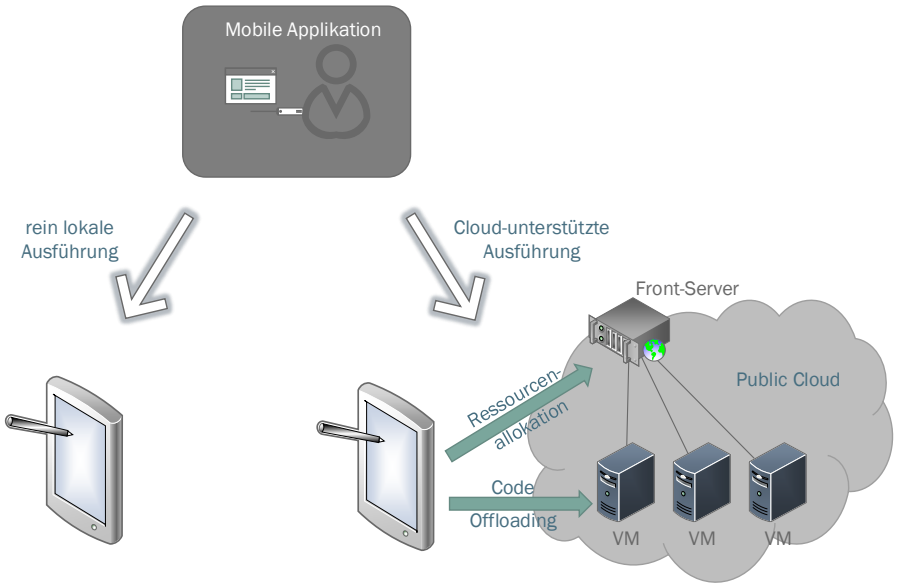


Abbildung 3.1: Ausführung einer *Mobilen Cloud-unterstützten Anwendung* unter Verwendung von Code-Offloading im Vergleich zu einer rein lokalen Ausführung.

sich für die berechnungsintensiven Programmteile die Ausführungszeit besser erfassen, obwohl andere Programmteile von Benutzereingaben abhängig sein können. Ein Programmlauf L repräsentiert die Instanziierung einer Reihenfolge von Programmteilläufen, also genau einer möglichen Ausführungsreihenfolge des Programms. Ein Programmlauf L ist eine durch die Ordnungsrelation $\leq$ vollständig geordnete Menge und wie folgt definiert:

$$L \quad = \quad ((p_0, e_0), \ldots, (p_n, e_n)) \quad mit \quad n = |P|. \tag{3.2}$$

Die Ordnungsrelation $\leq$ bedeutet, dass alle Elemente $(p_i, e_i) \in L$ nach ihrem Index i aufsteigend angeordnet sind. Die Ordnungsrelation $\leq$ gilt somit auch auf allen Teilmengen von L. Jeder Programmteillauf (p_i, e_i) hat eine Ausführungszeit Δt_i unter Verwendung einer konkreten Eingabe auf einer konkreten Ressource R. Als spezieller Programmteillauf kann dabei eine Wartephase (p_{wait}) auftreten, beispielsweise zur Erfassung von Benutzereingaben oder Ein-/Ausgabeoperationen. Während einer Wartephase wird kein Programmcode ausgeführt. Aus der Summe der Programmteilläufe lässt sich wiederum die Gesamtausführungszeit Δt_L für einen Programmlauf zusammensetzen. Dies ist nachfolgend beschrieben:

$$\Delta t_i \quad = \quad lauf(R, p_i, e_i) \quad \text{mit} \quad (p_i, e_i) \in L, \tag{3.3}$$

$$\Delta t_L \quad = \quad \sum_{i=0}^{|L|} \Delta t_i. \tag{3.4}$$

Hierbei wird die nebenläufige Ausführung von Programmteilen explizit ausgeschlossen. Innerhalb eines Programmteils p_i kann jedoch eine nebenläufige Ausführung erlaubt sein. Dies ist gleichzeitig auch eine Restriktion des Modells, das nicht jede mögliche Dekomposition

Tabelle 3.1: Modell zur Beschreibung der Ausführungszeit von Computerprogrammen

Definition	Beschreibung
$p_i \quad , i \in \mathbb{N}$	i-tes Programmmodul oder -funktion
$e_i \quad , i \in \mathbb{N}$	eine beliebige gültige Eingabe für p_i
$P = \{p_0, \ldots, p_n\} \quad , P \neq \emptyset$	Gesamtprogramm
$E = \{e_0, \ldots, e_n\} \quad , E \neq \emptyset$	Gesamteingabe des Programms
R	Maschine/Ressource, auf der das Programm ausgeführt wird
$lauf(R, L)/lauf(R, p_i, e_i)$	Ausführungszeit von P bzw. p_i auf R mit der Eingabe E bzw. e_i
$L = ((p_0, e_0), \ldots, (p_n, e_n)) \quad , n = \lvert P \rvert$	spezifischer Programmlauf
$\Delta t_L = lauf(R, L)$	Ausführungszeit des Programmlaufs L auf R
$\Delta t = lauf(R, p_i, e_i)$	Ausführungszeit eines Programmmteillaufs auf R

eines Programms P abbilden kann. Beispielhaft kann etwa eine Applikation zur Bildverarbeitung genannt werden, bei der nur die einzelnen Bildverarbeitungsschritte durch Parallelverarbeitung beschleunigt werden können. Die Auswahl, das Einlesen und Anzeigen der Bilder ist jedoch ein sequentieller Vorgang. Somit repräsentieren Auswahl, Einlesen, Anzeigen und Verarbeiten jeweils einzelne Programmteilläufe. An diesem Beispiel wird zudem deutlich, dass die vollständige Eingabe zu einem Programmteillauf nur beim Verarbeitungsschritt und eventuell beim Anzeigen verfügbar ist. Die beiden anderen Programmteilläufe erwarten weitere Benutzereingaben zur Laufzeit. Aus diesem Szenario lässt sich außerdem ableiten, dass man Programmteilläufe in zwei Arten kategorisieren kann, nämlich in die Menge $L_{processing}$ der Programmteilläufe, die tatsächlich etwas verarbeiten und in die Menge L_{idle} der Programmteilläufe, die nur aus Wartezeiten, beispielsweise auf (Benutzer)-Eingaben, bestehen und weder Programmcode ausführen noch eine Eingabe haben. Ein solcher spezieller Programmteillauf wird als (p_{wait}, e_{empty}) angegeben.

$$L_{idle} = \{(p_i, e_i) \in L \quad mit \quad p_i = p_{wait} \quad \text{und} \quad e_i = e_{empty}\} \tag{3.5}$$

$$L_{processing} = L \setminus L_{idle}. \tag{3.6}$$

Die Ausführungszeit eines Programmlaufs L kann nun also wie folgt zusammengesetzt werden:

$$\Delta t_{processing} = \sum_{\forall (p_m, e_m) \in L_{processing}} \Delta t_m, \tag{3.7}$$

$$\Delta t_{idle} = \sum_{\forall (p_n, e_n) \in L_{idle}} \Delta t_n, \tag{3.8}$$

$$\Delta t_L = \Delta t_{processing} + \Delta t_{idle}. \tag{3.9}$$

Für die nachfolgenden Betrachtungen sind nur die Programmteilläufe aus $L_{processing}$ interessant. Für diese lässt sich sagen, dass sie von der Eingabe und der zur Berechnung verwendeten Maschine abhängen. Weiterhin wird jedoch auch sichtbar, dass die vom Benutzer abhängigen Wartezeiten die Gesamtprogrammausführungszeit deutlich beeinflussen können. Da diese Wartezeiten kaum durch Serverunterstützung beeinflussbar sind, werden sie von den

nachfolgenden Betrachtungen ausgeschlossen. Es werden nur Programme oder Programmteile betrachtet, die ausschließlich aus Programmteilläufen bestehen, die keine $(p_n, e_n) \in L_{idle}$ enthalten.

3.1.2 Server-unterstützte Anwendung

In diesem Abschnitt wird gezeigt, welche Erweiterungen für das vorgestellte Modell zur Erfassung von Serveruntstützung für Anwendungen notwendig sind. Ein vormals lokal ausgeführtes Programm wird dazu in serverseitig ausgeführte und lokal ausgeführte Programmteilläufe partitioniert. Dies bedeutet, dass einige Programmteilläufe nun auf einer anderen (Server-) Ressource ausgeführt werden. Außerdem muss die neu auftretende Kommunikationszeit zusätzlich ins Modell aufgenommen werden.

Ein vollständig auf dem Mobilgerät ausgeführter Programmlauf wird als $L_{processing\,mobil}$ bezeichnet und dessen Ausführungszeit als $\Delta t_{processing\,mobil}$ angegeben. Die Ausführungszeit wird durch Anpassung von Gleichung (3.7) wie folgt berechnet:

$$\Delta t_{processing\,mobil} \quad = \quad \sum_{m=0}^{j} \Delta t_{mobil\,m} \quad mit \quad j = |L_{processing\,mobil}|\,. \tag{3.10}$$

Dabei beschreibt Δt_{mobil} die lokale Ausführungszeit und Δt_{remote} die Ausführungszeit für einen entfernt ausgeführten Programmteillauf. Auf einem Mobilgerät lässt sich nun die Aufspaltung in eine Server-unterstützte Variante der Ausführung modellieren, bei der k Programmteilläufe entfernt ausgeführt werden. Die Ausführungszeit $\Delta t_{processing\,hybrid}$ für eine so zusammengesetzte Art der Ausführung wird wie folgt modelliert:

$$\Delta t_{processing\,hybrid} \quad = \quad \sum_{m=0}^{j-k-1} \Delta t_{mobil\,m} + \sum_{l=j-k}^{j} \Delta t_{remote\,l}. \tag{3.11}$$

Um das Ziel der schnelleren Ausführung Server-unterstützter Anwendungen zu verdeutlichen, sollte die Ausführungszeit des gleichen entfernt ausgeführten Programmteillaufs stets kleiner sein als die lokal ausgeführte Variante:

$$\Delta t_{mobil\,m} \quad \geq \quad \Delta t_{remote\,m} \quad , \quad (j-k) \leq m < j. \tag{3.12}$$

Durch die Ausführung auf einem entfernten Server werden zusätzlich zu serverseitigen Berechnungszeiten Δt_{work} Datenübertragungszeiten Δt_{comm} notwendig, die auch das Codieren und eventuell das Verschlüsseln der Daten mit einbeziehen. Somit lässt sich $\Delta t_{remote\,l}$ erneut aufspalten in:

$$\Delta t_{remote\,l} \quad = \quad \Delta t_{comm\,l} + \Delta t_{work\,l}. \tag{3.13}$$

Dabei hängt Δt_{comm} von verschiedenen Faktoren wie der Größe der zu übertragenden Daten, der verfügbaren Netzwerkperformance und der Codierung ab. Zur Vereinfachung wird Δt_{comm} zunächst nur durch die Funktion $comm(e_l)$ repräsentiert, die lediglich von der Eingabe e_l abhängig ist. Für die Kommunikationszeit spielt insbesondere die Größe der Eingabe in Byte eine Rolle.

$$\Delta t_{comm} \quad = \quad comm(e_l). \tag{3.14}$$

Der Einfluss einzelner Faktoren wie der Übertragungsrate auf die Kommunikationszeit wird in Kapitel 4 genauer betrachtet. Anhand des Modells wird jedoch bereits ersichtlich, dass es schwierig ist, für Server-unterstützte Anwendungen eine gleichbleibende Anwendungsperformance zu realisieren. Dies ist bedingt durch folgende Ursachen.

In traditionellen Client/Server Anwendungen werden Client-Anfragen auf die verfügbaren Server-Ressourcen in der Art zugewiesen, dass eine gleichmäßige Lastverteilung auf den verfügbaren Ressourcen erreicht wird. Trotz dieser Lastbalancierung kann meist nur eine obere Schranke bezüglich der Antwortzeit garantiert werden. Und selbst diese ist nur in beispielsweise 99,9 % aller Fälle garantiert. Für das Modell bedeutet dies, dass Δt_{work} für gleiche Anfragen zu unterschiedlichen Zeiten stark schwanken kann. In Kriterium (3.12) wird jedoch eine Mindest-Performance vorgegeben, um sicherzustellen, dass sich eine entfernte Ausführung positiv auf die Gesamtausführungszeit auswirkt.

Weiterhin reduzieren sehr viele gleichzeitige Anfragen an einen Server auch die verfügbare Netzwerkperformance für jede einzelne Anfrage, weil sich alle Clients die verfügbare Bandbreite zum Server teilen müssen. Dadurch steigt auch die Kommunikationszeit Δt_{comm} eventuell an, was zu einer Verschlechterung der Gesamtreaktionszeit führt. Insgesamt betrachtet kann damit auf einem mobilen Gerät auch bei Kenntnis der aktuellen Netzwerkperformance und der zu übertragenden Daten keine Garantie gegeben werden, dass die Ausführungszeit einer Server-unterstützten Anwendung ohne weitere Verbesserungen durch eine obere Schranke begrenzt wird.

3.2 Mobile Cloud-unterstützte Anwendungen

Technologien aus dem Bereich *Cloud Computing* können an dieser Stelle eingesetzt werden, um einige der Restriktionen im Bereich *Mobiler Server-unterstützter Anwendungen* aufzuweichen. Durch Virtualisierung von Komponenten im Bereich IaaS und PaaS kann die Anzahl der benötigten Serverressourcen besser an den aktuellen Bedarf angepasst werden. Zentrale Idee dabei ist es, dass jede Anfrage auf eine dedizierte Serverinstanz R zugewiesen wird. Damit wird erreicht, dass die Netzwerkperformance und die Verarbeitungsgeschwindigkeit für jede Anfrage konstant sind. Genauer wird davon ausgegangen, dass exakt gleiche Programmausführungen (p_l, e_l) zu jedem Zeitpunkt $t_1, t_2, \ldots, t_n$ auf der gleichen Ressource mit der gleichen Ausführungs- und Kommunikationszeit bearbeitet werden. Dazu werden die Funktionen $lauf(R, p_l, e_l)$ und $comm(e_l)$ aus den Gleichungen (3.3) und (3.14) um eine Zeitkomponente erweitert. Es gilt für beliebige Zeitpunkte t_1 und t_2:

$$comm(e_l, t_1) = comm(e_l, t_2), \tag{3.15}$$

$$lauf(R, p_l, e_l, t_1) = lauf(R, p_l, e_l, t_2). \tag{3.16}$$

In Bezug auf Gleichung (3.13) bedeutet dies, dass eine gewisse obere Schranke für Δt_{comm} und Δt_{work} ermittelt werden kann, die jedoch von der Güte der Informationen, etwa bezüglich der aktuell verfügbaren Netzwerkperformance, abhängt. Dadurch lässt sich nun abschätzen, ob das Kriterium (3.12) durch die Auslagerung eingehalten werden kann. Außerdem ist es durch Verwendung von IaaS-Ressourcen R möglich, Δt_{work} zu beeinflussen, indem man die zur Berechnung verwendete Ressource entsprechend der Anforderungen anpasst. Da auch Cloud-Ressourcen nicht unbegrenzt zur Verfügung stehen, bedarf es einer Scheduling-Komponente,

die Anfragen auf verfügbare Cloud-Ressourcen zuweist. Diese Scheduling-Komponente ist auch dafür verantwortlich, die Anzahl vorgehaltener Cloud-Ressourcen zu regulieren und dem aktuellen Bedarf anzupassen. Man spricht dabei vom horizontalen Skalieren (siehe Kapitel 2.3.3). Dabei kann es vorkommen, dass bei einer großen Anzahl plötzlich auftretender Anfragen nicht genügend Ressourcen (VMs) zur Verfügung stehen. Die Allokationszeit Δt_{alloc} beschreibt die Zeit, die vergeht, bis eine Anfrage auf eine verfügbare VM zugewiesen werden konnte. Im *Cloud Computing* ist es üblich, die Verwendung von Ressourcen nach dem *pay-per-use* Modell in Rechnung zu stellen. Dies impliziert eine Authentifikation zum Beginn der Nutzung von Cloud-Ressourcen, die die Nutzung einer Ressource einem Benutzer eindeutig zuordnet. Die Zeit, die für die Authentifizierung benötigt wird, wird mit Δt_{auth} angegeben. Daraus resultiert, dass Gleichung (3.13) zur Erfassung der genannten Aspekte erweitert werden muss:

$$\Delta t_{remote} \quad = \quad \Delta t_{auth} + \Delta t_{alloc} + \Delta t_{comm} + \Delta t_{work}. \tag{3.17}$$

Durch die Verwendung von Cloud-Ressourcen kann eine gewisse Verarbeitungsgeschwindigkeit garantiert werden. Jedoch wird auch ein gewisser Overhead für Allokation und Authentifizierung notwendig. Somit lohnt sich der Einsatz erst, wenn die Berechnungsdauer einen gewissen Umfang überschreitet. Außerdem wird aus Gleichung (3.3) auch deutlich, dass das Einhalten des Kriteriums (3.12) von der gewählten Cloud-Ressource R abhängig ist. Cloud-Ressourcen werden üblicherweise auf Stundenbasis gemietet. Unterschiedliche Ressourcentypen sind dabei verschieden teuer. Daraus ergibt sich für die Ressourcenauswahl zudem eine Kostenbeschränkung $cost_{limit}$:

$$cost_{per\,h}(R) \cdot ceil(\Delta t_{remote}) \leq cost_{limit}. \tag{3.18}$$

Die Funktion $cost_{per\,h}(R)$ liefert die Mietkosten für die Ressoure R pro Stunde und die Funktion $ceil(\Delta t_{remote})$ liefert die auf volle Stunden aufgerundete Nutzungsdauer. Diese stundenweise Abrechnung kann zu großem finanziellen Overhead führen, wenn die Nutzungszeit die Abrechnungsperiode immer nur kurz übersteigt, so werden beispielsweise 61 min als 2 h abgerechnet. Hier bedarf es also einer Wiederverwendungsstrategie für Cloud-Ressourcen R, um diesen Overhead zu minimieren.

3.3 Fazit

Das Modell beschreibt *Mobile Cloud-unterstützte Anwendungen*. Dabei wird eine Applikation in lokal ausgeführte Teile und entfernt ausgeführte Teile zerlegt. Das Modell ist in der Lage, den Einfluss verschiedener Faktoren auf die Gesamtausführungszeit abzubilden. In den nachfolgenden Kapiteln werden besonders die nachfolgenden Faktoren genauer untersucht:

- Art und Ausprägung der Laufzeitfunktion $lauf(R, p, e)$, die die Abhängigkeit zwischen einer Eingabe e und der Ausführungszeit eines Programmmoduls p berechnet,
- Abhängigkeit der Datenübertragungszeit Δt_{comm} vom verwendeten Netzwerk und der gewählten Codierung,
- Abhängigkeit der Allokationszeit einer Ressource (Δt_{auth} und Δt_{alloc}) von der gewählten Systemarchitektur, dem verwendeten Schedulingverfahren und der Implementierung des Service-Lebenszyklus.

Außerdem werden folgende Einschränkungen modelliert:

- Die Ausführungszeit der *Mobilen Cloud-unterstützten Anwendung* darf die Ausführungszeit der rein mobilen Anwendung nicht übersteigen.
- Die als maximal definierten Kosten für die Ausführung dürfen nicht überschritten werden.

Anhand des Modells soll nun die Frage beantwortet werden, welche Applikationen unter konkreten Instantiierungen des Modells realisierbar sind. Besonderes Augenmerk liegt dabei auf der Optimierung der Kosteneffizienz.

4 Verbindungen in 3G und WLAN Netzwerken

Aus dem Studium vorhandener Literatur in Kapitel 2 ist hervorgegangen, dass manche Technologien aus dem Bereich MCC zwar die verfügbare Kommunikationsperformance beachten, jedoch sind dazu keine Bewertungsmaßstäbe genannt. Ebenso wird auf die speziellen Eigenschaften mobiler Kommunikationstechnologien in der Literatur oft nicht eingegangen. Da die Performance mobiler Kommunikationsverbindungen auch immer von der konkreten Konfiguration abhängt, beschäftigt sich dieses Kapitel genauer mit der Analyse von kablellosen Netzwerkverbindungen aus dem Bereich WLAN und 3G Mobilfunk. Eine der beiden Verbindungsarten ist in Deutschland mit hoher Wahrscheinlichkeit an jedem Ort verfügbar. Ziel dieses Kapitels ist es, eine Möglichkeit zu finden, um die Kommunikationsperformance zur Laufzeit möglichst realistisch zu bewerten. Außerdem soll eine Kommunikationsbibliothek entwickelt und implementiert werden, die die bestmögliche Ausnutzung der mobilen Kommunikationsverbindung gewährleistet. Es werden Parameter bestimmt, die zur Laufzeit messbar sind und auf deren Basis eine Performancevorhersage getroffen werden kann. Zudem wird analysiert, welche Eigenschaften für eine mobile Kommunikationstechnologie charakteristisch sind, damit die Ergebnisse dieses Kapitels auch auf zukünftige Technologien übertragen werden können.

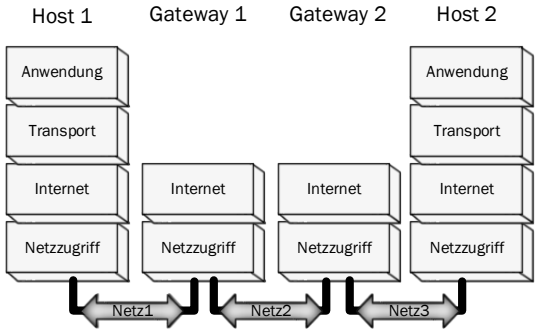
Eine belastbare Bewertung der verfügbaren Netzwerkperformance ist wichtig, da die Ausführungszeit serverseitig ausgeführter Programmmodulen von *Mobilen Cloud-unterstützten Anwendungen* davon abhängt. Dies ist in Gleichung (3.13) zu sehen, in der die Dauer der Netzwerkübertragung durch $\Delta t_{comm\,l}$ repräsentiert ist. Außerdem beeinflusst die Netzwerkperformance die Auswahl der Qualität der Server-Ressource. Es ist beispielsweise nicht nötig, eine sehr schnelle und teure Maschine zu allokieren, wenn diese dann aufgrund einer schmalbandigen Netzwerkkommunikation nicht schnell genug mit Daten versorgt werden kann.

Alle in dieser Arbeit betrachteten Netzwerkverbindungen bauen auf dem im Internet verwendeten TCP/IP-Protokollstapel auf. Somit startet die Analyse mit der Vorstellung dieses Modells und dessen Schichten. Anschließend werden die Netzzugangstechnologien WLAN und UMTS näher betrachtet. Weiterhin werden die Eigenschaften von TCP Ende-zu-Ende Verbindungen untersucht. Hierzu zählen die Duplexfähigkeit, Flusskontrolle und Verschlüsselung. Außerdem werden Kommunikationstechnologien für *Java* untersucht, vor allem im Hinblick auf deren Codierungsleistung. Abschließend wird das Pipelining-Verfahren vorgestellt, welches in der Lage ist, die durch die Technologien gegebenen Eigenschaften gut auszunutzen und dem Programmierer eine Schnittstelle auf der Anwendungsschicht anzubieten.

Der TCP/IP-Protokollstapel, auch Internetprotokollstapel genannt, ist ein vierschichtiges Modell, welches eine Spezialisierung des allgemeinen 7-schichtigen OSI/ISO Netzwerkprotokollstapels darstellt [Com06]. Er setzt sich zusammen aus Netzzugangsschicht, Internet-Schicht, Transportschicht und der Anwendungsschicht. Abbildung 4.1 verdeutlicht das Schichtenmodell und nennt Beispiele für Protokolle der jeweiligen Schicht. Jede Schicht ist dabei so ausgelegt, dass sie auf Diensten und Schnittstellen der darunter liegenden Schicht aufbaut und selbst auch Dienste für die darüber liegende Schicht anbietet. Dies soll dafür sorgen, dass man verschiedene Implementierungen einer Schicht einfach gegeneinander austauschen

OSI-Schicht	TCP/IP-Schicht	Beispiele
Anwendungen (7)	Anwendungen	HTTP, FTP, POP, SOAP, CORBA, RMI
Darstellung (6)		
Sitzung (5)		SOCKS
Transport (4)	Transport	TCP, UDP
Vermittlung (3)	Internet	IP (IPv4, IPv6)
Sicherung (2)	Netzzugang	Ethernet, Token Bus, Token Ring, 802.11, W-CDMA
Bitübertragung (1)		

Abbildung 4.1: OSI/ISO und korrespondierendes TCP/IP-Schichtenmodell.



(vgl. Onlineressource `http://www.linux-praxis.de/lpic1/images/routing.png` (abgerufen am 21.5.2014))

Abbildung 4.2: Beispiel einer Ende-zu-Ende Verbindung im TCP/IP-Schichtenmodell.

kann. Auf der Netzzugangsschicht sind unter anderem Implementierungen für kabelgebundene Technologien wie Ethernet und für kabellose Technologien wie WLAN verfügbar. Nachfolgend wird ein Überblick über die Aufgaben der einzelnen Schichten gegeben. Für weiterführende Informationen sei auf [Com06, S. 159ff.] verwiesen[1].

Netzzugang Auf der Netzzugangsschicht, manchmal auch MAC-Schicht[2] genannt, werden Verbindungen eines physikalischen Netzwerks aufgebaut. In einem physikalischen Netzwerk befinden sich üblicherweise Knoten, die über ein Kabel miteinander verbunden sind oder die beispielsweise via WLAN nach dem Standard 802.11[3], Ethernet oder UMTS verbunden sind. Auf dieser Schicht können keine Kommunikationsverbindung über das physikalische Netzwerk hinaus aufgebaut werden.

Internet Um auch über Grenzen physikalischer Netzwerke hinweg zu kommunizieren, führt die Internet-Schicht ein allgemeines Paketvermittlungsverfahren ein und weist jedem Netzwerkgerät eine Adresse zu. Die Internet-Schicht wird auch als IP-Schicht bezeichnet und sorgt dafür, dass ein Datenpaket von einem Absender zu jedem beliebigen adressierbaren Empfänger im Internet gelangt.

[1] Alle Internet Standards sind im Original als *Request for Comments* (RFC) veröffentlicht. Einige wichtige RFCs sind beispielsweise RFC791 (IP), RFC1166 (IP-Adresse), RFC793 (TCP) und RFC2616 (HTTP 1.1). Alle RFC-Dokumente sind unter Onlineressource `http://tools.ietf.org/html/rfc791` verfügbar, wobei jeweils der gewünschte RFC in der URL substituiert werden muss. (abgerufen am 10.4.2014)

[2] *„Das Medium Access Control Protocol (MAC, Layer 2) hat [...] unter anderem folgende Aufgaben: Es regelt den Zugriff der Endgeräte auf das Übertragungsmedium. Jedem Datenpaket wird ein MAC-Header vorangestellt, der unter anderem die Adresse des Senders und Empfängers (MAC-Adressen) enthält.“* (siehe [Sau13, S. 311])

[3] 802.11 ist die Oberbezeichnung verschiedener IEEE-Standards, die seit dem Erscheinen als Weiterentwicklungen der WLAN-Technologie erschienen sind. Die wichtigsten Standards für das 2,4 GHz und das 5 GHz Frequenzband sind nachfolgenden aufgelistet. 802.11b (2,4 GHz, bis 11 Mbit/s) 802.11g (2,4 GHz, bis 54 Mbit/s) 802.11a (5 GHz, bis 54 Mbit/s) 802.11n (2,4 und 5 GHz, bis 600 Mbit/s) 802.11ac (5 GHz, bis 6,93 Gbit/s) (siehe [Sau13, S. 298])

Transport Die Transportschicht realisiert Dienste, die die IP-Schicht um nützliche Funktionen erweitern, wie etwa Mechanismen zum Trennen von Paketen verschiedener Anwendungen, Mechanismen zur besseren und fairen Ausnutzung der verfügbaren Bandbreite zwischen verschiedenen Verbindungen (Flusskontrolle) und Mechanismen zur Erkennung von Übertragungsfehlern. Vor allem das *Transmission Control Protocol* (TCP) liefert zuverlässige Ende-zu-Ende Verbindungen, die von Anwendungen verschiedenster Art genutzt werden können.

Anwendung Auf Anwendungsschicht sind Protokolle spezifiziert, die eher der Organisation und Adressierung der eigentlichen Nutzdaten dienen. Sie werden auch benötigt, um die korrekte Codierung von Daten sicherzustellen. Ein sehr bekanntes Protokoll auf dieser Schicht ist beispielsweise das HTTP-Protokoll.

Sendet ein Client eine Nachricht zu einem Server unter Verwendung einer konkreten Instanziierung dieses TCP/IP-Protokollstapels, so durchläuft die Nachricht alle vier Schichten zunächst auf Clientseite. Ein Beispiel für eine Ende-zu-Ende Verbindung zwischen Client und Server über zwei Netzwerkknoten (englisch *Gateways*) unter Verwendung des TCP/IP-Protokollstapels ist in Abbildung 4.2 dargestellt. Tatsächlich versendet wird das Paket auf der MAC-Schicht. Über verschiedene zwischengelagerte Netzwerkknoten gelangt die Nachricht schließlich zum Empfänger, wo sie erneut durch alle vier Schichten bis zur entsprechenden Anwendung weitergereicht wird. Auf jeder Schicht wird der Nachricht ein weiterer Header mit für diese Schicht spezifischen Informationen hinzugefügt. Die primäre Aufgabe der IP-Schicht ist beispielsweise das Hinzufügen der IP-Adressen von Empfänger und Absender der Nachricht. Auf dieser Schicht ist es demnach nicht möglich, eine Nachricht einer Anwendung zuzuordnen. Dazu muss der Header der Transportschicht analysiert werden. Verschiedene Zwischenknoten auf dem Weg der Nachricht bis zum Empfänger müssen das Nachrichtenpaket eventuell auch bis zu einer gewissen Schicht analysieren, um dessen Weiterleitung zu ermöglichen. Ein Gateway beispielsweise extrahiert die IP-Adresse, um den korrekten Ausgang für die Weiterleitung des Pakets zu bestimmen.

Neben dem TCP-Protokoll für zuverlässige Ende-zu-Ende Verbindungen im Internet gibt es auf Transportschicht auch das häufig eingesetzte *User Datagram Protocol* (UDP). UDP spezifiziert das zusammenhanglose Übertragen von einzelnen Datenpaketen. Dabei kommen weder Mechanismen zur folgerichtigen und vollständigen Übertragung noch Mechanismen zur Flusskontrolle zum Einsatz. Somit ist die erfolgreiche Übertragung von UDP-Paketen nicht gewährleistet. Durch den nicht notwendigen Verbindungsaufbau werden einzelne UDP-Pakete jedoch mit gegenüber TCP-Paketen erheblich verkürzter Verzögerung übertragen. Deshalb wird UDP oft für fehlertolerante Anwendungen wie Internet-Telefonie (*Voice-Over-IP*) verwendet.

4.1 Aufbau mobiler Kommunikationsnetze

Mobile Kommunikationsnetze bestehen üblicherweise aus einem oder mehreren sogenannten Zugangs-Netzwerken und aus einem Kern-Netzwerk, welches die Zugangs-Netzwerke untereinander verbindet. Über das Kern-Netzwerk werden auch mobile Datenverbindungen ins Internet aufgebaut. Die Verbindungen im Zugangs-Netzwerk sind dabei kabellos. *Ad-hoc*-Netzwerke, die zwischen einzelnen Mobilgeräten aufgebaut werden und ohne Netzzugangspunkt funktionieren, werden in dieser Arbeit nicht betrachtet. Daten werden also im Zugangs-

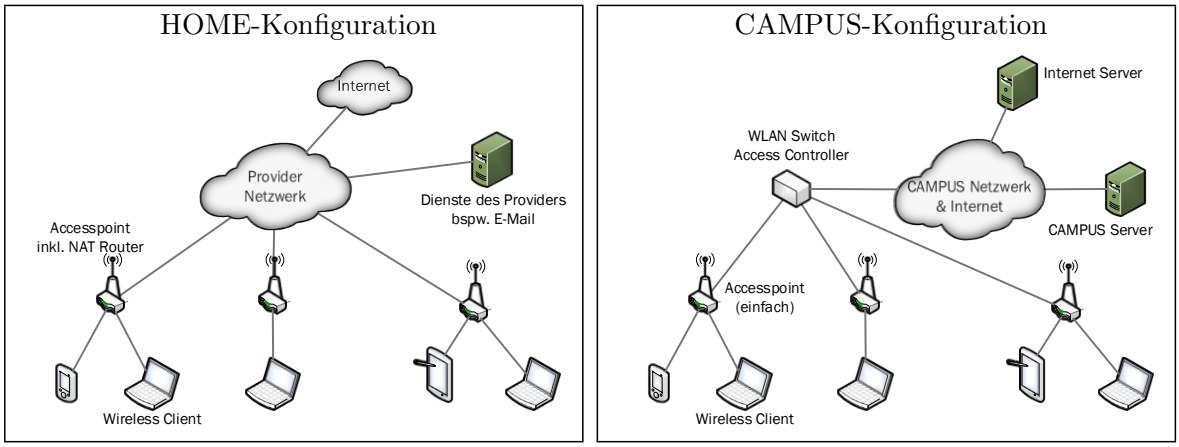


Abbildung 4.3: Konfigurationen verbreiteter WLAN Infrastrukturen. (vgl. Onlineressource `http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_9-3/wireless_lan_switches.html` (abgerufen am 23.1.2014))

Netzwerk über einen *Accesspoint* (AP) oder eine UMTS Antennenstation ins kabelgebundene Netzwerk eingeschleust. Im Vergleich zu kabelgebundenen Netzwerken treten durch die Funkübertragung vermehrt Störeinflüsse auf. Auch die Anzahl der Teilnehmer im Funknetzwerk beeinflusst Übertragungsrate und Latenz. Kurzum, die verfügbare Bandbreite und Latenz kann sich während der Dauer einer Übertragung stark ändern. Nachfolgend werden weitere Eigenschaften für WLAN und UMTS unterschieden.

Um den Einfluss der Eigenschaften verschiedener mobiler Netzwerktechnologien bewerten zu können, werden nachfolgend die Technologien WLAN und UMTS genauer betrachtet. Hierbei werden Eigenschaften wie Bandbreite, Latenz, Duplexfähigkeit oder Reichweite betrachtet. Darüber hinausgehend können weitere Details zu den verwendeten Netzwerktechnologien in [Sau13, Com06, KR08] nachgelesen werden. Für die nachfolgenden Untersuchungen wurden zwei WLAN- und eine Mobilfunkkonfiguration festgelegt. Deren Eigenschaften sind in Tabelle 4.1 zusammengefasst und werden im Text näher erläutert.

4.1.1 WLAN

WLAN ist eine sehr verbreitete kabellose Netzwerktechnologie für lokal begrenzte Areale, die nach dem IEEE-Standard 802.11 spezifiziert ist [20112]. WLAN wird häufig an privaten Breitbandanschlüssen eingesetzt, um die einzelnen Computer, Smartphones, etc. in einem Haushalt mit einer Internet-Verbindung zu versorgen. Diese HOME-Konfiguration ist in Abbildung 4.3 (links) zu sehen. Über das Kern-Netzwerk des Internet Providers, hier *Provider Netzwerk* genannt, wird jeder WLAN *Accesspoint*, hier inklusive *Router*, mit einem Internetanschluss versorgt, der an Mobilgeräte weitergereicht wird. In Abbildung 4.3 (rechts) ist hingegen eine CAMPUS-Konfiguration mit mehreren zu einem WLAN gehörenden *Accesspoints* abgebildet. Dies ist notwendig, um große Flächen, beispielsweise einen Universitäts-Campus, mit einem einheitlichen WLAN Netzwerk zu versorgen. Das Management der Clients ist hierbei aus den einfachen *Accesspoints* in einen *Access Controller* ausgelagert. Die Anbindung ins Internet erfolgt ebenfalls über diesen *Access Controller*. Bei der HOME-Konfiguration

Tabelle 4.1: Konfigurationen der untersuchten mobilen Kommunikationstechnologien

	HOME	CAMPUS	UMTS/HSPA
Zugangstechnologie	WLAN 802.11n	WLAN 802.11g	UMTS/HSPA
Zugangstechnologie Download/Upload	300 Mbit/s	54 Mbit/s	7,2 / 5,76 Mbit/s
Internetzugang Download/Upload	32 / 2 Mbit/s	>54 Mbit/s	7,2 / 5,76 Mbit/s

steht üblicherweise in den heute noch immer sehr stark verbreiteten IPv4[4] Netzwerken nur eine einzige IP-Adresse für alle Geräte hinter einem Breitbandanschluss zur Verfügung. Um dennoch mehrere Geräte anschließen zu können, wird ein NAT-*Router* zwischen *Accesspoint* und Breitbandanschluss geschaltet. Oft ist dieser schon wie in Abbildung 4.3 (links) mit dem *Accesspoint* kombiniert. Dieser NAT-*Router* weist allen Geräten IP-Adressen aus einem frei verfügbaren Pool von privaten IP-Adressen[5] zu. Da diese IP-Adresen im Internet keine Gültigkeit haben, übersetzt der *Router* dann via *Network Address Translation* (NAT) alle Pakete zwischen der vom Provider zugewiesenen IP-Adresse und der privaten IP-Adresse des entsprechenden Gerätes im WLAN-Netzwerk. Bei großen CAMPUS-Konfigurationen kann es auch sein, dass die Mobilgeräte Adressen aus dem Pool von öffentlichen Adressen der Institution zugewiesen bekommen.

Sehr verbreitet sind die Substandards 802.11g mit bis zu 54 Mbit/s und 802.11n, welches in der verfügbaren Konfiguration eine Übertragungsbandbreite von bis zu 300 Mbit/s erreicht. In der gesteten Konfiguration wurde für WLAN das 2,4 GHz Frequenzband verwendet. Die typischen, real erreichbaren Übertragungsraten einer ungestörten WLAN-Verbindung unter Benutzung einer TCP-Netzwerkverbindnung liegen üblicherweise weiter unterhalb des theoretischen Maximums. Beispielsweise wurde für die 802.11g-Verbindung der CAMPUS-Konfiguration eine maximale Bandbreitenausnutzung von ca. 12 Mbit/s gemessen. Tabelle 4.2 fasst einige gemessene maximale Übertragungsgeschwindigkeit für die verschiedenen Konfigurationen zusammen. Die typische Umlaufzeit (englisch *Round Trip Time* (RTT)) für kleine Nachrichten liegt für 802.11g bei 13-14 ms. WLAN 802.11n liefert zwar durchweg bessere Performance, bleibt jedoch auch sehr weit vom theoretischen Maximum entfernt. Die Gegenstelle für die Messungen wurde im Campus-Netzwerk der Universität Bayreuth und für WLAN 802.11n (HOME-Konfiguration) in unmittelbarer Nähe am WLAN Acesspoint platziert, sodass der Einfluss von Störsignalen minimiert wurde.

Die geringe Bandbreitenausnutzung bei WLAN liegt begründet im Zugriffsverfahren der

[4] *„Es [IPv4] war die erste Version des Internet Protocols, welche weltweit verbreitet und eingesetzt wurde, und bildet eine wichtige technische Grundlage des Internets. Es wurde in RFC 791 im Jahr 1981 definiert. IPv4 benutzt 32-Bit-Adressen, daher sind maximal 4.294.967.296 eindeutige Adressen möglich.“* (Onlineressource `http://de.wikipedia.org/wiki/IPv4` (abgerufen am 23.1.2014)) *„ Das Internet Protocol Version 6 (IPv6), früher auch Internet Protocol next Generation (IPnG) genannt, ist ein von der Internet Engineering Task Force (IETF) seit 1998 standardisiertes Verfahren zur Übertragung von Daten in paketvermittelnden Rechnernetzen, insbesondere dem Internet. [...] Im Internet soll IPv6 in den nächsten Jahren die gegenwärtig noch überwiegend genutzte Version 4 des Internet Protocols ablösen, da es eine deutlich größere Zahl möglicher Adressen bietet, die bei IPv4 zu erschöpfen drohen. [...] IPv6-Adressen sind 128 Bit lang.“* (Onlineressource `http://de.wikipedia.org/wiki/IPv6` (abgerufen am 23.1.2014))

[5] Private IP-Adressen können innerhalb privater Netze verwendet werden. Es handelt sich um IP-Adressbereiche, die im Internet nicht geroutet werden und deshalb nur Gültigkeit in lokalen Netzwerken haben. Private IP-Adressen können aus folgenden Bereichen benutzt werden: 10.0.0.0 bis 10.255.255.255, 172.16.0.0 bis 172.31.255.255 und 192.168.0.0 bis 192.168.255.255 (siehe [RMK+96])

Tabelle 4.2: Gemessene Eigenschaften verschiedener mobiler Kommunikationstechnologien

	Dell Streak 7 Tablet		SONY Xperia P Smartphone	
	max. Übertragungsrate	RTT Ø	max. Übertragungsrate Ø	RTT Ø
WLAN 802.11g	11,74 Mbit/s	ca. 13 ms	9,81 Mbit/s	ca. 14 ms
WLAN 802.11n	31,05 Mbit/s	ca. 6 ms	25,99 Mbit/s	ca. 6 ms
UMTS/HSPA	Down:2,88 Mbit/s Up:1,55 Mbit/s	ca. 92 ms	Down:4,35 Up:3,97 Mbit/s	ca. 85 ms

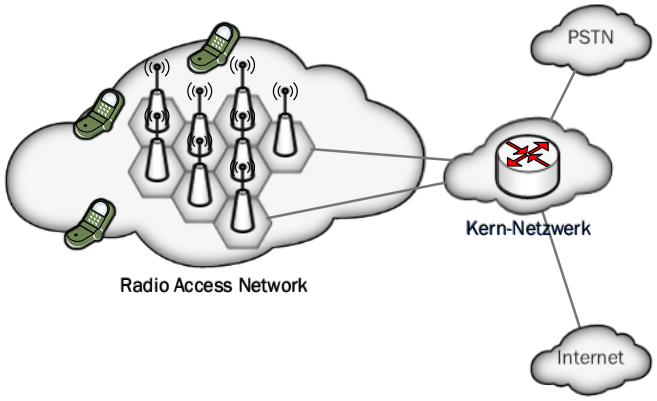
WLAN-Clients auf die Luftschnittstelle. WLAN verwendet den *Carrier Sense Multiple Access/Collision Avoidance* (CSMA/CA)-Mechanismus [Sau13, S.311 ff.]. Dabei überwacht ein Endgerät zunächst die Luftschnittstelle und sendet erst, wenn diese für eine gewisse Zeit frei war. Der Empfänger muss den Erhalt der Nachricht explizit bestätigen. Dennoch kann es vorkommen, dass mehrere Geräte gleichzeitig senden und die übertragenen Daten dadurch nicht mehr empfangbar sind. Durch das Abwarten und den möglichen Fehlerfall kommt die erhöhte Latenz zustande und es reduziert sich so auch die nutzbare Bandbreite. Voll-Duplex-Betrieb ist in WLAN Netzwerken zwar prinzipiell möglich und auch standardisiert, jedoch sind dazu zwei unabhängige Übertragungsfrequenzen notwendig. Da dies technisch wegen der doppelten Anzahl Antennen nur sehr teuer zu realisieren ist und zudem die Fequenzüberschneidung beispielsweise in Mehrfamilienhäusern ohnehin schon sehr hoch ist, wird das Voll-Duplex-Verfahren quasi nicht implementiert. Es kann also stets nur empfangen oder gesendet werden.

In WLAN Netzwerken wird die verfügbare Bandbreite zwischen Mobilgerät und Accespoint je nach Signalstärke und Häufigkeit der Störsignale entsprechend in verschiedenen Stufen ständig neu ausgehandelt. In WLAN Netzwerken können selbstverständlich auch UDP-Pakete versendet und empfangen werden. Das Empfangen von TCP- und UDP-Verbindungen ist jedoch oft bei der HOME-Konfiguration eingeschränkt, da der dort verwendete NAT-*Router* nicht erkennt, welchem Gerät er die eingehende Verbindung zuordnen soll. Man spricht deshalb auch manchmal von einer NAT-Firewall. In der CAMPUS Konfiguration ist es zusätzlich oft so, dass UDP-Pakete nicht weitergeleitet werden, um ressourcenhungrige UDP-Dienste wie *Voice-over-IP* zu unterbinden. Die UDP-Fähigkeit von WLAN-Netzwerken ist nur vom Willen des Providers abhängig, jedoch unterbinden gerade öffentliche Provider UDP Traffic, beispielsweise in Bahnhöfen, Flughäfen, etc.

4.1.2 UMTS

UMTS ist ein Mobilfunkstandard der 3. Generation (3G). Nach Standardisierung durch das *Europäische Institut für Telekommunikationsnormen* (ETSI) wurde der Standard von der *Internationalen Fernmeldeunion* (ITU) als einer der drei Mobilfunkstandards der 3. Generation ausgewählt. Heute wird der Standard vom *3rd Generation Partnership Project* (3GPP) gepflegt und weiterentwickelt[6]. Seit 2004 ist UMTS in Deutschland mit einer Datenübertragungsrate von 384 kbit/s verfügbar. Als wichtige Weiterentwicklungen sind die Erweiterungen

[6] Im 3GPP partizipieren Standardisierungsorganisationen und Parxispartner aus der ganzen Welt. Das 3GPP veröffentlicht in regelmäßigen Abständen Weiterentwicklungen der Mobilfunkstandards in *Releases*. UMTS wurde erstmalig im *Release 99* aus dem Jahre 2000 standardisiert. Aktuell ist *Release 11* aus dem Jahr 2012, welches der achte Nachfolger von *Release 99* ist. (siehe Onlineressource `http://www.3gpp.org/` (abgerufen am 14.4.2014))



PSTN = Public Switched Telephone Network (Netzwerk für öffentliche Telefonie)

Abbildung 4.4: Konfiguration einer 3G Infrastruktur (UMTS Mobilfunknetz). (vgl. Onlineressource `http://www.altera.com/end-markets/wireless/cellular/wir-cellular-infrastructure.html` (abgerufen am 23.1.2014))

High Speed Downlink Packet Access (HSDPA) und *High Speed Uplink Packet Access* (HSUPA) zu nennen, zusammen HSPA, die die Übertragungsraten auf 14,4 Mbit/s im Downstream und 5,76 Mbit/s im Upstream erhöht haben. Außerdem hat sich durch HSPA gegenüber UMTS auch die RTT im Durchschnitt auf unter 100 ms verringert. Dennoch ist die RTT gegenüber WLAN sehr viel höher. Tabelle 4.2 zeigt Messungen für UMTS/HSPA im Vergleich zu WLAN. HSPA erreicht hierbei bis zu 4,35 Mbit/s im Download und 3,97 Mbit/s im Upload. Da sich bei den verwendeten Testgeräten HSPA nicht mehr deaktivieren lässt, konnten leider keine Messungen mit dem UMTS-Standard gemacht werden. UMTS liefert jedoch stets mindestens 384 kbit/s im Download und 96 kbit/s im Upload. Für Vergleichsmessungen bezüglich UMTS sei auf [TLKF09] verwiesen. Weitere Verbesserungen für UMTS sind unter dem Namen HSPA+ standardisiert und bei einigen Anbietern in Deutschland bereits verfügbar [PH13]. Zudem befindet sich in Deutschland bereits auch das Mobilfunknetz der 4. Generation unter dem Namen *Long Term Evolution* (LTE) im Aufbau, welches nochmals verbesserte Datenraten und kürzere Latenzzeiten verspricht. HSPA+ und LTE werden in dieser Arbeit nicht betrachtet. Die Ergebnisse der Analyse der hier betrachteten Technologien sollte es jedoch ermöglichen, Parameter zu bestimmen, um zukünftig auch diese und weitere Technologien bezüglich ihrer Leistungsfähigkeit bewerten zu können.

UMTS ist ein flächendeckendes Mobilfunknetzwerk, welches aus Zellen besteht. Eine vereinfachte grafische Darstellung einer Mobilfunkkonfiguration ist in Abbildung 4.4 zu sehen. Jede Zelle wird dabei von einer festen Antenne versorgt und deckt ein großes Flächengebiet von mehreren Quadratkilometern ab. Die Komplexität der Netzwerke ist dabei jedoch höher als beispielsweise für WLAN, auch aufgrund der häufigen Erweiterungen, sodass sehr hohe Latenzen für mobile Datenpakete ins Internet entstehen [TLKF09, S.120f.]. Für die hier untersuchte UMTS/HSPA-Konfiguration (siehe Tabelle 4.1) wurde eine RTT von durchschnittlich 85 ms bestimmt (siehe Tabelle 4.2). Für UMTS sind sowohl das *Frequency Divisi-*

on Duplex (FDD) als auch das *Time Division Duplex* (TDD) Verfahren spezifiziert [Leh03, S. 67]. Bei FDD werden für Upload und Download verschiedene Frequenzen benutzt, was gleichzeitigen Upload und Download von Daten ermöglicht. Dieses Verfahren wird von allen deutschen Mobilfunkprovidern unterstützt. Da jedoch Upload und Download verschiedene maximale Übertragungsraten aufweisen, kann man nur von asynchronem Voll-Duplex sprechen. Beim TDD-Verfahren werden Download und Upload abwechselnd zeitlich getrennt auf einer einzigen Frequenz übertragen. Gegenüber beispielsweise WLAN sind Sendezeitpunkte und Sendedauer hier aber festgelegt. Das TDD Verfahren wird in Deutschland jedoch nicht implementiert. Es entspricht eher einem Half-Duplex-Verfahren. Im Gegensatz zum CSMA/CA-Verfahren des Zugriffs auf die Luftschnittstelle bei WLAN wird bei UMTS/H-SPA das WCDMA-Verfahren (*Wideband Code Division Multiple Access*) verwendet. Dabei handelt es sich um ein Codemultiplexverfahren, bei dem das gleichzeitige Senden mehrerer Nachricht durch verschiedene Sender möglich ist. Somit kann die ausgehandelte Datenrate einer UMTS/HSPA-Verbindung oft auch erreicht werden, da der aufwendige Kollisionserkennungsmechanismus wie bei WLAN nicht implementiert werden muss[7].

In Deutschland und auch vielen anderen Ländern Europas werden an UMTS Endgeräte nur private IP-Adressen ausgegeben. Die Geräte befinden sich also praktisch hinter einem NAT-*Router*, wenn sie Verbindungen ins Internet aufbauen. Dieser Umstand macht die Verwendung von UDP-Paketen zur Kommunikation unmöglich. Auch hier sind demnach nur TCP-Verbindungen möglich. Eine Verbindung zum Endgerät aus dem Internet ist nie möglich. Im Vergleich zu WLAN ist durch die erhöhte Reichweite auch eine wirkliche mobile Nutzung der Datenverbindung, also eine Nutzung im Freien, möglich. Durch die mobile Nutzung nehmen jedoch auch die Störeinflüsse zu, was die Verbindung instabiler macht und als Folge dessen auch in der verfügbaren Bandbreite einschränkt. Besonders kann sich dies auswirken, wenn ein Teilnehmer den Bereich einer Zelle verlässt und in eine neue Zelle hinein kommt. Hierbei kommt es zum *Handover* der Verbindung. Diese wird jedoch nicht, wie bei Sprachverbindungen üblich, blitzschnell übergeben, ohne dass der Teilnehmer davon Notiz nehmen würde. Da Datenverbindungen als nicht so wichtig eingestuft werden, werden diese abgebaut und für jede Antenne neu aufgebaut. Dieser *Handover* kann mehrere Sekunden, im Extremfall sogar bis zu 1 min dauern. Während des *Handover* können keine Daten transferiert werden. Es ist somit nicht ratsam, *Mobile Cloud-unterstützte Anwendungen* in der Bewegung via UMTS zu nutzen. Weitere Untersuchungen zur Nutzung von UMTS in der Bewegung sind beispielsweise in [TLKF09, S.117ff.] zu finden.

[7] *„Die entscheidende Neuerung von UMTS war [...] die Verwendung eines neuen Zugriffsverfahrens auf der Luftschnittstelle. Statt Frequenz- und Zeitmultiplex wie bei GSM verwendet UMTS ein Codemultiplex-Verfahren, um über eine Basisstation mit mehreren Benutzern gleichzeitig zu kommunizieren. Dieses Verfahren wird Code Division Multiple Access (CDMA) genannt. Im Unterschied zum Zeit- und Frequenzmultiplex von GSM senden hier alle Teilnehmer auf der gleichen Frequenz und zur gleichen Zeit. Die Daten jedes Teilnehmers werden dabei mit einem Code multipliziert, der möglichst große Unterschiede zu Codes aufweist, die von anderen Teilnehmern zur gleichen Zeit verwendet werden. Da alle Teilnehmer gleichzeitig senden, addieren sich alle Signale auf dem Übertragungsweg zur Basisstation. Die Basisstation kann jedoch die Daten der einzelnen Teilnehmer wieder aus dem empfangenen Signal herausrechnen, da ihr die Sendecodes der Teilnehmer bekannt sind.“* [Sau13, S. 143]

4.2 Eigenschaften kabelloser TCP-Verbindungen

Da in vielen öffentlichen Netzwerken keine UDP-Pakete zulässig sind, sollen im Folgenden ausschließlich TCP-Verbindungen betrachtet werden. Zudem wird in dieser Arbeit davon ausgegangen, dass sich das Mobilgerät nicht in einer schnellen Bewegung befindet, sondern während der Nutzung von *Mobilen Cloud-unterstützten Anwendungen* seinen Standpunkt höchstens langsam und um nur wenige Meter verändert. TCP ist ein Protokoll, welches in der Lage ist, Fehlübertragung von Daten zu erkennen und in gewissem Maße auch zu korrigieren. Dazu werden TCP-Verbindungen durch einen sogenannten 3-Wege Handshake zwischen zwei Kommunikationsteilnehmern etabliert. Es werden Sequenznummern vereinbart, die die korrekte und folgerichtige Übertragung von Datenpaketen sicherstellen. Ferner verwendet TCP auch eine Flusskontrolle, um die verfügbare Bandbreite möglichst gut auszunutzen, jedoch nicht zu überlasten. Mobile Kommunikationsnetzwerke stellen jedoch besondere Anforderungen an das TCP-Protokoll. Einerseits sind die auftretenden Latenzzeiten sehr hoch und andererseits kann sich die maximal verfügbare Bandbreite häufig und stark ändern. Auf die Auswirkungen dieser Phänomene auf die Performance der Endbenutzer-Applikation soll nachfolgend eingegangen werden. Danach wird der Einfluss von Verschlüsselungstechnologien untersucht, die für diese Arbeit eine obligatorische Anforderung darstellen, da sensible Daten wie Benutzernamen und Passwörter zur Authentifizierung und Abrechnung übertragen werden müssen.

Die Kommunikationszeit Δt_{comm}, die bei der entfernten Ausführung von Programmmodulen auftritt (siehe Gleichung (3.13)), wird von den nachfolgend genannten Faktoren beeinflusst, die in dieser Arbeit betrachtet werden:

- die Größe der zu übertragenden Daten (msg_size),
- die verfügbare Netzwerkperformance ($Throughput(msg\_size)$),
- die Verarbeitungszeit für Marshalling ($\Delta t_{marshall}$),
- und die für Verschlüsselung benötigte Zeit ($\Delta t_{encrypt}$).

Die nachfolgenden Gleichungen bringen die Einflussfaktoren in Zusammenhang. Dabei setzt sich die Dauer für einen einzelnen Datentransfer $\Delta t_{transfer}$ im hier verwendeten Modell wie in Gleichung (4.2) dargestellt zusammen. Ein Datentransfer vom Client zum Server wird als $\Delta t_{transfer_{upload}}$ bezeichnet. In entgegengesetzter Richtung wird er als $\Delta t_{transfer_{download}}$ bezeichnet. Zu beachten ist auch, dass die Nachrichtengröße msg_size durch Marshalling (marshall(...)) und die verwendeten Kommunikationsprotokolle (protocol(...)), insbesondere der Verschlüsselung, verändert werden kann. Die nachfolgenden Kapitel untersuchen jeweils eine der genannten Einflussgrößen näher.

$$\Delta t_{\text{comm}} = \Delta t_{transfer_{upload}} + \Delta t_{transfer_{download}}, \tag{4.1}$$

$$\Delta t_{\text{transfer}} = \frac{msg\_size}{Throughput(msg\_size)} + \Delta t_{marshall} + \Delta t_{encrypt}, \tag{4.2}$$

$$msg\_size = \text{protocol}(\text{marshall}(raw\,data)). \tag{4.3}$$

4.2.1 Analyse der Grundperformance

Wie bereits im Kapitel 4.1 festgestellt wurde, kann die technologiespezifische Bandbreitenangabe nicht zur realistischen Bewertung der tatsächlichen Übertragungsperformance genutzt

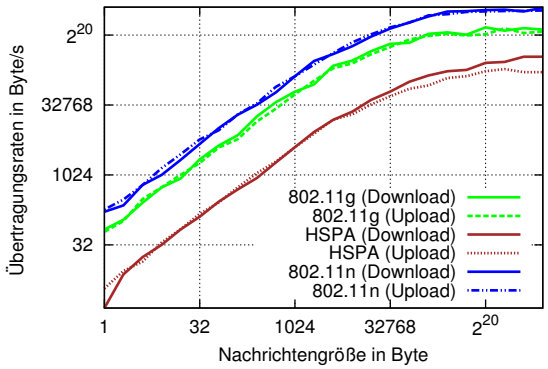


Abbildung 4.5: Kommunikationsperformance Dell Streak 7 Tablet

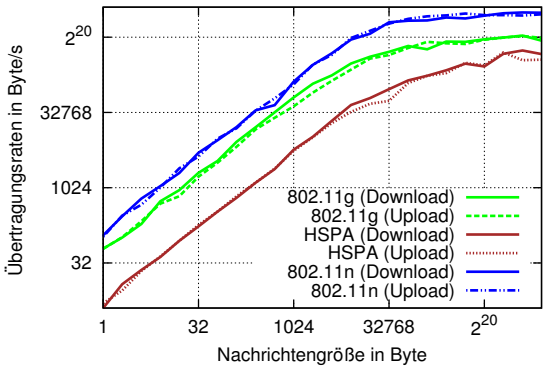


Abbildung 4.6: Kommunikationsperformance SONY Xperia P Smartphone

werden. In diesem Abschnitt wird untersucht, welchen Einfluss die Größe der Übertragungsdaten, die verfügbare Übertragungsgeschwindigkeit und die Netzwerklatenz auf die nutzbare Übertragungsrate $Throughput(msg\_size)$ haben. Dazu sind in den Abbildungen 4.5 und 4.6 die gemessenen durchschnittlichen Übertragungsraten für verschiedene Nachrichtengrößen unter Verwendung einer einzelnen TCP-Verbindung angegeben. Es wurden verschiedene Geräte für die clientseitigen Messungen genutzt. Es kamen ein Dell Streak 7 Tablet und ein Sony Xperia P Smartphone zum Einsatz. Beide Geräte verwenden *Android* als Betriebssystem. Eine genaue Beschreibung der Geräte ist in Anhang A zu finden. Alle Messungen, die Mobilfunk verwenden, wurden im deutschen Mobilfunknetz von E-Plus und unter Verwendung des Anbieters blau.de[8] durchgeführt. Bei dieser Konfiguration sind maximal 7,2 Mbit/s im Download und 5,76 Mbit/s im Upload verfügbar. Messungen im WLAN Netzwerk wurden aus max. 10 m Entfernung zum AP in Sichtweite ausgeführt. Die Gegenstelle für die Messungen wurde erneut im Campus-Netzwerk der Universität Bayreuth und für WLAN 802.11n (HOME-Konfiguration) in unmittelbarer Nähe am WLAN AP platziert.

Die Abbildungen 4.5 und 4.6 zeigen, dass die erreichbare Übertragungsrate von der Nachrichtengröße abhängt. Dies ist in Gleichung (4.2) durch die Funktion $Throughput(msg\_size)$ berücksichtigt. Dabei steigt die Übertragungsrate mit der Nachrichtengröße zunächst linear an, bis sie sich einem maximalen Wert nähert, auf dem die Übertragungsrate dann etwa gleich hoch bleibt. In den Abbildungen ist nicht dargestellt, dass die Übertragungsrate mit einem weiteren starken Anstieg der Nachrichtengröße wieder abfällt, da die Geräte dann zunehmend Probleme haben, die Datenmenge zu verarbeiten. Dieser Effekt konnte bereits ab ca. 128 MB Nachrichtengröße beobachtet werden. Er wird jedoch nicht näher betrachtet, da die Nachrichtengrößen für MCC-Anwendungen üblicherweise wesentlich kleiner sind. Bei beiden Geräten konnte eine sehr ähnliche Performance festgestellt werden, obwohl das Tablet eine etwas höhere durchschnittliche Übertragungsrate bei WLAN erreichte. Tabelle 4.2 fasst die maximalen Übertragungsraten und die durchschnittlichen Latenzen zusammen. Da die ser-

[8] *„Die blau Mobilfunk GmbH ist ein Mobilfunk-Discounter mit Sitz in Hamburg. Die Kernmarken sind der Discount-Tarif „blau.de“. Als Mobilfunkprovider nutzt blau Mobilfunk die Netzinfrastruktur von E-Plus und unterhält keine eigenen Filialen. [...] Die Netzabdeckung erreicht 2011 nach Angaben des Unternehmens deutschlandweit über 98% der Bevölkerung. Durch das UMTS-Netz können 2011 Verbindungen mit einer Geschwindigkeit von bis zu 384 kbit/s aufgebaut werden und mittels HSDPA als UMTS-Erweiterung bis zu 7,2 Mbit/s.“* (Onlineressource `http://de.wikipedia.org/wiki/Blau.de`(abgerufen am 24.1.2014))

verseitigen Komponenten bei den Messungen gleich waren, liegen die Unterschiede tatsächlich in den Geräten selbst begründet. Die geringeren Übertragungsraten für kleine Nachrichtengrößen sind einerseits im prozentual höheren Protokoll-Overhead für kleinere Nachrichten zu suchen. Andererseits werden sie auch von der TCP-Flusskontrolle und dem Handshake beim TCP-Verbindungsaufbau verursacht. Dies wird nachfolgend erläutert.

Ein hoher Protokoll-Overhead ensteht beispielsweise für sehr kleine Nutzdaten, da immer ein vollständiges TCP-Paket versendet werden muss, welches mindestens 20 Byte Headerdaten enthält. Zusätzlich kommen noch die Paketheader der unteren TCP/IP-Schichten hinzu, also IP-Header und MAC-Header. Dieser Overhead ist jedoch auf der Anwendungsschicht nicht messbar und relativiert sich mit größer werdenden Nutzdaten. Die geringste Protokolloverhead wird erreicht, wenn die Nachrichtengröße immer genau der maximalen Paketgröße entspricht. Diese maximale Paketgröße wird als MTU (*Maximum Transmission Unit*) bezeichnet und ist spezifisch für jedes physikalische Netzwerk. Die MTU wird üblicherweise für die Internet-Schicht angeben. Sie beträgt beispielsweise 2312 Byte für WLAN 802.11 [Mau04, S. 162 ff.]. Die Nutzdatengröße beträgt dann abzüglich der 20 Byte TCP-Header 2292 Byte. Für UMTS/HSPA Verbindungen wird in [20009] ein MTU-Werte von 1436 Byte angegeben. Wichtiger ist jedoch die sogenannte *Path MTU*. Diese beschreibt die kleinste MTU bei der Übertragung einer kompletten Ende-zu-Ende Verbindung. Da die MTU für jede physikalische Verbindung spezifisch ist und die meisten Verbindung ins Internet via Ethernet geleitet werden, beträgt die maximale MTU oft nur so viel wie die MTU für Ethernet, welche 1500 beträgt bzw. 1480 für die Anwendungsschicht unter Verwendung von TCP. Zwar können TCP-Pakete auch größer sein als die MTU, sie müssten dann jedoch auf IP-Schicht aufwendig fragmentiert und auf der Gegenseite wieder zusammengesetzt werden. Dies vermeidet man indem man die TCP-Paketgröße (*Maximum Segment Size* (MSS)) an die MTU anpasst.

Außerdem wird zum Beginn einer TCP-Verbindung ein 3-Wege Handshake durchgeführt. Bei diesem werden zwar kaum Daten übertragen, denn er dient nur zur Synchronisation von Client und Server, er kostet aber etwas Zeit, was sich auf die Gesamtübertragungsrate auswirkt. Für den 3-Wege Handshake muss mindestens die dreifache Latenz veranschlagt werden, da eine Anfrage, eine Bestätigung und deren Rückbestätigung zu übertragen sind. Somit ist der Verbindungsaufbau gerade in Netzwerken mit hoher Latenz sehr zeitaufwendig. Dies lässt sich gut in den Abbildungen 4.5 und 4.6 erkennen. Da für die UMTS/HSPA-Verbidnung die Latenz wesentlich höher ist, ist die Übertragungsrate für kleine Nachrichten geringer als für WLAN.

Schätzt man die Dauer für den Handshake mit der dreifachen Latenz ab und subtrahiert diesen Wert von der Übertragungszeit, erhält man unter Verwendung der MTU als Paketgröße für einzelne TCP-Datentransfers nur eine Übertragungsperformance $Throughput^*(msg\_size)$ zwischen 10-100 kB/s. Dieser Effekt wird durch die TCP-Flusskontrolle verursacht. Die Flusskontrolle in TCP dient dazu, eine potentiell langsamen Empfänger nicht zu überlasten, indem man die Datenrate bei Senden, ausgehend von einem kleinen Startwert, nur langsam erhöht. Dieses Verfahren wird als *slow start* bezeichnet. Dies hat gleichzeitig den Effekt, dass man auch eine schmalbandige Netzwerkverbindung nicht überlastet. Überlasten bedeutet hier, dass mehr Daten gesendet werden als in Puffern der beteiligten Geräte wie *Routern* oder dem Empfänger gespeichert werden können. Somit käme es zum Verlust von Paketen, die dann erneut übertragen werden müssten. Ziel ist es, mehrere Pakete direkt nacheinander zu senden, ohne zunächst die Bestätigung des Empfängers abzuwarten. Der Sender definiert also ein Fenster von Paketen, die gesendet werden. Den Erhalt aller Pakete bestätigt der Empfän-

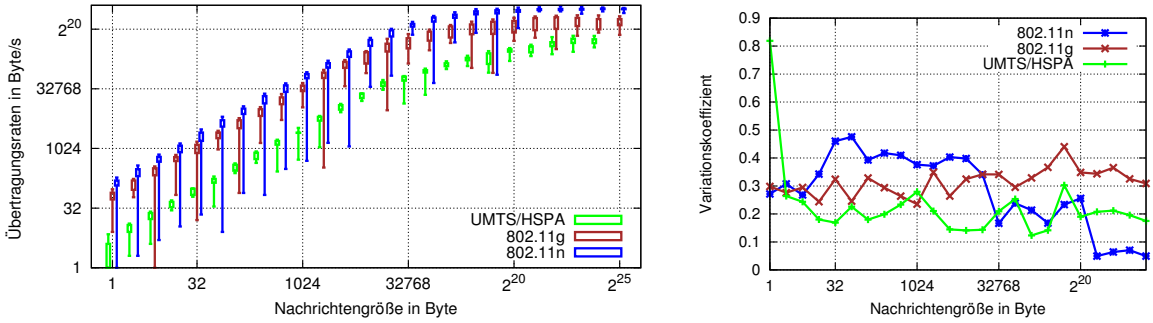


Abbildung 4.7: Verteilung (links) und Variationskoeffizient (rechts) mehrerer Performancemessungen verschiedener mobiler Kommunikationsverbindungen (Download).

ger dann eventuell mit nur einer Nachricht. Die Größe dieses Fensters, auch *congestion window* genannt, wird soweit erhöht, bis Pakete verloren gehen. Wenn Pakete verloren gehen, ist die maximale Übertragungsgeschwindigkeit erreicht. Danach können verschiedene Techniken angewendet werden, um den weiteren Verlauf der Kommunikation zu kontrollieren. Im Mobilbereich werden die Verfahren *TCP-Vegas* oder *TCP-Westwood* häufig eingesetzt [CFM04]. Das Fenster wird jedoch zunächst verkleinert, um wieder mehrere Pakete gemeinsam senden zu können. Das *congestion window* wird weiterhin ständig angepasst, um auf wechselnde Übertragungsleistung zu reagieren. Schwankungen können hierbei durch auftretende Störungen, beispielweise bei WLAN oder UMTS, ausgelöst werden. Außerdem kann auch die Anzahl von Netzwerk-Clients die jeweilige Übertragungsperformance einzelner Teilnehmer beeinflussen. Ein Überblick über verschiedene Mechanismen zur TCP-Flusskontrolle in kabellosen Netzwerken ist in [BDGS08] zu finden. Ein Performancevergleich zwischen verschiedenen Flusskontrollmechanismen unter Verwendung einer kabellosen Verbindung ist in [CFM04] zu finden. Darin sind unter anderem auch Experimente enthalten, die zeigen, dass die maximale Bandbreitenausnutzung einer TCP-Verbindung im Mobilbereich erst nach einigen Sekunden Verbindungsdauer erreicht wird. Somit bleibt die Erkenntnis, dass insbesondere einzelne kleine Nachrichtentransfers am besten über eine einzige TCP-Verbindung abgewickelt werden sollten, da die maximale Übertragungsbandbreite erst nach einer gewissen Zeit erreicht wird.

Um die Zuverlässigkeit und Störanfälligkeit der mobilen Kommunikationstechnologien besser bewerten zu können, wurden zusätzlich Messungen an drei Punkten durchgeführt. Dazu wurden je 30 Messungen für WLAN 802.11g, 802.11n und UMTS/ HSPA mit dem XPERIA Smartphone durchgeführt. Abbildung 4.7 (links) zeigt das Minimum, Maximum und den Interquartilsabstand ($\pm 25\,\%$ der Werte um den Median) der Messwerteverteilung. Die Mittelwerte der Messungen sind bereits als Kurve in Abbildung 4.6 dargestellt. Zur besseren Übersicht wurde in Abbildung 4.7 (links) nur der Download visualisiert. Es ist zu erkennen, dass für beide WLAN Standards gegenüber UMTS/HSPA deutlich häufiger Ausreißer nach unten vorkommen. Eine bessere Vergleichbarkeit der Ergebnisse ist in Abbildung 4.7 (rechts) gegeben, in der der Variationskoeffizient[9] der Messreihen abgebildet ist. Dieser ist ein relatives Maß und gibt an, wie stark die Messreihe um den Erwartungswert (hier wurde der Mittelwert

[9] Der Variationskoeffizient ist ein relatives Maß für die Schwankung einer Stichprobe/Messreihe X. Er ist wie folgt definiert: $VarK(X) = \frac{Standardabweichung(X)}{Erwartungswert(X)}$, wobei als Erwartungswert hier der Mittelwert gewählt wurde. (siehe [Koh05, S.82])

gewählt) schwankt. Hier lässt sich erkennen, dass UMTS/HSPA für kleine Nachrichten weniger stark schwankt als die WLAN Standards. WLAN 802.11n schwankt stark für kleine Nachrichten und wird besser mit steigender Nachrichtengröße. Das durchweg gute Abschneiden von UMTS/HSPA ist im Zugriffsverfahren auf die Luftschnittstelle begründet. Im Vergleich zu WLAN gibt es hier kaum Störeinflüsse durch andere Mobilfunkteilnehmer. Außerdem sind bei UMTS/HSPA die Sende- und Empfangsfrequenzen unter den Mobilfunkprovidern fest zugeteilt. Sie behindern sich nicht gegenseitig wie beispielsweise bei WLAN Netzwerken in Mehrfamilienhäusern. Insgesamt wird festgehalten, dass mit einer Schwankung von ca. 30 % um den gemessenen Wert der Übertragungsgeschwindigkeit gerechnet werden muss.

Aus der recht starken Schwankung auch bei zeitlich sehr nahen Messungen muss abgeleitet werden, dass die Verfügbarkeit der messbaren Maximalbandbreite eher pessimistisch bewertet werden muss. Zudem ist es sinnvoll, die maximale Bandbreite nicht durch eine einzelne Messung zu bestimmen, sondern über den Mittelwert von aufeinander folgenden Messungen.

4.2.2 Verschlüsselungsverfahren

Als Nebenproblemstellung wurde bereits ausgeführt, dass die Verschlüsselung der Übertragungsdaten im Cloud-Umfeld unerlässlich ist. Dies dient nicht nur dem Schutz von Benutzerdaten, sondern auch der Sicherheit des Abrechnungsmechanismus. Das Stehlen des *Sessionkeys* einer Cloud-Session könnte einen Angreifer in die Lage versetzen, auf Kosten des eigentlichen Benutzers dessen Cloud-Ressourcen unerlaubt zu nutzen. Das Verschlüsselungsverfahren hat Einfluss auf die Datengröße und es wird Zeit benötigt, um die Verschlüsselung durchzuführen (siehe Gleichungen (4.2) und (4.3)). Nachfolgend werden zwei gängige Verschlüsselungsverfahren für den Internet-Datenverkehr vorgestellt und deren Eignung für MCC wird analysiert. Die Analyse und Auswahl der Verfahren basiert dabei auf [Eck12] und kann dort noch detailierter nachgelesen werden.

SSL/TLS

Die heute von über 99 % aller verschlüsselten Websites unterstützten Protokolle *SSL 3.0* und *TLS 1.0* stammen aus den späten 1990er Jahren[10]. Ohne zu tief in die Details des *Secure Socket Layer* (SSL) und die *Transport Layer Security* (TLS) zu gehen, sollen hier dennoch die für die Übertragung wichtigen Eigenschaften etwas näher erläutert werden. Vor allem *TLS* gilt heute als Standard in der Verschlüsselung von Übertragungsdaten im Internet. Dies liegt nicht zuletzt auch an der transparenten Implementierung auf Basis von regulären TCP-Netzwerkverbindungen. Bibliotheken für die Umsetzung von SSL/TLS sind für nahezu alle Programmiersprachen vorhanden. Dabei sind diese Verschlüsselungsprotokolle eigentlich Protokolle der Anwendungsschicht, werden jedoch normalerweise alternativ zum regulären TCP-Protokoll angeboten. Der Funktionsumfang von SSL/TLS wird wie folgt zusammengefasst:

> „*Die Hauptaufgaben von SSL sind die Authentifikation der Kommunikationspartner unter Verwendung von asymmetrischen Verschlüsselungsverfahren und Zertifikaten, die vertrauliche Ende-zu-Ende-Datenübertragung unter Nutzung eines gemeinsamen Sitzungsschlüssels und schließlich auch die Sicherstellung der Integrität der transportierten Nachrichten unter Nutzung von Message Authentication Codes.*“[Eck12, S. 789]

[10] siehe Onlineressource `https://www.trustworthyinternet.org/ssl-pulse/` (abgerufen am 16.2.2014)

Das SSL-Verschlüsselungsverfahren umfasst das Handshake-Protokoll, welches dem Aushandeln der Verschlüsselungsalgorithmen und der Authentifizeriung der Kommunikationspartner dient, und das Record-Protokoll, welches dem verschlüsselten Transfer der eigentlichen Nutzdaten unter Verwendung eines Sessionschlüssels dient.

Die Authentifikation basiert dabei auf asymmetrischen *Public-Key*-Verschlüsselungsalgorithmen und der Verwendung von Zertifikaten. Asymmetrisch bedeutet, dass zum Verschlüsseln und Entschlüsseln verschiedene Schlüssel benutzt werden (siehe [Eck12, S. 415 ff.]). Jeder Kommunikationspartner besitzt einen öffentlichen und einen privaten Schlüssel. Zudem sichert die Verwendung von Zertifikaten auch die Authentizität der Daten. Mit einem öffentlichen Schlüssel wird ein Klartext in einen Geheimtext umgewandelt. Der Klartext kann mit dem zugehörigen geheimen Schlüssel wieder entschlüsselt werden. Der geheime Schlüssel muss geheim gehalten werden und es muss praktisch unmöglich sein, ihn aus dem öffentlichen Schlüssel zu berechnen. Der öffentliche Schlüssel muss jedem zugänglich sein, der eine verschlüsselte Nachricht an den Besitzer des geheimen Schlüssels senden will. Praktisch werden die Schlüssel mit Zertifikaten gespeichert. Ein heute oft noch verwendeter *Public-Key*-Verschlüsselungsalgorithmus ist der RSA (Rivest, Shamir und Adleman) Algorithmus. Zertifikate werden üblicherweise im X.509 Format bereitgestellt und selbst verschlüsselt, sodass man die entsprechende Zertifikatdatei nur mit einem bekannten Passwort entschlüsseln kann. Ein Zertifikat verbindet dabei einen öffentlichen Schlüssel (*Public Key*) mit einem Servernamе bzw. einer Server-IP-Adresse. Damit kann ein Empfänger den Absender einer Nachricht authentifizieren. Zertifikate allein sind jedoch nicht ausreichend, um einen Absender hinreichend sicher zu authentifizieren, denn der Absender kann sich ein Zertifikat üblicherweise selbst ausstellen. Um die Gültigkeit von Zertifikaten zu sichern, gibt es sogenannte *Trusted Authorities*, denen sowohl Client als auch Server vertrauen. Eine solche Infrastruktur wird als *Public-Key-Infrastruktur* (PKI) bezeichnet (siehe [Eck12, S. 415 ff.]). Eine *Trusted Authority* „unterschreibt“ die Zertifikate nach Vorlage und sichert somit, dass das vorgezeigte Zertifikat wirklich gültig ist. Üblicherweise wird im Internet-Kontext nur der Server authentifiziert. Eine Client-Authentifizierung ist zwar ebenso möglich, jedoch durch wechselnde IP-Adressen oft unpraktisch und wird deshalb selten angewendet.

Der eigentliche Verbindungsaufbau einer SSL/TLS verschlüsselten Verbindung beginnt, nachdem bereits eine TCP-Verbindung zwischen den Kommunikationspartnern aufgebaut wurde. Danach wird zunächst ein sogenannter SSL/TLS Handshake nach einem festgelegten Protokoll durchgeführt. Dies geschieht in vier Schritten [Eck12, S. 793 ff.].

1. In der Aushandlungsphase einigen sich Client und Server zunächst auf die zu benutzende TLS-Version. Anschließend wird eine *CipherSuite* ausgehandelt, welche ein Tupel aus Schlüsselaustauschalgorithmus, Verschlüsselungsalgorithmus, *Message Authentication Code* (MAC)[11] und einer Pseudozufallsfunktion ist. Anschließend wird je nach verwendeter *CipherSuite* ein gemeinsames sogenanntes *Master Secret* von Client und Server generiert, was dann zur Initialisierung des eigentlichen Verschlüsselungsalgorithmus für die Nutzdaten verwendet wird.

[11] *„Wollen zwei Kommunikationspartner Alice und Bob überprüfen können, dass die zwischen ihnen ausgetauschten Dokumente einen authentischen Ursprung besitzen, so können sie dazu einen Message Authentication Code (MAC) verwenden. [...] Ein Message Authentication Code ist somit eine Hashfunktion mit Einweg-Eigenschaften, die zusätzlich noch einen geheimen Schlüssel K verwendet. Der Schlüssel ist nur den beiden Kommunikationspartnern bekannt, die authentifizierte Dokumente austauschen möchten.“* [Eck12, S. 387]

2. Der Client sendet zunächst eine authentifizierte und verschlüsselte Nachricht zum Server, die dort entsprechend entschlüsselt wird. Dadurch wird die korrekte Aushandlung und Verwendung der Schlüssel verifiziert.

3. Gleiches wie in 2. wird auch vom Server an den Client gesendet.

4. War der Handshake erfolgreich, beginnt nun die eigentliche Übertragungsphase der Nutzdaten.

In SSL/TLS wird zur eigentlichen Verschlüsselung der Nutzdaten häufig eine Stromchiffre wie RC4[12] verwendet. Stromchiffren haben den Vorteil, dass sie Daten beliebiger Länge verschlüsseln können und somit kein Overhead für das eventuelle Auffüllen der Daten auf eine festgelegte Blocklänge wie bei Blockchiffren erforderlich ist.

Die Aushandlung und Verschlüsselung einer TCP-Netzwerkverbindung bringt auch einen gewissen Overhead mit sich. Zunächst wird durch das auf TCP aufgesetzte *TLS record protocol* ein geringer Overhead pro TCP-Paket erzeugt. Das TLS record protocol spezifiziert dabei 5 Byte zur Codierung des Verbindungszustands, der verwendeten Version und der Länge der Nutzdaten. Zusätzlich fallen 16-20 Byte für die MAC an. Insgesamt sind demnach pro TCP-Paket also 21-25 Byte an Overhead zu erwarten. Zwar können via TLS Pakete von bis zu 65 kB verwendet werden, jedoch sind die Pakete im mobilen Umfeld durch geringe MTUs oft auf ca. 1500 Byte begrenzt. Insgesamt ergibt sich ein minimaler Overhead von ca 1,7% pro Paket. Ein Zertifikat hat üblicherweise eine Größe von ca. 1 kB. Zusätzlich müssen weitere Schlüssel und Nachrichten ausgetauscht werden. Dies führt außerdem dazu, dass noch vor der Übertragung der eigentlichen Nutzdaten bereits ca. 2 kB an Daten für den Handshake ($msg_{Handshake}$) über die TCP-Verbindung übertragen werden müssen. Die Nachrichtengröße msg_size erhöht sich insgesamt für eine SSL-verschlüsselte TCP-Übertragung wie folgt in msg_size_{SSL}:

$$msg\_size_{SSL} = msg_{Handshake} + msg\_size + \left\lceil \frac{msg\_size}{\text{MTU}} \right\rceil \cdot 25. \tag{4.4}$$

Zudem kann vor allem für den Handshake ein recht hoher zeitlicher Overhead beobachtet werden. Dieser setzt sich zusammen aus Berechnungszeit für das *Master Secret*, der Übertragung der Daten in der Handshakephase selbst und dem Öffnen und Laden der Zertifikatdateien von der Festplatte. Abbildung 4.8 zeigt die Verschiebung der Durchsatzkurve nach unten bei Verwendung einer SSL-Verschlüsselung gegenüber einer unverschlüsselten Verbindung. Eine deutliche Beeinträchtigung ist jedoch nur für kleine Nachrichten festzustellen. Die SSL-Verschlüsselung beeinträchtigt die maximale Übertragungsrate nicht. Abbildung 4.9 visualisiert den gemessenen Overhead für den Verbindungsaufbau unter Verwendung von TLS gegenüber einem unverschlüsselten Verbindungsaufbau. Dabei ist zu beachten, dass die dargestellte Zeit den Verbindungsaufbau und zusätzlich die Zeit für eine nachfolgende Echo-Kommunikation mit 2 Byte Daten darstellt. Dies ist notwendig, da die Verschlüsselung nach dem Aufbau der TCP-Verbindung erst mit dem Senden der ersten Nachricht etabliert wird.

[12] *„RC4 is a stream cipher designed in 1987 by Ron Rivest for RSA Security. It is a variable key-size stream cipher with byte-oriented operations. The algorithm is based on the use of a random permutation. [...] Eight to sixteen machine operations are required per output byte, and the cipher can be expected to run very quickly in software.“* [Sta07, S. 45]

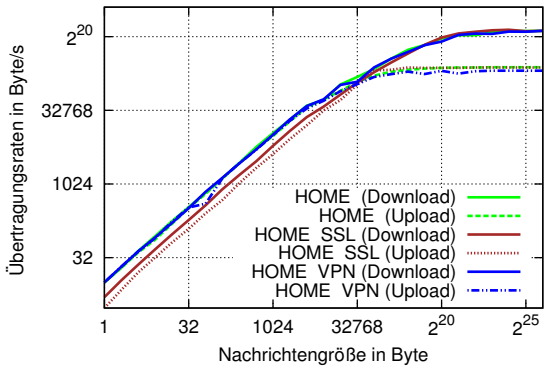


Abbildung 4.8: Mobile Kommunikationsperformance zu einem Server im Campus-Netzwerk der Universität Bayreuth aus der HOME-Konfiguration. (Testgerät Dell Streak 7)

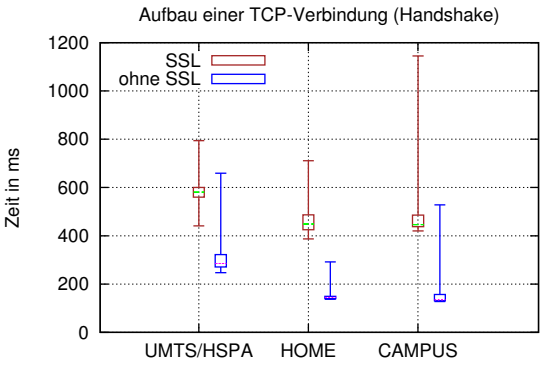


Abbildung 4.9: Dauer des TCP-Verbindungsaufbaus zur VM mit und ohne SSL-Verschlüsselung. (Testgerät Dell Streak 7)

Der Overhead $\Delta t_{encrypt}$ aus Gleichung (4.2) beträgt etwa 300-400 ms. Dieser zusätzliche Overhead beim Verbindungsaufbau zeigt ebenfalls die Vorteile der Wiederverwendung einer schon etablierten TCP-Verbindung.

Im Cloud-Umfeld hat die zertifikatsbasierte Authentifizierung jedoch auch einen Nachteil. Da Cloud-Ressourcen üblicherweise einer dynamischen Lastbalancierung und Skalierung unterliegen, ändern sich die IP-Adressen der Ressourcen häufig. Da jedoch ein Zertifikat stets einem konkreten Server zugewiesen ist, kann eine solch restriktive Vergabe der Zertifikate nicht erfolgen. Es müsste sonst für jede mögliche IP-Adresse ein Zertifikat erzeugt und zur Laufzeit auf dem Server installiert werden. Um dieses Problem zu lösen, werden oft sogenannte Wildcards eingesetzt. Dabei weist man ein Zertifikat einer Menge von Servern zu, für die das Zertifikat dann gültig ist. Verzichtet man auf die Authentizitätsprüfung des Zertifiktas durch eine *Trusted Authority*, kann man Zertifikate zur Laufzeit auch selbst ausstellen und unterschreiben, um das Problem wechselnder IP-Adressen zu umgehen.

Virtuelles Privates Netzwerk (VPN)

Ein virtuelles privates Netzwerk (VPN) stellt über die reine Kommunikationsverschlüsselung hinaus zusätzlich noch die Möglichkeit bereit, Clients direkt in ein entferntes Subnetz zu integrieren. Es wird ein virtuelles Subnetz erstellt, welches es Rechnern aus verschiedenen Netzen erlaubt, so zu kommunizieren, als wären sie lokal über ein Subnetz verbunden. Anwendung findet dieses Verfahren beispielsweise, um Laptops von Außendienstmitarbeitern Zugriff auf eigentlich nur lokal verfügbare Unternehmensdienste wie E-Mail zu ermöglichen. Dazu wird durch den Client eine Verbindung zu einem VPN-Gateway aufgebaut, bei dem sich der Client bereits authentifizieren muss. Danach bekommt der Client eine virtuelle IP-Adresse aus dem entsprechenden Subnetz zugewiesen. Diese virtuelle Verbindung wird auf Clientseite durch einen entsprechenden Netzwerktreiber oder eine Kernel-Erweiterung etabliert. Dadurch funktioniert sie für Anwendungen im Allgemeinen transparent. Die Einwahl in ein VPN-Subnetz hat gegenüber der SSL/TLS-Implementierung den Vorteil, dass Server innerhalb des Sub-

netzes üblicherweise keine weitere Verschlüsselung und Authentifizierung mehr benötigen. Für das Cloud-Umfeld ergibt sich hier der Vorteil, dass eine einmal etablierte Verbindung zur Cloud via VPN für unterschiedliche Serveranfragen genutzt werden kann, ohne dabei die Verschlüsselung der Kommunikation jeweils immer neu initialisieren zu müssen.

Es gibt zwei Arten von VPN-Verbindungen in Bezug auf deren technische Realisierung. Die bereits seit langem verfügbare Variante ist die auf TCP und SSL/TLS aufsetzende VPN-Verbindung. Hierzu wird über einen Treiber auf Clientseite ein virtuelles Netzwerkinterface etabliert, welches die Daten über eine reguläre verschlüsselte TCP-Verbindung zu einem VPN-Gateway überträgt. Die Variante impliziert die gleichen Eigenschaften bezüglich des Overheads wie im vorherigen Abschnitt beschrieben. Diese Variante arbeitet auf der Transportschicht im TCP/IP-Protokollstapel. Da das Etablieren der Verschlüsselung bei VPN bereits vor dem Aufbau der eigentlichen Datenverbindung erfolgt ist, wirkt sich dieser Overhead auch nicht auf die Übertragungsrate aus. Abbildung 4.8 zeigt die gemessene Übertragungsrate für eine SSL-basierte VPN-Verbindung.

Eine andere Variante setzt direkt auf der IP-Schicht auf, IPsec. Hierbei werden direkt IP-Pakete verschlüsselt übertragen. Dabei wird das sogenannte *IP Tunneling* angewandt, bei dem IP-Pakete komplett verschlüsselt und in ein neues IP-Paket abgelegt werden. Dies eröffnet eine größere Flexibilität, da Routing und andere Mechanismen von der VPN-Verbindung keine Notiz nehmen. Da IPsec jedoch auf einer sehr niedrigen Protokollebene angesiedelt ist, bedarf es für die Umsetzung üblicherweise Kernel-Unterstützung oder den Workaround über die Anpassung der betriebssystemspezifischen Netzwerkimplementierung. Der Overhead für IPsec Header ist nicht immer gleich. Da IPsec Blockchiffren verwendet, die nur blockweise verschlüsseln können, kommen zum eigentlichen Header eventuell noch Fülldaten hinzu, um auf die benötigte Länge aufzufüllen. Unter Verwendung einer 3DES-64bit[13] Blockchiffre beträgt der Overhead zwischen 50-57 Byte. Zudem muss man noch einen neuen IP-Header hinzu addieren, da der ursprüngliche mit verschlüsselt wird. Mit dem IP Header, der eine Mindestlänge von 20 Byte aufweist, kann so ein Overhead von 70 Byte und mehr pro Paket entstehen. Dies ist ein Vielfaches des Overheads, den eine SSL/TLS Verschlüsselung verursacht. Ein weiterer Nachteil von IPsec ist, dass das zum Schlüsselaustausch verwendete *Internet-Key-Exchange-Protokoll* (IKE) den UDP Port 500 verwendet. Damit ist das Verfahren für MCC nur sehr bedingt anwendbar, da UDP im Mobilfunknetz oft nicht geroutet wird.

VPN-Verbindungen beider Ausprägungen sind für mobile Geräte oft auch nicht verfügbar. Dies liegt einerseits an fehlender Unterstützung durch die Hersteller. So benötigt man auf *Android*-Geräten zum Etablieren einer VPN-Verbindung Administratorrechte (root), die man regulär nicht hat. Erst ab *Android* Version 4.0 werden die wichtigsten VPN-Protokolle direkt unterstützt, ohne Administratorrechte haben zu müssen. Es liegt andererseits auch an der fehlenden Lizenzierung. Betreiber von VPN-Gateways von Cisco müssen beispielsweise gesonderte Lizenzen erwerben, um *Android*-Geräten eine Verbindung zu ermöglichen. Eine weitere Eigenart von VPN-Verbindungen ist, dass üblicherweise sämtlicher Datenverkehr zum Internet über die VPN-Verbindung geführt wird. Dies ist jedoch für MCC nicht gewollt und sogar kontraproduktiv, da jedes übertragene Byte zur Cloud abgerechnet wird und bezahlt werden muss. Clientseitig kann zwar ein sogenannter *Traffic-split* zur Aufspaltung in VPN-relevanten

[13] *„Der Data Encryption Standard (DES) ist eine Blockchiffre, die einen Eingabeblock von 64 Bit mit einem 64-Bit Schlüssel, von dem aber nur 56 Bit relevant sind, zu einem Ausgabeblock von 64 Bit verschlüsselt. [...] Meist wird heutzutage jedoch nicht mehr der reine DES eingesetzt, sondern der DES wird in abgewndelter Form, meist als 3DES mit mehreren Schlüsseln verwendet, um das Problem des zu kleinen Verschlüsselungsschlüssels des DES abzumildern [...] .“* [Eck12, S. 334]

und anderen Datenverkehr konfiguriert werden, jedoch nur, wenn die VPN-Implementierung dies auch zulässt.

Zusammenfassung Verschlüsselung
Eine SSL-verschlüsselte TCP-Verbindung ist aktuell die beste Variante, um eine sichere Verbindung im MCC-Umfeld zu realisieren. Sie bietet gegenüber einer VPN-Variante folgende Vor- und Nachteile:

Vorteile	Nachteile
• geringer Overhead für die Datenverschlüsselung im Vergleich zu IPsec • für Mobilgeräte und *Android* problemlos anwendbar	• Zertifikatmanagement im Cloud-Umfeld schwierig • hoher Overhead bei Aufbau für jede einzelne Verbindung

Somit wird im Weiteren die SSL/TLS Verschlüsselung der TCP-Verbindung selbst bevorzugt. Die hohe Startup-Zeit der SSL-Verbindung muss jedoch durch eine Wiederverwendung der Netzwerkverbindung relativiert werden. Für Zertifikate werden Wildcards verwendet, um wechselnde IP-Adressen zu unterstützen.

4.2.3 Duplexfähigkeit

Die Duplexfähigkeit beschreibt die Fähigkeit, Daten nicht nur vom Sender zum Empfänger, sondern auch in umgekehrter Richtung zu transferieren. Dabei sind im TCP/IP-Protokollstapel alle gängigen Protokolle, bis auf einige der MAC-Schicht, bereits duplexfähig. Die Duplexfähigkeit hat großen Einfluss auf die Datenübertragungsgeschwindigkeit, wenn Daten gleichzeitig in beide Richtungen übertragen werden. Im Gegensatz zu halb-duplex ist bei voll-duplex die Übertragung in beide Richtungen gleichzeitig möglich. Beim voll-duplex Verfahren sind mindestens zwei verschiedene Kanäle vorhanden, die gleichzeitig genutzt werden können. Beim halb-duplex Verfahren ist oft nur ein Medium vorhanden, welches abwechselnd für die Übertragung in die verschiedenen Richtungen genutzt wird. In Kapitel 4.1 wurde bereits festgestellt, dass UMTS das asynchrone voll-duplex Verfahren und WLAN das halb-duplex Verfahren unterstützen. In diesem Kapitel wird die Duplexfähigkeit von Ende-zu-Ende Verbindung zwischen Client und Server genauer untersucht.

Da ein Netzwerkpaket üblicherweise einen Weg durch mehrere verschiedene physikalische Netzwerktechnologien zwischen zwei Kommunikationspartnern geht, ist die Duplexfähigkeit einer Verbindung nicht einfach zu bestimmen. Im Allgemeinen kommt es hier darauf an, welcher der beteiligten Netzwerkabschnitte die geringste Bandbreite hat und wie dessen Duplex-Eigenschaften sind. Als Beispiel stelle man sich folgendes Szenario vor: Ein kabelgebundener DSL-Anschluss hat eine Bandbreite von 16 Mbit/s im Download und 1 Mbit/s im Upload. Diese Internet-Verbindung wird von einem Laptop via WLAN 802.11n mit bis zu 300 Mbit/s genutzt. Obwohl WLAN im Vergleich zum DSL-Anschluss im Regelfalle nur halb-duplex-fähig ist, wird die WLAN-Verbindung nicht zum Flaschenhals. Dies liegt daran, dass die schnelle WLAN-Verbindung auch im halb-duplex-Verfahren noch schneller Daten übertragen kann als der DSL-Anschluss im (asynchronen) voll-duplex-Verfahren. Da bei MCC-Anwendungen stets mehrere verschiedene physikalische Netzwerke zwischen Client und Server liegen, wurde

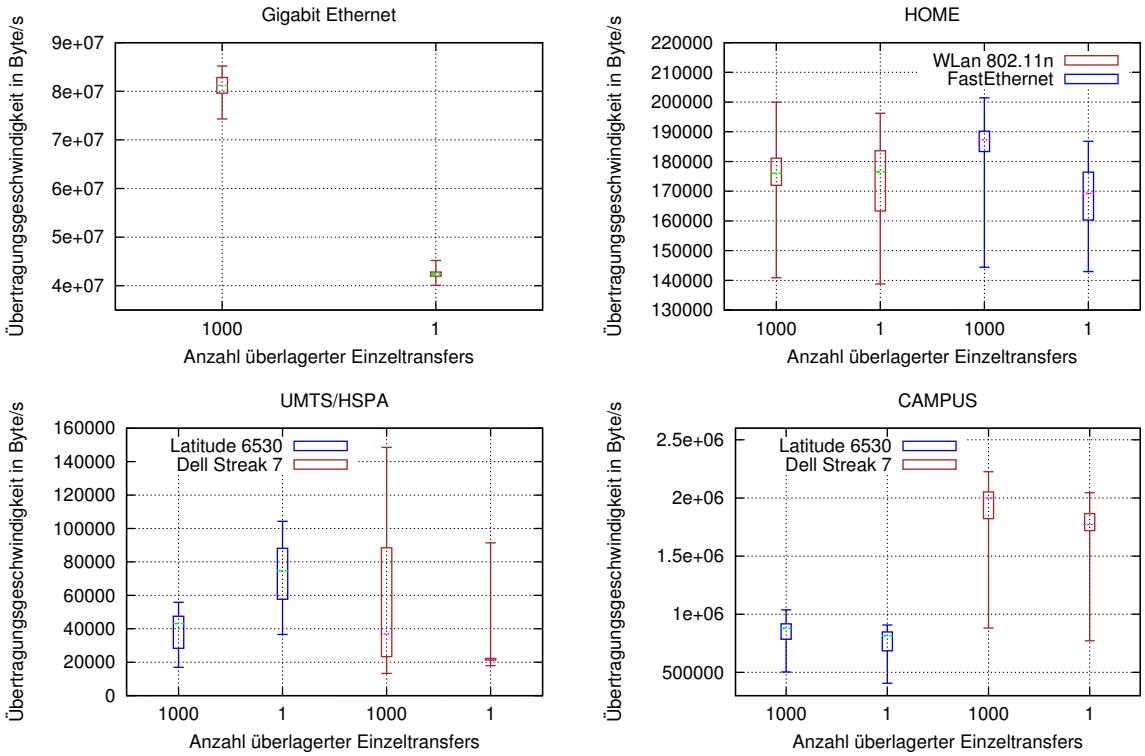


Abbildung 4.10: Verteilung der gemessenen Gesamtübertragungsgeschwindigkeiten von überlagertem Upload/Download Traffic im Vergleich zu einzelnen Request/Response-Verbindungen (Quartile). Es wurden jeweils 30 aufeinander folgende Messungen am gleichen Standort durchgeführt.

dieser Sachverhalt noch etwas genauer untersucht.

Die Duplexeigenschaften von Ende-zu-Ende Verbindungen wurden im Rahmen dieser Arbeit durch verschiedene Messungen bestimmt. Es wurden jeweils 30 Messungen mit großen Datenmengen zwischen zwei Endpunkten durchgeführt. Die Daten wurden zunächst zum Server übertragen und anschließend unverändert zurückgesendet. Dabei wurden die Daten einerseits nacheinander übertragen und in einem zweiten Test in Pakete aufgeteilt, die sofort nach Erhalt vom Server wieder zurückgeschickt wurden, ohne dabei auf weitere eingehende Daten zu warten. Gemessen wurde die resultierende Gesamtübertragungszeit, die sich im zweiten Test auf einer voll-duplex-fähigen Infrastruktur etwa halbieren sollte. Die Gesamtübertragungsgeschwindigkeit sollte sich beim voll-duplex-Verfahren etwa verdoppeln. Bei dem asynchronen voll-duplex-Verfahren ergibt sich ein Mittelwert, der üblicherweise oberhalb der geringeren Bandbreite (Minimum aus Upload- und Downloadbandbreite) liegt.

Abbildung 4.10 zeigt die gemessenen Übertragungsbandbreiten bei Überlagerung von Upload und Download im Vergleich zu sequentieller Kommunikation. Es sind neben dem Medianwert auch die minimalen und maximalen Messwerte angegeben sowie der Interquartilsabstand von $\pm 25\,\%$ der Messwerte um den Median. Daraus ist zu erkennen, wie die überlagerte Kommunikation in verschiedenen Umgebungen unterstützt wird. Eine Vergleichsmessung für über genau einen Switch verbundene Kommunikationspartner zeigt, dass die voll-duplex-Fähigkeit

der Gigabit Ethernet Verbindung gut ausgenutzt werden kann. Durch die überlagerte Übertragung von 1000 Paketen konnte die Übertragungsbandbreite im Vergleich zur Übertragung als einzelnes Paket nahezu verdoppelt werden. Es wurden je Messung 300 MB jeweils hin und zurück übertragen.

Eine weitere Messreihe wurde in der HOME-Konfiguration durchgeführt (siehe Tabelle 4.1). Hier wurden je 2 MB Daten zu einem Server im Campus-Netzwerk der Universität Bayreuth hin und zurück übertragen. Dabei war der Client einmal via FastEthernet mit dem Modem verbunden und einmal über einen WLAN-*Router* mit WLAN 802.11n und 300 Mbit/s Übertragungsrate. Als Client wurde hierbei ein Dell Latitude 6530 Laptop verwendet. Man kann in Abbildung 4.10 (oben rechts) erkennen, dass, obwohl WLAN in diesem Szenario nur halb-duplex-fähig ist, durch Überlagerung von Upload und Download trotzdem ein Geschwindigkeitszuwachs erreichbar ist. Für FastEthernet fällt dieser jedoch etwas höher aus. Bei dieser asynchronen Verbindung kann ohnehin nur ein Zuwachs von maximal 1/16 der Bandbreite (ca. 6%) erreicht werden, da nur der ohnehin schnelle Download mit dem Upload überlagert werden kann. Dieser machte jedoch auch im sequentiellen Fall nur 1/17 der Gesamtübertragungszeit aus. Somit kann die Bandbreite auf maximal 17/16 des sequentiellen Falls steigen. Für FastEthernet liegt der gemessene Zuwachs bei ca. 10% und für WLAN bei ca. 2%. Zu beachten ist auch, dass die geringen Zuwächse auch im Bereich der Schwankungsbandbreite der Verbindungen selbst liegen können.

Weitere Messungen für WLAN wurden in der CAMPUS-Konfiguration mit zwei verschiedenen Geräten durchgeführt. Als Gerät wurde erneut das Dell Streak 7 *Android* Tablet benutzt. Zusätzlich wurde auch der Dell Latitude 6530 Laptop benutzt. Der Server befand sich wie bei der Messung in der HOME-Konfiguration im Campus-Netzwerk der Universität Bayreuth. Deshalb waren Upload und Download-Datenrate bei dieser Messung gleichhoch. Es wurden 20 MB jeweils hin und zurück übertragen. Die Ergebnisse sind in Abbildung 4.10 (unten rechts) visualisiert. Die WLAN Verbindnung wies bei diesem Test die geringste Bandbreite auf der End-zu-Ende Verbindung zum Server auf. Die Maximalbandbreite war demnach durch die 802.11g-Verbindung limitiert. Trotzdem konnte durch die Überlagerung bei beiden Geräten eine Erhöhung der Datenübertragungsgeschwindigkeit gemessen werden (Latitude 6530: ca. 10% Streak 7: ca. 9%). Dies liegt wahrscheinlich in der besseren Ausnutzung der verfügbaren Bandbreite beim Senden durch den *Accesspoint* begründet. Dieser bestätigt nicht nur empfangene Pakete, sondern sendet auch gleich die Daten zurück (*Piggybacking*[14]), womit die Sendephasen des *Accesspoints* besser genutzt werden.

Abschließend wurden in der UMTS-Konfiguration auch Messungen zur Duplexfähigkeit im Mobilfunknetz durchgeführt (siehe Abbildung 4.10 unten links). Hierbei zeigte sich ein ganz unterschiedliches Bild zwischen den benutzen Mobilgeräten. Auf beiden Geräten betrug die Downloadbandbreite etwa das Zehnfache der Uploadbandbreite, wobei wir ca. 1,2 Mbit/s für den Download bei großen Datentransfers (>1MB) messen konnten. Obwohl bei UMTS durch das FDD-Verfahren ein simultaner Upload und Download möglich sein sollte, konnte dies nur bei dem Streak 7 gemessen werden. Hier konnte eine Steigerung von 133% und bei dem Dell Latitude eine Verminderung um 46% gemessen werden. Erwartet wurde durch die asynchrone Verbindung ein Zuwachs von 1/10. Die starken Unterschiede sind durch sehr stark schwanken-

[14] Piggybacking beschreibt in Bezug auf TCP die Möglichkeit, beim Bestätigen eines Pakets (ACK) zusätzlich auch Nutzdaten an das Bestätigungspaket anzuhängen, um so die Anzahl von Paketen einer Duplexverbindung zu reduzieren und damit die effektive Bandbreitenausnutzung zu maximieren. (siehe [KR08, S. 328])

de Messwerte begründet. Diese starke Schwankung trat beispielsweise beim Bestimmen der Maximalbandbreiten nicht auf (siehe Abbildung 4.7). Um dieses Problem weiter einzugrenzen, wurden erneute Messungen durchgeführt mit dem Ziel, Upload- und Downloadbandbreite gleichzeitig auszuschöpfen. Dazu wurden erneut 1000 überlagerte Datentransfers durchgeführt, bei denen jeweils das Zehnfache an Downloaddaten gegenüber den Uploaddaten übertragen wurde. Hier konnte tatsächlich bei beiden Geräten die Übertragungsdauer gegenüber einem Einzeltransfer verringert werden. Somit konnte so die Duplexfähigkeit nachgewiesen werden. Weiterhin konnte beobachtet werden, dass nachdem große und schnelle Downloads erfolgt waren, nachfolgende einzelne Request/Response-Verbindungen recht hohe Datenübertragungsgeschwindigkeiten aufwiesen, ehe sie dann allmählich auf ein konstantes niedriges Niveau absanken. Der Grund für diese Beobachtungen ist die oft als Problem beschriebene dynamische Bandbreitenanpassung bei HSPA [AZ06]. Dabei wird die pro Benutzer verfügbare Bandbreite für jedes Gerät in kurzen Abständen angepasst. Für große Downloads wird die Bandbreite entsprechend hochgesetzt. Bei den hier durchgeführten Request/Response-Anfragen können jedoch nur so viele Daten heruntergeladen werden wie auch hochgeladen wurden. Dies bedeutet für die getestete Konfiguration, dass der Upload stets ausgelastet ist, der Download aber nur zu etwa 10%. Somit wird dem Gerät die zugeteilte Bandbreite gekürzt. Dies führt dann unter Umständen dazu, dass die Downloadbandbreite so stark sinkt, dass auch durch Überlagerung kein Durchsatzgewinn erzielt werden kann. Eine Lösung dieses Problems ist kaum zu bewerkstelligen, ohne die Anpassungsroutinen des jeweiligen Mobilfunkproviders zu kennen. Außerdem hängt die Duplexfähigkeit auch von der Implementierung des UMTS-Modems ab. Im Durchschnitt ist aber die Überlagerungsstrategie meist nicht schlechter als die Einzeltransfers und es kann ein kleiner Durchsatzgewinn erreicht werden. Die Übertragungsbandbreite $Throughput_{max\,Duplex}$ für überlagerte Request/Response Kommunikation, mit der gleichzeitig in beide Richtungen Daten übertragen werden können ($Throughput_{max\,Download}$ und $Throughput_{max\,Upload}$), kann somit für diese Arbeit unter Benutzung von WLAN oder UMTS wie folgt abgeschätzt werden:

$$Throughput_{max\,Duplex} = MIN(Throughput_{max\,Download}, Throughput_{max\,Upload}). \quad (4.5)$$

4.2.4 Protokolle und Codierung auf Anwendungsschicht

Beim Übersenden von Daten via Netzwerk müssen Protokolle und Codierungen verwendet werden, die sowohl Client als auch Server verstehen, damit die Daten korrekt interpretiert werden können. Dies ist auf den unteren TCP/IP-Schichten ebenfalls notwendig und wird durch die bereits vorgestellten Protokolle implementiert. Die Codierung beschreibt einerseits den Datenaustausch und andererseits auch die Anordnung der Daten und Datenformate, die die eigentliche Nachricht ausmachen. Um die nachfolgend vorgestellten Technologien einordnen zu können, werden zunächst wichtige abstrakten Konzepte für Client/Server Kommunikation vorgestellt. Die Zusammenfassung basiert auf [Emm03, S. 93 ff.].

entfernter Prozeduraufruf / entfernte Objekte Bei dieser Art der Kommunikation handelt es sich meist um einen synchronen Nachrichtenaustausch. Das heißt, der anfragende Client blockiert so lange, bis er die Antwort vom Server erhalten hat. Dabei fasst man dieses Request/Response-Verhalten als eine Art Prozeduraufruf auf. Der Server stellt eine Methode zur Verfügung, empfängt Übergabeparameter und sendet den Rückgabewert der Methode zurück zum Client. Diese Art der Modellierung fügt sich sehr leicht

in das Programmierkonzept von prozeduralen Programmiersprachen ein. Eine Weiterentwicklung für objektorientierte Programmiersprachen ist im Konzept der entfernten Objekte zu sehen (beispielsweise *Java RMI*). Dabei wird auf Serverseite ein ganzes Objekt instanziiert, dessen Methoden dann entfernt durch den Client aufgerufen werden können. Die Verbindung zwischen dem Client und dem Objekt wird dabei über eine Session hergestellt. Oft wird dem Client vom Server eine abstrakte Interfacebeschreibung der verfügbaren Methoden und Objekte zur Verfügung gestellt. Diese Beschreibung nutzt der Client, um die entfernt verfügbare Funktionalität korrekt aufrufen zu können.

asynchroner Nachrichtenaustausch Beim asynchronen Nachrichtenaustausch wird keine festgelegte Request/Response-Reihenfolge der Nachrichten eingehalten. Nachrichten in beide Richtungen können jederzeit gesendet werden. Dieses Verfahren impliziert, dass jede Nachricht komplett selbstbeschreibend ist, da ihr kein Gegenstück zugeordnet werden kann. Asynchroner Nachrichtenaustausch kommt oft in Event-basierten Systemen zum Einsatz. Die Entkopplung von Request und Response kann beispielsweise sinnvoll zur taskbasierten, entfernten Ausführung eingesetzt werden. Unabhängige Tasks werden dabei zum Server übertragen, dort verarbeitet und die Ergebnisse werden in der Reihenfolge der Beendigung der Tasks zurückgesendet. Dabei können sich Anfrage- und Antwortnachrichten auch überlagern. Dieses Verfahren wird als Pipelining bezeichnet. Es kann sinnvoll eingesetzt werden, um die verfügbare Übertragungsbandbreite besser zu nutzen und eventuell hohe Netzwerklatenzen zu verstecken.

Bei Verwendung des entfernten Prozeduraufrufs machen sich hohe Netzwerklatenzen oft negativ bemerkbar, da auf das Eintreffen des Ergebnisses vom Server gewartet wird. Im Vergleich dazu ist es beim asynchronen Nachrichtenaustausch notwendig, dass die entfernt ausgeführten Funktionen keine bestimmte Reihenfolge voraussetzen. Oft kann das Verhalten synchroner Kommunikation (entfernter Prozeduraufruf) über einen asynchronen Nachrichtenaustausch emuliert werden, indem man zusammengehörige Request/Response-Nachrichten entsprechend markiert. Somit ist es im Rahmen dieser Arbeit ausreichend, zunächst nur den asynchronen Nachrichtenaustausch zu unterstützen.

Zudem unterscheidet man zwischen zustandsbehafteter und zustandsloser Kommunikation.

zustandsbehaftete Kommunikation Hierbei wird zwischen Client und Server ein Zustand gespeichert. Dieses Verfahren wird beispielsweise im Electronic Commerce angewendet, um den Fortschritt eines Online-Bestellvorgangs zu dokumentieren. Dabei gibt es unter anderem die Variante der expliziten Zustandsmanipulation über eine Session-Implementierung. Der übermittelte *Sessionkey* identifiziert dabei zusammengehörige Aufrufe. Man kann aber auch eine implizite Zustandsmanipulation über eine einzige TCP-Verbindung implementieren, da eine TCP-Verbindung bereits zusammengehörige Datenübertragungen identifiziert. Letzteres hat den Vorteil, dass kaum zusätzliches Zustandsmanagement beim Client und Server implementiert werden muss und der Lebenszyklus des Zustands automatisch mit dem Abbau der Verbindung endet. Ersteres hat wiederum den Vorteil, dass die Session auch beim Abbruch der Kommunikation durch einen Fehler zunächst nicht verloren geht.

zustandslose Kommunikation Hierbei wird serverseitig kein Zustand gespeichert und es wird im Allgemeinen für jede Anfrage eine neue Verbindung aufgebaut. Jede Anfrage liefert bei gleichen Übergabeparametern dasselbe Ergebnis zurück. Diese Art der Modellierung

erhöht die Skalierbarkeit einer entfernt verfügbaren Funktionalität, da der Server leicht repliziert werden kann.

Zur Abrechnung der Nutzung von Cloud-Ressourcen, muss deren Nutzungsdauer durch eine Zustandsimplementierung erfasst werden. Eine Analyse der mobilen Kommunikationstechnologien in Abschnitt 4.2 ergab, dass die Wiederverwendung von bereits etablierten TCP-Verbindungen angestrebt werden sollte. Somit ist eine zustandsbehaftete Kommunikation, die durch eine TCP-Verbindung reflektiert wird, zu bevorzugen.

Oft gehören das Kommunikationsprokoll und die Datencodierung auch zusammen. Nachfolgend werden einige populäre Technologien genannt und deren Eignung für MCC-Anwendungen wird diskutiert.

SOAP Das *Simple Object Access Protocol* (SOAP) [BEK+00] wurde für den standardisierten Datenaustausch zwischen Rechnern entwickelt, die verschiedene Hardware und Software benutzen und via Internet miteinander verbunden sind. Dabei basiert SOAP auf dem Austausch von XML-Nachrichten, die wiederum über ein Protokoll wie HTTP oder *Java Message Service* (JMS) übertragen werden können. SSL-verschlüsselte Verbindungen sind ebenfalls möglich. SOAP ist damit vielfältig einsetzbar und bietet eine Reihe von Erweiterungen, die aus dem stark verwandten Bereich der *Web Services* stammen und gut standardisiert sind. SOAP ist vor allem auch im *Business-to-Business* (B2B) Verkehr beheimatet. Da das Protokoll jedoch als relativ ineffizient gilt, unter anderem wegen der XML-Verarbeitung und dem daraus resultierenden hohen Protokoll-Overhead, wird es im Web zunehmend verdrängt von schlankeren Implementierungen. Die Größe einiger via SOAP übertragener Standarddatentypen ist in Tabelle 4.3 zusammengefasst. Alle namhaften *Java*-Applikationsserver und Entwicklungsumgebungen bieten Untersützung für die Entwicklung von SOAP *Web Services*. SOAP ist via der ksoap-Library auch für *Android* verfügbar. SOAP kann sowohl für den entfernten Prozeduraufrauf (RPC), wie auch für asynchrones Messaging verwendet werden.

RESTeasy/JSON Als eine Art Gegenbewegung zu SOAP *Web Services* sind seit einigen Jahren *RESTful Web Services* verbreitet (REST = *Representational State Transfer*) [Fie00]. Sie nutzen ausschließlich eine spezielle Untermenge des HTTP-Standards für den effizienten Datenaustausch aus. Zur Codierung der Nutzdaten können neben XML auch das *Javascript Object Notation* (JSON) Format [Cro06] und andere verwendet werden. Dabei ist JSON, wie auch XML, ein textuelles Format, welches jedoch auf die meisten Metainformationen verzichtet und somit wesentlich weniger Overhead als XML hat. *RESTful Web Services* sind zwar auch standardisiert, jedoch wird die Standardisierung nicht von vielen großen Unternehmen voran getrieben und somit ist auch die Werkzeugunterstützung für Entwicklung solcher *Web Services* geringer. REST ist dank HTTP auch für *Android* verfügbar und kann via SSL-verschlüsselter Verbindung genutzt werden. Zudem kann HTTP eine bereits aufgebaute TCP-Verbindung mehrfach verwenden. Es existieren auch Bibliotheken für die JSON-Codierung. *RESTful Web Services* arbeiten ressourcenorientiert. Eine Ressource kann dabei unter anderem ein Bild, ein Text oder auch ein Objekt sein. Somit kann REST auch für RPC-Aufrufe auf entfernten Objekten eingesetzt werden.

CORBA Die *Common Object Request Broker Architecture* (CORBA) beschreibt einen Standard, der wie SOAP die Kommunikation zwischen verschiedenen Rechnern und Software-

Stacks ermöglicht [Obj12]. CORBA 1.0 wurde bereits 1991 veröffentlicht und seither kontinuierlich weiterentwickelt. Im November 2012 wurde der bis dato aktuellste CORBA-Standard in der Version 3.3 beschlossen. Auch CORBA bietet verschiedene Erweiterung, wie etwa eine Service-Discovery, Services-Naming, Lastbalancierung und weitere. CORBA transferiert die Daten jedoch im Binärformat und erzeugt damit wesentlich kleinere Nachrichten als die bereits vorgestellten textuellen Formate. CORBA ist ursprünglich eher für Intranets entwickelt worden und verschiedene CORBA-Implementierungen sind nicht immer 100% kompatibel. CORBA verwendet eine eigene Interface-Beschreibung, die *Interface Definition Language* (IDL), aus der ein Client beispielsweise den Code für eine spezifische Programmiersprache erzeugen kann. CORBA implementiert das Konzept von verteilten Objekten. Eine bekannte CORBA-Implementierung für *Java*, JacORB[15], unterstützt sowohl SSL-Verschlüsselung als auch die Wiederverwendung einer TCP-Verbindung. Es gibt jedoch keine CORBA Implementierung, die für *Android* verfügbar ist.

RMI *Remote Method Invocation* (RMI) ist ein *Java*-spezifischer Standard, der ebenfalls das Konzept von *Remote Object* implementiert [ORA11]. Dabei arbeitet RMI, wie CORBA auch, binär. Eine Besonderheit ist hierbei, dass Marshalling-Zeiten fast vollständig wegfallen, da Objekte direkt aus dem Speicher versendet werden können. Wie auch CORBA verwendet RMI einen Namensdienst, der entfernte Objekte zwischen Client und Server vermittelt. Namensdienste können im Cloud-Umfeld sehr hilfreich sein, da trotz häufig wechselnder IP-Adressen der RMI-Server durch den Namensdienst mit fester IP immer ein Frontend zur Verfügung steht, welches Objekte nach außen vermitteln kann. RMI verwendet TCP-Verbindungen wieder und unterstützt SSL-Verschlüsselung. Leider sind die effizienten Protokolle CORBA und RMI für *Android*-Geräte nicht verfügbar. Dies liegt an den in der *Android* Systembibliothek fehlenden Klassen für *Java* RMI[16], die auch für eine CORBA-Implementierung benötigt werden.

Zusammenfassung

Die Wahl einer effizienten Codierung und eines effizienten Protokolls erscheint zunächst vielfältig. Tabelle 4.3 zeigt jedoch, dass nur binäre Codierungen den Anforderungen von MCC bezüglich Datenoverhead entsprechen können. Zudem sind die beiden sehr effizienten Technologien RMI und CORBA für *Android* nicht verfügbar. Die einzige Standardvariante, die zur Kommunikation im MCC-Umfeld tauglich ist, ist eine persistente HTTP-Verbindung, beispielsweise in Kombination mit *RESTful Web services*, und eine binäre Datencodierung als Byte-Datenstrom. Damit wird die Modellierung der Marshallingzeit $\Delta t_{marshall}$ und des Datenoverheads für Marshalling $marshall(msg\_size)$ für die Performancevorhersage der Netzwerkkommunikation obsolet. Es wird im Weiteren davon ausgegangen, dass die Übertragungs-

[15] Verfügbar unter Onlineressource `http://www.jacorb.org/` (abgerufen am 16.2.2014)

[16] Java-Bytecode wird auf einer JVM ausgeführt. Auf der Android Plattform wird hingegen die Dalvik VM benutzt. Diese führt modifizierten Bytecode aus, der nicht JVM-kompatibel ist. Dieser modifizierte Dalvik-Bytecode wird aus dem Java-Bytecode durch einen speziellen Compiler erzeugt. Da RMI Objektserialisierung verwendet, um Objekte auszutauschen, kann es zu Inkompatibilitäten zwischen Dalvik-Objekten und Java-Objekten kommen. Deshalb steht RMI für Android nicht zur Verfügung. (siehe Onlineressource `http://www.pcwelt.de/ratgeber/Unterschiede-zwischen-JVM-und-Dalvik-VM-Smartphone-Grundlagen-1005308.html`(abgerufen am 25.1.2014))

Tabelle 4.3: Nachrichtengrößen von 1.048.576 Werten verschiedenen Typs in verschiedenen Codierungen in MB

	Art	double	int	byte	String *
SOAP/Axis2	Text	54.59	35.51	1.42	1.06
RESTeasy/JSON	Text	19.61	7.08	1.01	1.03
RMI	binär	8.11	4.06	1.02	1.02
CORBA/JacORB	binär	8.05	4.06	1.02	1.02

* String mit 1.048.576 Zeichen.

daten bereits als Byte-Datenstrom vorliegen und dass somit durch die Kommunikationstechnologie kein weiteres Marshalling erfolgen muss.

4.2.5 Zusammenfassung Grundperformance

Die gemessenen Übertragungsbandbreiten lagen für Ende-zu-Ende Verbindungen in den drei Konfigurationen CAMPUS, HOME und UMTS/HSPA (siehe Tabelle 4.1) bei maximal ca. 10 Mbit/s und sind oft wesentlich niedriger. Diese geringen Werte machen es notwendig, die Kommunikation so effizient wie möglich zu gestalten. Dies wirkt sich vor allem auf die Codierung der Daten aus, die binär übertragen werden sollten. Als positiv hat sich herausgestellt, dass die meisten Verbindungsarten duplexfähig sind und somit Download und Upload gleichzeitig genutzt werden können. Ein sehr negativer Fakt ist der daten- und zeitintensive Aufbau einer verschlüsselten TCP-Verbindung. Für feingranulare Datenübertragungen sollte deshalb eine bestehende TCP-Verbindung wiederverwendet werden. Als einzig verfügbare Kommunikationstechnologie, die Wiederverwendung und binäre Datenübertragung unterstützt, kann HTTP verwendet werden. Aus diesem Grund werden HTTP und verwandte Technologien sowie verschiedene Implementierungen im folgenden Abschnitt genauer untersucht.

Ein etwas unerwartetes Ergebnis dieses Kapitels war, dass UMTS/HSPA-Verbindungen im Durchschnitt weniger störanfällig waren als WLAN-Verbindungen. Diese Beobachtung macht aber auch deutlich, dass die mobile Kommunikation im Freien der Performance von WLAN in Gebäuden ähnlich ist und somit können per WLAN genutzte Applikationen mit ähnlicher Geschwindigkeit auch im Freien genutzt werden. Aus Sicht der Kommunikationsperformance kann somit die Anforderung, vormals stationär genutzte Applikationen auch unterwegs zu nutzen, erfüllt werden.

4.3 Wiederverwendung von TCP-Verbindungen und Pipelining

Der Aufbau einer verschlüsselten TCP-Verbindung im mobilen Umfeld ist sehr zeitintensiv. Dies wurde in den vorangegangenen Unterkapiteln beschrieben. Es ist demnach sinnvoll, eine einmal etablierte verschlüsselte TCP-Verbindung für verschiedene Serveranfragen wiederzuverwenden. Diese Technik wird im Web-Umfeld bereits häufig eingesetzt und ist unter dem Namen *HTTP Pipelining* standardisiert [FGM+99]. Dieses Kapitel untersucht, wie die entfernte Ausführung von Programmcode für MCC-Anwendungen über eine einzige TCP-Verbindung organisiert werden kann und vergleicht verschiedene Implementierungen. Dass

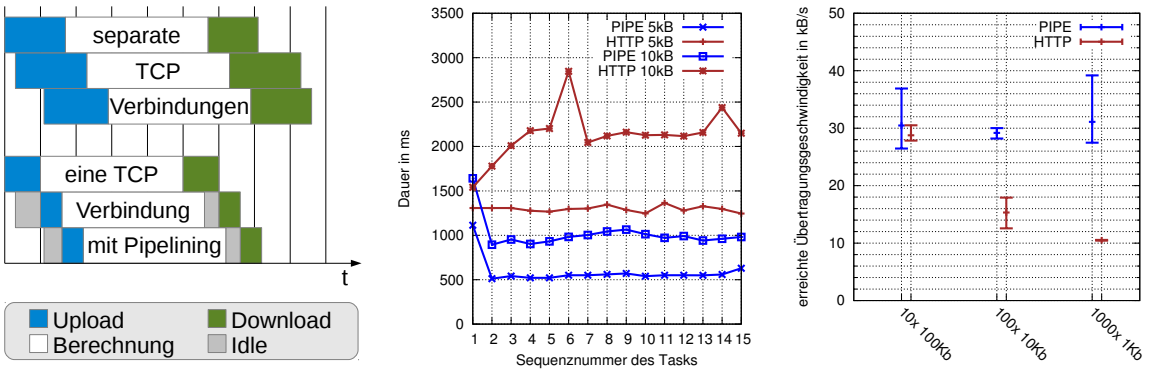


Abbildung 4.11: *Links:* Schematische Darstellung Pipelining. *Mitte:* Übertragungsdauer von Nachrichten (5 kB und 10 kB) in einer Sequenz mit und ohne Pipelining. *Rechts:* Bandbreitenausnutzung überlagerter Datentransfers (Min, Avg, Max).

Pipelining für MCC-Anwendungen Vorteile bringen kann, wurden in [FR12] bereits gezeigt und wird nachfolgend noch ausführlicher betrachtet.

4.3.1 Voraussetzungen und Modellierung von Pipelining

Beim Pipelining werden unabhängig zu verarbeitende Daten direkt nacheinander an den Server zur Verarbeitung gesendet, ohne jeweils die Bearbeitung der Zwischenergebnisse abzuwarten [KEEH10]. Abbildung 4.11 (links) zeigt eine schematische Darstellung des Pipelining-Konzepts im Vergleich zu jeweils einzelnen TCP-Verbindungen für jeden Aufruf. In der Darstellung wurde davon ausgegangen, dass sich aufgrund der geringeren Datenübertragungszeiten durch Pipelining die Gesamtverarbeitungsdauer für mehrere Anfragen verringert.

Beim Pipeling werden die Daten verschiedener Verarbeitungsschritte über nur eine TCP-Verbindung geleitet. Nachdem eine TCP-Verbindung etabliert wurde, werden die Eingabedaten sequentiell über den selben TCP-Kanal zum Server übertragen. Auf Serverseite werden die ankommenden Daten verarbeitet. Die Ergebnisse werden über den gleichen TCP-Kanal zurück zum Client übertragen. Besonders gut funktioniert dieses Verfahren, wenn die Verarbeitungsschritte unabhängig sind und dadurch keine Reihenfolge eingehalten werden muss. So kann ein fortlaufender Datenstrom entstehen, der die Übertragungsbandbreite möglichst gut ausnutzt. Ein Verarbeitungsschritt kann als Task aufgefasst werden. Ein Task besteht dabei aus den Eingabedaten, die zum Server übertragen werden, der Verarbeitungsfunktion und den resultierenden Ausgabedaten, die durch den Server generiert und zurück zum Client übertragen werden. Es ist demnach günstig, wenn sich ein Programm in einzelne unabhängige Tasks zerlegen lässt. Führt man nur einen Task aus, ist das Verhalten einem entfernten Prozeduraufruf ähnlich. Lässt sich ein Programm nicht in einzelne unabhängige Tasks zerlegen, so kann das Pipelining nicht effizient verwendet werden.

In der Mitte von Abbildung 4.11 sind die Ergebnisse synthetischer Benchmarks abgebildet. Diese zeigen die Ausführungszeit einer Sequenz von einzelnen Tasks (5 kB und 10 kB Übertragungsdaten) unter Verwendung einzelner HTTP-Verbindungen und unter Verwendung von Pipelining (PIPE) auf einer einzigen TCP-Verbindung. Es wurde ein Echo-Task verwendet, der keine Verarbeitung durchführt, sondern nur die Eingabedaten zurücksendet. Gemessen

wurde in der UMTS/HSPA-Konfiguration zu einem Server im *Amazon EC2* Rechenzentrum in Irland unter Verwendung des Dell Streak 7 Tablets. Wie erwartet braucht mit der Pipeling Technik nur die erste Anfrage etwas länger im Vergleich zu den nachfolgenden Anfragen, die über den gleichen TCP-Kanal gesendet werden. Dabei werden in diesem Beispiel RPC-Aufrufe sogar überlappt, um die Bandbreitenausnutzung zu verbessern. Da die überlappten Aufrufe aber nicht synchronisiert sind, können einzelne Aufrufe sogar länger dauern als nicht überlappte, da die verfügbare Netzwerkbandbreite eventuell von mehreren konkurrierenden Aufrufen geteilt werden muss. Für die RPC-Aufrufe bleibt die Ausführungszeit aufgrund konkurrierender Netzwerknutzung konstant oder steigt sogar an. Durch Pipelining kann ab dem zweiten Task bereits davon ausgegangen werden, dass die volle verfügbare Übertragungsbandbreite $Throughput_{max\,Duplex}$ genutzt werden kann. Dadurch vereinfacht sich die Berechnung von $\Delta t_{\text{transfer}}$ zu $\Delta t_{\text{transfer PIPE}}$ wie folgt:

$$\Delta t_{\text{transfer PIPE}} = \frac{msg\_size}{Throughput_{max\,Duplex}}. \tag{4.6}$$

In Abbildung 4.11 (rechts) ist der gemessene Durchsatz für insgesamt 1 MB Nutzdaten und verschiedene Anzahlen von Tasks zu sehen. Es wurden 10 Messungen für jedes Szenario vorgenommen. Es ist zu sehen, dass der Overhead für HTTP mit jeweils wenig Nutzdaten pro Aufruf höher ist. Dem gegenüber ist der Durchsatz bei allen drei Tests für das Pipelining Konzept nahezu gleich. Beim Pipelining ensteht durch die asynchrone Verarbeitung jedoch auch eine Wartezeit zwischen dem Zeitpunkt, zu dem ein Task versendet wird und dem Zeitpunkt, zu dem die verarbeiteten Daten wieder am Client vorliegen. Diese Zeit Δt_{delay} setzt sich zusammen aus der entfernten Verarbeitungszeit inklusive Datentransfer (siehe Gleichung (3.13)) und der eventuellen Wartezeit Δt_{queue}, die der Task aufgrund des Pipelinings in den Warteschlangen auf Client- und Serverseite verbringt:

$$\Delta t_{\text{delay}} = \Delta t_{\text{remote}} + \Delta t_{\text{queue}}. \tag{4.7}$$

Diese Wartezeit ist in Abbildung 4.11 (links) grau dargestellt. Es bedarf somit Warteschlangen auf Serverseite und auf Clientseite, um solche wartenden Tasks zwischenzuspeichern. Dies ist notwendig, um die Taskverarbeitung auf Serverseite nicht zu blockieren. Die Warteschlange für eingehende Tasks auf Clientseite muss stets groß genug sein, um alle eingehenden Tasks zu speichern. Wird nämlich auf das Eintreffen eines bestimmten eingehenden Tasks gewartet während die Warteschlange voll wird, kommt es zu einem Deadlock, da keine neuen Tasks mehr aufgenommen werden können, aber auch keine entnommen werden.

4.3.2 Pipelining-Implementierungen

Um das Konzept der Task-Verarbeitung über eine einzige TCP-Verbindung zu realisieren, wurde im Rahmen dieser Arbeit ein Protokoll entworfen, welches den Nachrichtenaustausch zwischen Client und Server spezifiziert. Das Protokoll beginnt nach dem Aufbau einer SSL-verschlüsslten TCP-Verbindung zunächst durch den Client mit dem Senden einer Präambel. Diese wird vom Server ausgewertet und entsprechend beantwortet. Sinn und Gestalt der Präambel werden später in Kapitel 7 genauer beleuchtet. Danach tauschen Client und Server solange Nachrichten aus, bis einer von beiden, üblicherweise der Client, die Verbindung

trennt. Es wird in *Client –> Server*-Nachrichten und *Server –> Client*-Nachrichten unterschieden. Das Taskaustauschprotkoll wird dabei auf ein kompatibles Protokoll abgebildet, welches Pipelining unterstützen muss. Nachfolgend werden mögliche Protokolle, die Abbildung des Task-Konzepts auf diese und deren Eignung für MCC genauer betrachtet.

HTTP Pipelining

HTTP Pipelining ist eine bereits 1999 standardisierte Erweiterung des HTTP-Protokolls [FGM+99], die neben dem Etablieren einer persistenten TCP-Verbindung für weitere Anfragen mittels *keep-alive* auch die Überlagerung von Anfragen und Antworten über die TCP-Verbindung unterstützt. So muss nicht erst die Antwort auf eine Anfrage abgewartet werden, bevor die nächste Anfrage abgesendet wird.

HTTP Pipelining hat jedoch einen Nachteil, der die Anwendung des Pipelinings de facto unmöglich macht. *HTTP Pipelining* unterstützt nur synchrones Pipelining. Das bedeutet, dass beim Client eingehende Antworten in der gleichen Reihenfolge ankommen müssen, wie die Anfragen gesendet wurden. Sendet ein Client nun eine sehr komplexe Anfrage zuerst und schiebt viele einfache Anfragen hinterher, muss der Server die Ergebnisse der einfachen Anfragen zwischenpuffern, da die Antwort der komplexen Anfrage zuerst gesendet werden muss. Da serverseitige Puffer die Serverperformance aufgrund des erhöhten Speicherbedarfs stark beeinträchtigen können, wird das *HTTP Pipelining* nur selten angewandt. Übrig bleibt allein die Wiederverwendung der TCP-Verbindung für sequentielle Anfragen, was die Performance von Folgenanfragen jedoch auch schon stark verbessert.

Somit kann *HTTP Pipelining* für MCC-Anwendungen nicht im Zusammenspiel mit asynchroner Taskverarbeitung angewendet werden.

SPDY

Eine Weiterentwicklung des *HTTP Pipelinings* ist in dem seit einigen Jahren immer beliebter werdenden SPDY-Protokoll zu sehen [TJA12, Mat13]. SPDY ist keine Abkürzung, sondern leitet sich vom englischen Wort *speedy* (deutsch *schnell*) ab. Dieses ist bereits in einer Vielzahl von Webbrowsern integriert und unterstützt neben den aus HTTP bekannten Eigenschaften auch asynchrones Pipelining. Zudem können die Header in SPDY komprimiert werden, was den Datenoverhead für kleine Nachrichten stark reduziert. Da die Reihenfolge der Anfragen und Antworten bei SPDY nicht erhalten werden braucht, kann ein konstanter Datenstrom zwischen dem Client und einem Server realisiert werden, wodurch die Duplexeigenschaften einer TCP-Verbindung bestmöglich ausgenutzt werden und sich die Bandbreitenausnutzung maximiert.

SPDY implementiert ein bidirektionales Datenaustauschprotokoll, bei dem verschiedene Datenströme (Streams) über eine einzige (verschlüsselte) TCP-Verbindung gemultiplext werden können. Dazu nutzt SPDY verschiedene Rahmen (Frames), die zur Übermittlung von Nutzdaten eines Streams (Data Frames) oder zur Steuerung der Streams selbst (Control Frame) verwendet werden (siehe Abbildung 4.12). Ein Frame umfasst stets 8 Byte Headerdaten. Nachdem bereits eine TCP-Verbindung aufgebaut wurde, beginnt das Erstellen eines neuen Streams mit dem Senden eines `SYN_STREAM` Frames. Darin sind ebenfalls die Headerinformationen wie beispielsweise die URL der angefragten Web-Ressource codiert. Der Server antwortet mit einem `SYN_REPLY` Frame. Darin ist die ausgehandelte Stream-ID enthalten.

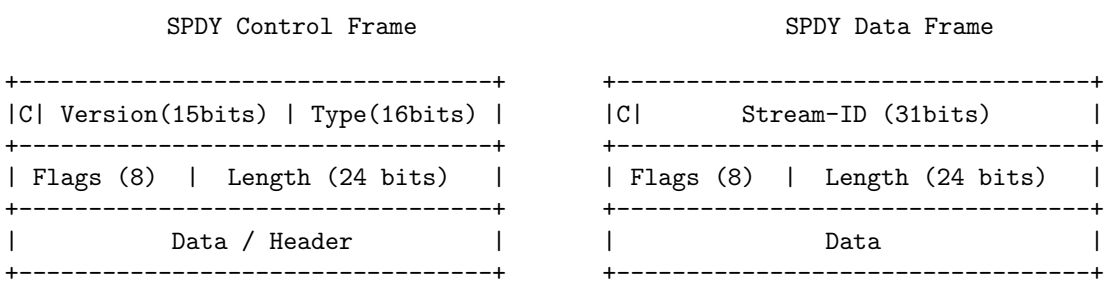


Abbildung 4.12: Schematische Darstellung der SPDY Protokoll Frames. (siehe Onlineressource `http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft1` (abgerufen am 16.2.2014))

Empfangene weitere Data Frames zu diesem Stream können nun als Antwort interpretiert werden. Außerdem können Data Frames auch zum Server übertragen werden. Zu beachten ist, dass eine große Nachricht in mehreren Data Frames übertragen werden kann. Das Schließen eines Streams wird durch das Senden des Control Frames `GOAWAY` realisiert. Eine einzelne Datenübertragung wird mittels SPDY üblicherweise in einem dedizierten Stream gekapselt.

SPDY befindet sich derzeit noch in der Standardisierungsphase[17]. Die Standardisierung von SPDY begann 2009 mit Draft 1 und ist 2013 bereits bei Draft 3 (SPDY/3) angelangt. Eine weitere Version, Draft 4, wird bereits erwartet. Für *Java* gibt es bereits einige SPDY-Implementierungen. In dieser Arbeit wird vor allem die Implementierung OkHttp[18] in ihrer aktuellen Version 1.1.1 für Android und Java-Applikationen betrachtet. Diese Implementierung unterstützt SPDY/3. Zudem wurde noch die Jetty-Implementierung des SPDY Protokolls[19] in der Version 7.6.2 betrachtet, welche jedoch nicht für *Android* verfügbar ist. Dies liegt daran, dass diese Implementierung *Java 7* benötigt, welches auf *Android*-Geräten derzeit noch nicht unterstützt wird. Die Jetty SPDY-Implementierung unterstützt SPDY/2 und verwendet *Java Socket Channels*, welche eine hohe messbare Übertragungsperformance versprechen. Die Jetty-Implementierung ist somit eher als eine Art Referenz zu betrachten und ermöglicht die bessere Einordnung der Performance anderer Implementierungen.

Pipelining

Zusätzlich wurde eine eigene Implementierung auf Basis von TCP-Sockets und SSL entwickelt. Die Implementierung ist für *Android*-Geräte und JRE-kompatible standardkonforme *Java*-Implementierungen wie dem *OpenJDK* oder *Oracle Java* verfügbar.

In Abbildung 4.13 sind die Frames der Übertragungsdaten beschrieben. In dieser Implementierung wird zwischen `Client -> Server` und `Server -> Client` Frames unterschieden. Die Eigenimplementierung ist bereits explizit an das Transferieren von Tasks angepasst. Die Zuordnung zwischen versendetem Task und eingehendem Ergebnis erfolgt auf Clientseite und wird durch eindeutige Task-IDs realisiert, die mit übertragen werden. Dies ist notwendig, da im Gegensatz zum *HTTP Pipelining* die Reihenfolge der eintreffenden Antworten verletzt werden darf.

[17] siehe Onlineressource `http://www.chromium.org/spdy/spdy-protocol` (abgerufen am 16.2.2014)
[18] Verfügbar unter Onlineressource `http://square.github.io/okhttp/` (abgerufen am 16.2.2014)
[19] Verfügbar unter Onlineressource `http://wiki.eclipse.org/Jetty/Feature/SPDY` (abgerufen am 16.2.2014)

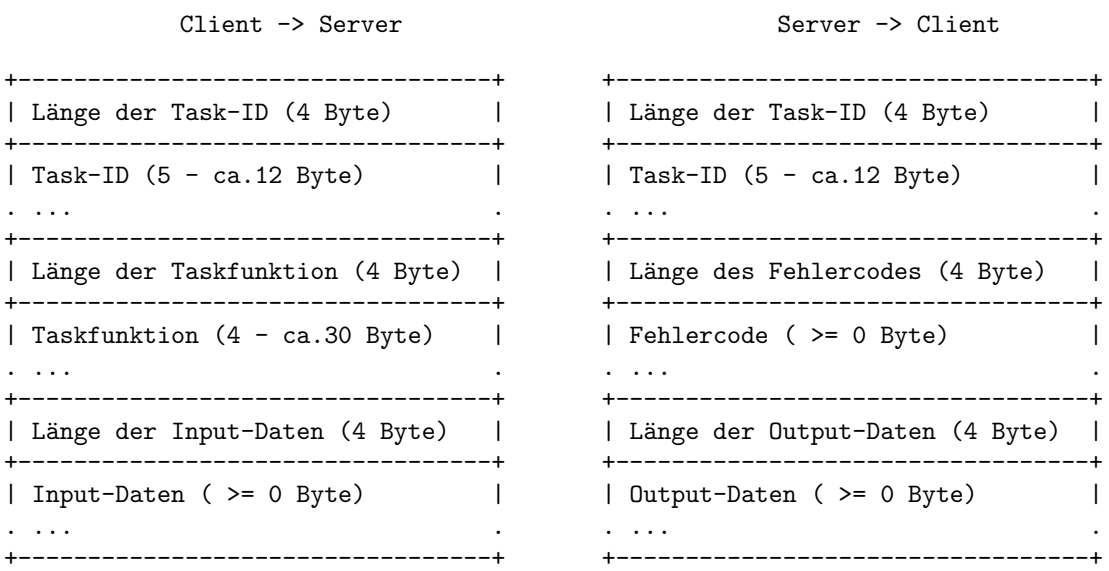


Abbildung 4.13: Schematische Darstellung der Pipelining Frames.

Ein Task wird bei dieser Implementierung unter Verwendung einer `Client->Server`-Nachricht vom Client zum Server übertragen. Eine Nachricht besteht aus der Task-ID, dem Namen der auszuführenden Task-Funktion und den eigentlichen Eingabedaten des Tasks. Somit ist mindestens mit einem Verwaltungsoverhead bezüglich der übertragenen Input-Daten von 21 Byte pro Task zu rechnen. Dieser Overhead kann jedoch aufgrund einer längeren Task-ID oder Task-Funktion auch leicht größer werden. Die Längen der Task-ID und Task-Funktion sind nicht begrenzt, sie können jedoch mit maximal 12 bzw. 30 Byte durch Erfahrungen aus der Praxis abgeschätzt werden.

In umgekehrter Richtung wird die nicht mehr benötigte Übertragung der Taskfunktion durch die optionale Übertragung einer Nachricht im Fehlerfalle ersetzt. Die Übertragung vom Server zum Client ist in einer `Sever->Client`-Nachricht gekapselt. In dieser Richtung ist demnach mit einem Overhead von mindestens 17 Byte pro Task zu rechnen.

4.3.3 Benchmarks

Die im vorangegangenen Abschnitt vorgestellten, für MCC-tauglichen Kommunikationstechnologien OkHttp, Jetty SPDY und das selbst implementierte Pipeliningprotokoll wurden bezüglich des erreichbaren Übertragungsdurchsatzes untersucht. Dazu wurde von dem Dell Latitude Laptop eine verschlüsselte Ende-zu-Ende Verbindung zu einer virtuellen Maschine im Rechenzentrum von *Amazon EC2* in Irland aufgebaut. Es wurden jeweils 1 MB Nutzdaten hin und zurück übertragen. Dazu wurden die Stückelungen 10x100 kB, 100x10 kB und 1000x1 kB verwendet. Es wurden 30 Messungen für alle drei Konfigurationen durchgeführt. Abbildung 4.14 visualisiert die Ergebnisse. Dargestellt sind Minimum, Maximum und der Mittelwert der jeweiligen Messreihe.

Zunächst ist wieder deutlich zu erkennen, dass die Messwerte stark schwanken können. Um die Übersichtlichkeit zu wahren, wurde die Standardabweichung der Messreihen hier nicht angegeben. Die Standardabweichung war jedoch bei allen Verfahren etwa ähnlich. Außerdem sind alle drei Technologien in der Lage, die Pipeliningfähigkeit auch umzusetzen,

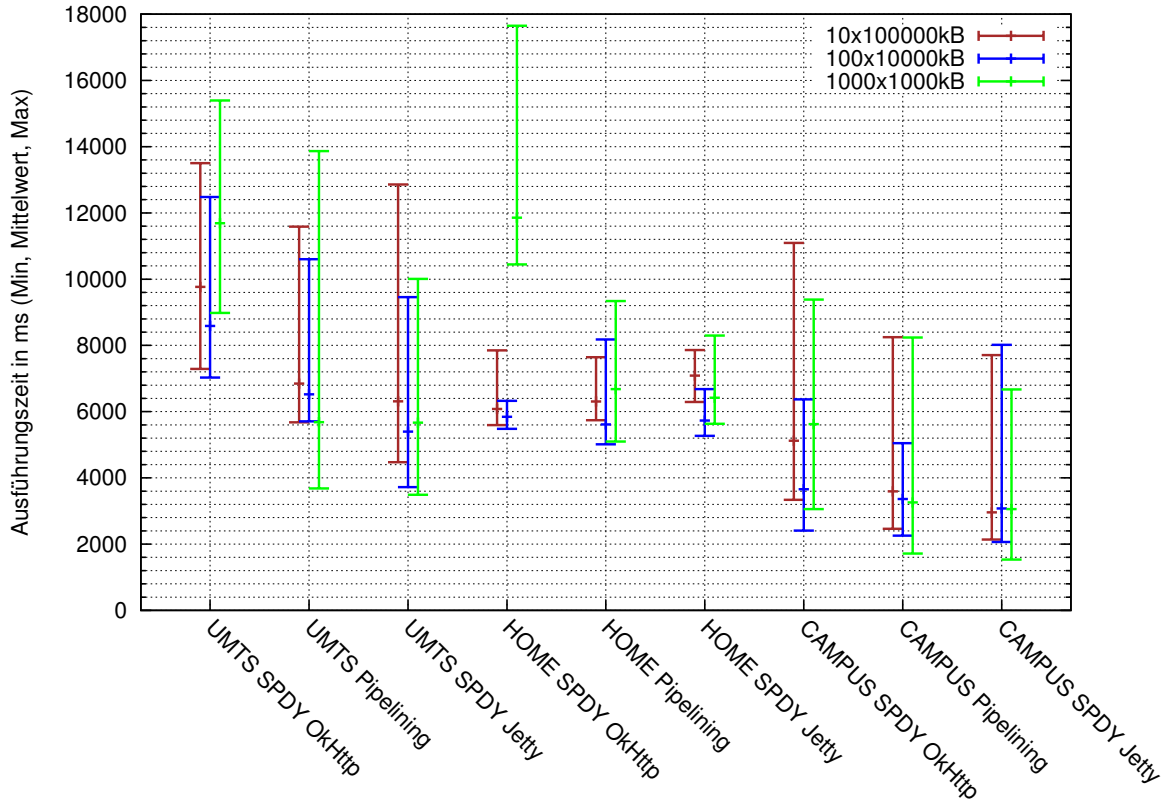


Abbildung 4.14: Performancevergleich verschiedener Pipelining-fähiger Kommunikationsprotokolle unter Verwendung verschiedener Konfigurationen.

was an den etwa gleichbleibenden Ausführungszeiten für verschiedene Stückelungen zu erkennen ist. Weiterhin kann an dem Diagramm abgelesen werden, dass in der CAMPUS-Konfiguration im Durchschnitt die höchste Datenrate zu erwarten ist. Lässt man die OkHttp-Implementierungen aus den Betrachtungen heraus, ist die Performance in der HOME-Konfiguration und in der UMTS-Konfiguration ähnlich hoch. Für sehr kleine Nachrichten ist die Performance von OkHttp oft schlecht und die höchste Übertragungsgeschwindigkeit erreicht OkHttp für die 100x10 kB Stückelung. Die Jetty SPDY und die eigene Pipelining Implementierung sind von der Performance ähnlich schnell.

Obwohl mit der Jetty SPDY-Implementierung eine schnelle und ausgereifte echte Pipelining-Implementierung bereits existiert, so ist diese leider nicht für *Android* nutzbar. Die OkHttp-Implementierung kann bezüglich der Performance besonders für kleine Nachrichten nicht überzeugen. Abschließend wird im Weiteren die eigene Pipelining-Implementierung genutzt, da diese eine gute Leistung erzielt und für alle betrachteten Geräte/Plattformen verfügbar ist. Die konkrete *Java*-API dieser Implementierung wird in Kapitel 7 vorgestellt.

4.4 Performancevorhersage in Alltagssituationen

Die Güte der Vorhersage in Alltagssituationen wurde durch mehrere Messungen unter Verwendung der eigenen Pipeliningimplementierung in den in Tabelle 4.1 beschriebenen Konfigu-

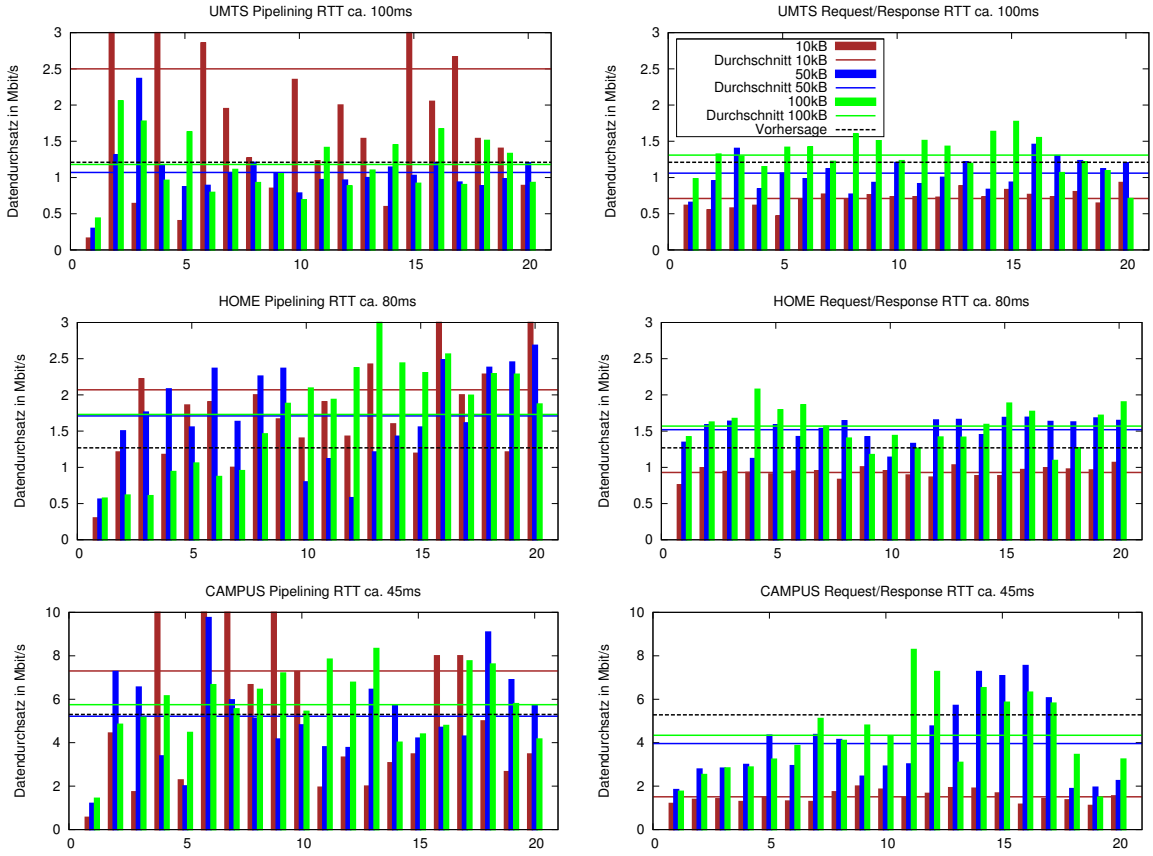


Abbildung 4.15: Auswertung der vorhergesagten Übertragungsgeschwindigkeiten für die Übertragung von Tasks im Pipeliningmodus und im Request/Response-Modus für verschiedene Konfigurationen.

rationen durchgeführt. Dazu wurde serverseitig eine virtuelle Maschine das Anbieters *Amazon EC2* EU Irland benutzt.

Zunächst wurde die verfügbare Netzwerkperformance in drei aufeinander folgenden Messungen bestimmt. Dazu wurde das frei verfügbare Werkzeug *SPEEDTEST.NET MINI*[20] benutzt, welches ebenfalls auf dem Testserver installiert wurde. *SPEEDTEST.NET MINI* ist eine Web-Applikation, die in einem Webbrowser ausgeführt wird. Sie überträgt Dateien zum Server und zurück und misst dabei die maximale Übertragungsbandbreite. Detailliertere Informationen zu *SPEEDTEST.NET MINI* sind in Kapitel 7.2.1 über Aufbau und Implementierung der entwickelten Middleware zu finden. Zusätzlich kann die RTT auf *Android*-Geräten beispielsweise über die kostenlose Applikation *Android Terminal Emulator*[21], welche eine Kommandozeile bereitstellt, und das Kommandozeilenprogramm *ping* bestimmt werden. Außer diesen Informationen und der verwendeten Kommunikationstechnologie hat der Anwender zur Laufzeit keine weiteren Informationen verfügbar.

Nach Bestimmung der durchschnittlichen Kommunikationsperformance wurde unter Be-

[20] Verfügbar unter Onlineressource `http://www.speedtest.net/mini.php` (abgerufen am 3.2.2014)

[21] Verfügbar unter Onlineressource `https://play.google.com/store/apps/details?id=jackpal.androidterm` (abgerufen am 3.2.2014)

achtung der Ergebnisse aus Abschnitt 4.2 und Gleichung (4.5) die realistische verfügbare Bandbreite bestimmt. Aufgrund des gemessenen Variationskoeffizienten von ca. 0.3 wurde die gemessene Geschwindigkeit pessimistisch eingeschätzt und um 30% verringert. Die nachfolgende Gleichung fasst die Bildung von $Throughput_{max}$ für die durchgeführten Messungen zusammen:

$$Throughput_{max} = MIN(\varnothing Throughput_{upload}, \varnothing Throughput_{download}) \cdot 0.7. \quad (4.8)$$

Anschließend wurde eine Pipeliningverbindung aufgebaut und 20 Echo-Tasks wurden gestartet. Ein Echo-Task $Task_i$ besteht aus einer Bytefolge an Eingabedaten $msg\_size_{Task_i}$, die zum Pipeliningserver übertragen werden. Der Server schickt die Daten als Ergebnis der Taskausführung jedoch sofort unangetastet zurück. Aus dem Abstand des Eintreffens der Ergebnisse $t_{Fin\ Task_i}$ und $t_{Fin\ Task_{i-1}}$ lässt sich dann bei Kenntnis der Länge der Bytefolge die ausgeschöpfte Übertragungsbandbreite $Throughput_{Task_i}$ auf der Anwendungsschicht messen. Sie wird wie folgt bestimmt:

$$Throughput_{Task_i} = \frac{msg\_size_{Task_i}}{t_{Fin\ Task_i} - t_{Fin\ Task_{i-1}}}. \quad (4.9)$$

Abbildung 4.15 (links) zeigt den gemessenen Durchsatz für alle 20 gestarteten Tasks in den drei verschiedenen Konfigurationen unter Verwendung des Dell Streak 7 Tablets. Dargestellt ist zudem auch der Mittelwert für die drei jeweils gemessenen Eingabegrößen 10 kB, 50 kB und 100 kB. Erkennbar ist zunächst, dass die erste Messung für $Task_0$ unbrauchbar ist, da vorher kein Zeitstempel zum Vergleich existiert. Hier wurde der Zeitstempel des Startens des ersten Tasks verwendet. Es ist weiterhin zu erkennen, dass die gemessenen Werte stark schwanken, insbesondere für die kleinste Testgröße 10 kB. Dies ist der Tatsache geschuldet, dass hier die Messzeiten sehr klein waren (<50ms), sodass sich kleine Abweichungen aufgrund von Störungen stark auf die Bandbreitenmessung auswirken. Für die größeren Eingabedaten schwanken die Werte weniger stark. Am geringsten fällt die Schwankung für UMTS aus, was bereits erwartet wurde (siehe Abbildung 4.15 (links)). Außerdem liegt der Mittelwert der gemessenen Bandbreite für alle Tasks in etwa im Bereich der Vorhersage. Dies trifft insbesondere für UMTS und die CAMPUS-Konfiguration zu. Für die HOME-Konfiguration lag der vorhergesagte Wert etwas unterhalb des Mittelwerts. Somit kann prinzipiell davon ausgegangen werden, dass die vorhergesagten Übertragungsbandbreiten im Durchschnitt auch real zu erreichen sind. Hierbei ist jedoch gerade für kleine Eingabegrößen mit großen Schwankungen zu rechnen.

In einem weiteren Test wurde ermittelt, welche Leistung bei der Abbildung von RPC-Aufrufen auf das Pipeliningverfahren zu erwarten ist. Dazu wurden Tasks synchron in Analogie zu RPC-Aufrufen ausgeführt. Es wurden erneut 20 Echo-Tasks unter den gleichen Bedingungen wie oben beschrieben gestartet. Doch bevor der jeweils nachfolgende Task gestartet wurde, wurde auf das Eintreffen des Ergebnisses des vorhergehenden Tasks gewartet. Eine Messung umfasst also den Zeitraum vom Absenden des Tasks bis zum Eintreffen des Ergebnisses auf der Anwendungsschicht. Da serverseitig keine Verarbeitung stattfindet und keine Marshalling benötigt wird, wird die messbare Verarbeitungszeit $\Delta t_{remote\ Task_i}$ eines Task $Task_i$ unter Beachtung von Gleichung (4.2) und Gleichung (4.6) wie folgt angegeben:

$$\Delta t_{remote\ Task_i} = \frac{msg\_size_{Task_i} \cdot 2}{Throughput_{Task_i}} + RTT. \tag{4.10}$$

Die $msg\_size_{Task_i}$ ist doppelt berücksichtigt, da die Daten jeweils hin und zurück übertragen werden. Die RTT wird in die Gleichung aufgenommen, damit die Gleichung auch Gültigkeit besitzt, wenn keine Nutzdaten übertragen werden. In diesem Fall würden nur wenige Byte zum Server und zurück übertragen werden, was in etwa der RTT entsprechen sollte. Nach Umstellung kann man die erreichte Übertragungsgeschwindigkeit für einen einzelnen Task $Task_i$ wie folgt ermitteln:

$$Throughput_{Task_i} = \frac{\Delta t_{remote\ Task_i} - RTT}{msg\_size_{Task_i} \cdot 2}. \tag{4.11}$$

Abbildung 4.15 (rechts) visualisiert die gemessene Datenübertragungsgeschwindigkeit für 20 aufeinander folgende sequentielle Task-Ausführungen. Die hier gemessenen Werte schwanken weniger stark als die der Pipelining-Messung. Dies liegt darin begründet, dass bei den RPC-Aufrufen kein gleichzeitiger Upload und Download stattfindet und somit keine zwei Datenströme abwechselnd übertragen werden müssen, was bei WLAN aufgrund der fehlenden Voll-Duplex-Fähigkeit zu starken Schwankungen führen kann. Jedoch sind auch vor allem bei der CAMPUS-Konfiguration Schwankungen über die Dauer der 20 Messungen zu erkennen. Auch hier ist der ausgeschöpfte Datendruchsatz bei kleinen Übertragungsdaten geringer. Hier spielt die RTT offenbar noch eine Rolle. Bei größeren Daten relativiert sich deren Einfluss. Auch hier ist erkennbar, dass für die größeren Daten die durchschnittliche Übertragungszeit in etwa dem vorhergesagten Wert entspricht.

Insgesamt zeigen die Messungen auch, dass bei Einsatz des Pipelinings sehr ähnliche Übertragungscharakteristiken für die einzelnen Technologien messbar sind. Zudem ist die Schwankungsbreite des messbaren Übertragungsdurchsatzes unter Verwendung von Pipelining immer noch sehr hoch. Somit sind die prognostizierten Übertragungswerte eher als Erwartungswert zu verstehen. Dies reicht aus, um bei einer Vielzahl von Tasks eine Taskverarbeitungsrate anzugeben. Es ist jedoch nicht möglich, einen definierten konstanten Abstand des Eintreffens der Ergebnisse der Taskverarbeitung einzuhalten. Für einzelne RPC-Aufrufe kann im Allgemeinen von weniger schwankenden Werten ausgegangen werden, sodass ab ca. 50 kB eine relativ genaue Vorhersage der Verarbeitungsdauer gemacht werden kann.

4.5 Fazit

Mobile Endgeräte und kabellose Netzwerkverbindungen haben besondere Charakteristiken und stellen besondere Anforderungen an Kommunikationstechnologien. Diese Charakteristiken wurden unter Verwendung verschiedener Konfigurationen und Geräte analysiert. Dabei wurden ausgehend von der Grundperformance die Duplexfähigkeit, der Einfluss von Verschlüsselung und der Datencodierung untersucht. Die Analyse ergab für die getesteten WLAN und 3G Mobilfunkverbindungen eine maximale Datenübertragungsrate im einstelligen Mbit-Bereich, oft hohe RTT-Werte bis in den hohen zweistelligen Millisekundenbereich und von Fall zu Fall unterschiedliche Duplexeigenschaften, die von der Beschaffenheit der tatsächlichen Ende-zu-Ende Verbindung abhängen. Im Ergebnis ist es deshalb empfehlenswert, eine

bereits etablierte TCP-Verbindung für mehrere Anfragen wiederzuverwenden, um auch für kleine Nachrichtengrößen eine gute Übertragungsperformance zu erreichen und die teils lange Dauer für den Aufbau einer verschlüsselten TCP-Verbindung zu relativeren. Außerdem sollten Upload und Download auch bei kabelloser Kommunikation überlagert werden, um die Gesamtübertragungszeit zu reduzieren. Aufgrund des geringen Overheads ist zudem eine binäre Datencodierung zu bevorzugen.

Um diese Ergebnisse zur effizienten Kommunikation im mobilen Umfeld umzusetzen, wurden verschiedene Pipelining-fähige Protokolle auf ihre Tauglichkeit untersucht und verschiedene Implementierungen miteinander verglichen. Dabei stellte sich heraus, dass nur die eigene Pipeliningimplementierung den Anforderungen nach stabilen Ergebnissen und der Einsatzfähigkeit für *Android* entsprach.

Abschließend wurde unter Benutzung der Pipeliningimplementierung überprüft, wie gut die anhand gemessener Parameter vohergesagten Übertragungsraten für einzelne Übertragungen tatsächlich erreicht werden. Dabei wurden ausschließlich die maximale Übertragungsrate für Upload und Download und die RTT zur Laufzeit gemessen. Diese realistische Überprüfung ergab, dass die vorhergesagten Werte für Eingabedaten ab 50 kB im Durchschnitt gut stimmen. Für kleinere Daten sind die Ergebnisse eher pessimistischer zu bewerten. Zusätzlich wurde auch erneut festgestellt, dass die real gemessenen Einzelwerte stark schwanken können. Dennoch kann unter Verwendung des Pipelinings die mobile Kommunikationsperformance anhand weniger ad hoc messbarer Werte belastbar abgeschätzt werden. Dieses Verfahren wird in den nachfolgenden Kapiteln vorausgesetzt, um die Gesamtverarbeitungsdauer entfernt ausgeführten Programmcodes vorherzusagen.

Um eine Vorhersage für weitere mobile Kommunikationstechnologien zu erstellen, müssen deren Eigenschaften bezüglich Duplexfähigkeit und Schwankungsbreite bestimmt werden, um zu bestimmen, ob Pipelining sinnvoll angewendet werden kann. Ist dies der Fall, so kann auch für diese Technologien die erwartete Übertragungsbandbreite anhand der durchschnittlichen Upload und Downloadbandbreite und der RTT einfach bestimmt werden.

5 Server-unterstützte mobile Anwendungen auf Cloud-Technologien

Mobile Cloud-unterstützte Anwendungen haben zeitlich begrenzte Anforderungen an entfernt verfügbare Ressourcen, die je nach Anwendung und Nutzerverhalten zwischen wenigen Sekunden bis Minuten betragen können. Üblicherweise werden Cloud-Ressourcen eines IaaS-Anbieters jedoch auf Stundenbasis abgerechnet, was zu einem erheblichen Kostenoverhead bei der Nutzung führen kann. In diesem Kapitel wird deshalb analysiert, wie die Granularität bezüglich der Ausleihdauer von Cloud-Ressourcen (VMs) für *Mobile Cloud-unterstützte Anwendungen* erhöht werden kann. Dieses Kapitel beschreibt mögliche Modellierungsvarianten, deren Vor- und Nachteile, sowie deren Abbildung auf öffentlich verfügbare Cloud-Infrastruktur-Anbieter. Da der Einfluss verschiedener Lastbalancierungsstrategien auf reale Cloud-Infrastruktur im großen Maßstab nur mit erheblichem finanziellen Aufwand bestimmbar wäre, wird stattdessen eine Simulation durchgeführt. Diese wird unter Verwendung möglichst realistischer Eingaben und Simulationsparameter durchgeführt, die aus der Literatur und durch eigene Beispielapplikationen unter Verwendung des IaaS-Anbieters *Amazon EC2* gewonnen wurden.

Compute-Ressourcen (VMs) aus dem Portfolio eines IaaS-Anbieters werden nach dem Selbstbedienungsprinzip genutzt, können jederzeit gestartet werden und werden üblicherweise auf Stundenbasis gemietet. So lässt sich deren Anzahl kurzfristig gut an die momentane Anfragemenge anpassen. In Kapitel 3.2 wurde vorgeschlagen, dass jede Ressourcenanfrage auf eine dedizierte VM zugewiesen werden sollte. Dadurch nutzt jeder Client die Ressource exklusiv und es kann sichergestellt werden, dass die Verarbeitungsgeschwindigkeit einer immer gleichen Anfrage auf der VM kaum schwankt. Zudem steht jedem Client durch die exklusive Nutzung auch die volle Netzwerkbandbreite zur Ressource zur Verfügung, was auch diesbezüglich eine stabile Übertragungsrate liefert. Die Gleichungen (3.15) und (3.16) bilden die Stabilität der Kommunikationsperformance und der Verarbeitungsgeschwindigkeit ab. Die Stabilität ist wichtig, um belastbare Aussagen bezüglich der Performance von *Mobilen Cloud-unterstützten Anwendungen* machen zu können. Denn sollte eine entfernte Ausführung in Betracht gezogen werden, so muss aufgrund des enstehenden finanziellen Aufwandes auch eine schnellere Ausführung garantiert sein und die Kostenbeschränkung muss eingehalten werden (siehe Bedingungen (3.12) und (3.18)).

Die Art der Modellierung der exklusiven Ressourcenallokation und die dafür notwendigen Lastbalancierungsstrategien sind jedoch in der Literatur noch nicht beschrieben worden. Würde man für jede Anfrage eine neue VM starten, so würde dies nicht nur lange dauern, sondern es würde aufgrund der Abrechnungsperiode im Stundentakt eventuell auch ein hoher Kostenoverhead entstehen, da die Nutzungsdauer eher im Minutenbereich liegt.

Da die Angebote der IaaS-Anbieter nicht verändert werden können, müssen VM-Instanzen für verschiedene Anfragen wiederverwendet werden. Es sind im Rahmen dieser Arbeit Allokationsstrategien und Lastbalancierungsstrategien zu entwerfen, die für Clients gewisse Schranken bezüglich der VM-Allokations- und Verarbeitungszeit garantieren, auf deren Basis dann eine realistische Vorhersage der Verarbeitungsdauer der *Mobile Cloud-unterstützte Anwen-*

dungen gemacht werden kann. Im Detail werden also in diesem Kapitel der Einfluss der Größen

- Dauer für Authentifizierung Δt_{auth},
- Dauer für Ressourcenallokation Δt_{alloc},
- Verarbeitungsdauer Δt_{work}
- und die daraus resultierenden Kosten *cost* in Abhängigkeit der gewählten Ressource R

auf die Gesamtverarbeitungszeit Δt_{remote} (siehe Gleichung (3.17)) unter Verwendung verschiedener Allokations- und Lastbalancierungsstrategien genauer untersucht.

5.1 Erhöhung der Granularität

Es wird davon ausgegangen, dass die zu entwerfende Middleware für *Mobile Cloud-unterstützte Anwendungen* für eine große Menge an Clients bereitgestellt wird. Somit können verschiedene Anfragen auf die verfügbaren VMs zugewiesen werden. Unabhängig von der gewählten Lastbalancierungsstrategie kann unter Verwendung eines IaaS-Anbieters die Anzahl verfügbarer Ressourcen dem tatsächlichen Bedarf flexibel angepasst werden. Somit kann zunächst jedem Client eine Ressource zur Verarbeitung zugewiesen werden.

Verschiedene *Mobile Cloud-unterstützte Anwendungen* benötigen jedoch auch verschiedenen serverseitigen Programmcode, was im Normalfall dazu führt, dass für jede Anwendung eine eigene Ausführungsumgebung in Form eines eigenen VM-Images bereitgestellt werden muss. Eine VM führt dann jeweils ein solches Image aus und somit nur eine einzige Anwendung. Jedes VM-Image besteht aus einer kompletten Betriebssysteminstallation nebst installiertem Softwarestack und der eigentlichen Anwendung.

Der zeitliche Zugriff auf drei verschiedene *Mobile Cloud-unterstützte Anwendungen*, die auf verschiedenen VMs ausgeführt werden, ist in Abbildung 5.1 (oben) dargestellt. Jede der drei Applikationen (App1, App2 und App3) wird auf einer zu diesem Anwendungstyp passenden VM ausgeführt. VM-Instanzen eines Applikationstyps können nicht für *Mobile Cloud-unterstützte Anwendungen* eines anderen Typs wiederverwendet werden. Da sich die zwei Nutzungszeiten der beiden Mobilgeräte für App1 und App3 für die dargestellte Abfolge jeweils überschneiden, müssen aufgrund der exklusiven Ressourcenallokationen jeweils zwei VMs dieses Typs gestartet werden. Obwohl VMs für einzelne Nutzungen auch wiederverwendet werden können (beispielsweise App2), werden für die Abfolge der dargestellten exklusiven Nutzungsperioden der drei Applikationen insgesamt fünf VMs benötigt. Diese sind rechts oben dargestellt. Als Resultat müssten in diesem Fall mindestens fünf volle Abrechnungsperioden, also 5 h Nutzungszeit, bezahlt werden.

Eine Erhöhung der Granularität der Nutzungsperiode könnte dieses Problem etwas reduzieren. Würde man die Nutzungsdauer auf Minutenbasis abrechnen, so würde der Kostenoverhead erheblich sinken. Eine Möglichkeit, um die Erhöhung der Granularität auf Basis der bestehenden IaaS-Angebote zu realisieren, ist die Wiederverwendung der VMs für verschiedene Anwendungen. Somit kann die Auslastung der VM pro Abrechnungsperiode erhöht werden und der Kostenoverhead sinkt. Jedoch müsste man bei Wiederverwendung möglicherweise sehr viele Applikationen gleichzeitig auf einer VM installieren. Dies führt unter Umständen zu Performance- und Managementproblemen. Um dieses Problem zu umgehen, wird

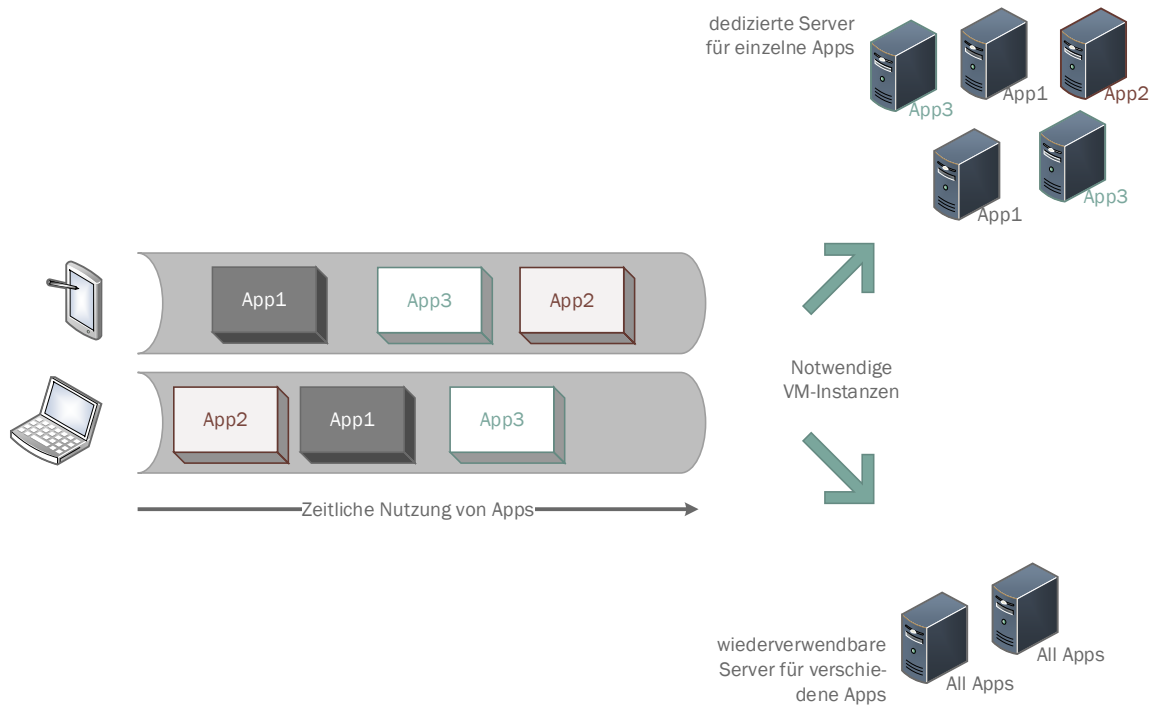


Abbildung 5.1: Visualisierung der zeitlichen Nutzung verschiedener *Mobiler Cloud-unterstützter Anwendungen*, hier *Apps* genannt, durch zwei Mobilgeräte unter Nutzung dedizierter und wiederverwendbarer VM-Instanzen.

in der hier vorgestellten Middleware serverseitig kein Programmcode vorinstalliert. Alle VM-Instanzen sind identisch und ein Client installiert den Programmcode für die benutzte *Mobile Cloud-unterstützte Anwendung* selbst. Diese Modellierung erhöht zwar den Datentransfer zur VM aufgrund der zusätzlichen Übertragung des Programmcodes, jedoch steigt durch die universellen VM-Instanzen die Wiederverwendungsrate stark an und somit sinkt der Kostenoverhead für die ungenutzten VM-Laufzeiten. Abbildung 5.1 (unten) visualisiert die bereits beschriebene Abfolge der Nutzungsperioden für die drei Applikationen unter Benutzung der universell wiederverwendbaren VMs. Dabei werden nur zwei VM-Instanzen benötigt, da sich stets nur maximal zwei Nutzungsperioden zeitlich überschneiden, was die Kosten für die dargestellte Abfolge erheblich reduziert.

Um eine Kostenreduktion zu erreichen, kann beim gleichzeitigen Starten einer gewissen Menge von VMs zusätzlich ein Rabatt gewährt werden. Dazu muss die Anzahl benötigter VMs im Voraus bekannt sein. Eine Broker-Architektur, die diesen Fakt berücksichtigt, ist beispielsweise in [WNLL12] beschrieben. Kostenreduktion kann auch durch die Verwendung von sogenannten *Spot*-Instanzen erreicht werden, die durch eine Auktion allokiert werden. Diese sind zwar im Schnitt billiger als *On-Demand*-Instanzen, sie sind jedoch nicht immer verfügbar und können unvermittelt beendet werden. Unter Verwendung dieser *Spot*-Instanzen können jedoch bei Kenntnis der Verarbeitungsdauer auch Performancegarantien gegeben werden. In [MD11] ist eine solche Strategie beschrieben. Da die Verarbeitungsdauer und die Menge benötigter VMs jedoch im Fall von *Mobilen Cloud-unterstützten Anwendungen* nicht immer bekannt ist, können diese Verfahren nicht verwendet werden.

Durch die Wiederverwendungstechnik wird das Installieren des Programmcodes zur Laufzeit notwendig. Diese Technik wird häufig als Code-Offloading bezeichnet und wurde in Kapitel 2.2 bereits vorgestellt. Die Codeinstallation zur Laufzeit kostet jedoch nicht nur Zeit $\Delta t_{install}$, es wird auch eine zusätzliche Übertragung von Daten notwendig ($msg\_size_{install}$), welche sich nach Gleichung (4.1) und (4.2) in der Kommunikationszeit Δt_{comm} niederschlägt. Die erhöhte Kommunikationszeit wird im folgenden deshalb durch Δt^*_{comm} angegeben. Das Code-Offloading beeinflusst die entfernte Ausführungszeit Δt_{remote} wie folgt:

$$\Delta t_{remote} = \Delta t_{auth} + \Delta t_{alloc} + \Delta t_{install} + \Delta t^*_{comm} + \Delta t_{work}. \tag{5.1}$$

Durch die Wiederverwendungsstrategie dürfen jedoch die Bedingungen (3.12) und (3.18) bezüglich der schnelleren Ausführung auf Serverseite und der Begrenzung der Kosten nicht verletzt werden. Dadurch ergeben sich bei Verwendung der Wiederverwendungstechnik folgende zusätzliche Einschränkungen:

$$\Delta t_{install} \ll \Delta t_{work}, \tag{5.2}$$
$$msg\_size_{install} \ll msg\_size_{data}. \tag{5.3}$$

Diese Einschränkungen sind jedoch nicht immer zwingend einzuhalten, da es trotz deren Verletzung noch möglich sein kann, dass die Bedingungen (3.12) und (3.18) nicht verletzt werden. Sie sind eher als Richtwerte zu verstehen. Da die Codeinstallation zur Laufzeit von der konkreten Anwendung abhängt, wird deren Einfluss nicht hier, sondern in Kapitel 6 untersucht.

Zusätzlich verringert die erhöhte Wiederverwendungswahrscheinlichkeit von VMs für einzelne Anfragen eventuell auch die Allokationszeit Δt_{alloc}. Da eine wiederverwendete VM bereits gestartet ist, fällt die Allokationszeit wesentlich geringer aus. Die Zeit zum Starten einer neuen VM-Instanz wird mit $\Delta t_{startup}$ beschrieben. Es gilt:

$$\Delta t_{startup} \geq \Delta t_{alloc}. \tag{5.4}$$

Im Beispiel in Abbildung 5.1 (oben) müssen insgesamt fünf VMs gestartet werden, was in fünf von sechs Fällen die entfernte Ausführungszeit verlängert. Im unteren Beispiel müssen nur in zwei von sechs Fällen neue VMs gestartet werden, was sich positiv auf die Verarbeitungszeit der vier anderen Anfragen auswirkt. Die Verringerung der durchschnittlichen Allokationszeit Δt_{alloc} ist bei Verwendung der Wiederverwendungstechnik nicht garantiert, die Wahrscheinlichkeit nimmt jedoch mit der Anzahl der Anfragen in einem festen Zeitraum im Allgemeinen zu. Dieser Sachverhalt wird nachfolgend durch eine Simulation noch genauer untersucht.

5.2 Authentifizierung, Abrechnung und Lebenszyklus

Bezug nehmend auf Gleichung (5.1) wird klar, dass Authentifizierung und Allokation der benötigten Ressource eine gewisse Zeit benötigen können. Nachfolgend werden verschiedene Modellierungsvarianten einer Middleware für *Mobile Cloud-unterstützte Anwendungen* vorgestellt. Es wird dabei davon ausgegangen, dass eine zertifikatsbasierte verschlüsselte

TCP-Verbindung verwendet wird. In Kapitel 4 wurde erarbeitet, dass eine persistente TCP-Verbindung und eine Pipeline-artige Anordnung der Datentransfers notwendig sind, um die verfügbare mobile Kommunikationsbandbreite gut auszunutzen und somit auch deren Performance einschätzen zu können. Demnach muss ein Client jeweils nur beim Verbindungsaufbau einmalig authentifiziert werden. Die vorgestellten Architekturen sind dabei auf VMs eines IaaS-Anbieters abgebildet.

Aufgrund der verwendeten Verbindung mit TCP-Pipelining kann zwischen der Zeit für die Bearbeitung einer einzelnen/ersten Anfrage und der Zeit für die Bearbeitung von Folgeanfragen unterschieden werden, bei denen dann $\Delta t_{auth} = 0$ gilt. Nachfolgend werden verschiedene Architekturen einer Middleware für *Mobile Cloud-unterstützte Anwendungen* vorgestellt und bezüglich Skalierungsfähigkeit und Lastbalancierung untersucht. Es wird dabei davon ausgegangen, dass mehrere Anfragen/Tasks durch den Client an die VM gesendet werden. Die Middlewarearchitektur beeinflusst die berechnungsintensiven Verarbeitungsschritte $\Delta t_{install}$ und Δt_{work} in der gleichen Weise. Somit wird für nachfolgende Betrachtungen folgende Vereinfachung angenommen:

$$\Delta t^*_{work} \quad = \quad \Delta t_{work} + \Delta t_{install}. \tag{5.5}$$

Untersucht werden die Auswirkung auf die Eigenschaften

- der Summanden in Gleichung (5.1) bzw. (5.5) ($\Delta t_{auth}, \Delta t_{alloc}, \Delta t^*_{comm}$ und Δt^*_{work}) und
- auf das Zertifikat-Management bzw. die Sicherheitsarchitektur bezüglich der Verschlüsselung der Kommunikation.

Um die Modellierung besser nachvollziehen zu können, wird mit einer Architektur bestehend aus einem einzelnen Front-Server begonnen.

5.2.1 Einzelner Front-Server

Steht nur ein einziger Server für Authentifizierung, Abrechnung und Ausführung der eigentlichen Anfragen zur Verfügung, so werden alle eingehenden Anfragen auf diesen Server zugewiesen. Eine einzelne Anfrage würde von einem einzelnen Front-Server dann in folgender Reihenfolge bearbeitet:

1. Authentifizierung des Anfragenden,
2. Bearbeiten der Anfrage(n).

Dabei würde der einzelne Front-Server alle benötigte Funktionalität übernehmen. Diese Variante bedarf nur eines Server-Zertifikats und ist in Abbildung 5.2 visualisiert. Bei mehreren gleichzeitigen Anfragen sinkt jedoch die Antwortgeschwindigkeit pro Client und es kann keine Performancegarantie gegeben werden. Außerdem müssen sich alle Clients eventuell eine einzige Netzwerkverbindung zum Server teilen. Somit sinkt auch hier die verfügbare Bandbreite pro Client bei mehreren gleichzeitigen Anfragen. Zusätzlich wird durch eine Überlastung eventuell auch die Dauer für die Authentifizierung verzögert, weshalb diese auch von der Anzahl der gleichzeitigen Anfragen abhängt. Zusammenfassend sind folgende Auswirkungen auf die Summanden in Gleichung (5.1) bzw. (5.5) zu erwarten:

Erste Anfrage	Folgende Anfragen
$\Delta t_{auth} \approx \#\,\text{Anfragen}$	$\Delta t_{auth} = 0$
$\Delta t_{alloc} = 0$	$\Delta t_{alloc} = 0$
$\Delta t^*_{comm} \approx \#\,\text{Anfragen}$	$\Delta t^*_{comm} \approx \#\,\text{Anfragen}$
$\Delta t^*_{work} \approx \#\,\text{Anfragen}$	$\Delta t^*_{work} \approx \#\,\text{Anfragen}$

Anfragen bezeichnet eine Anzahl gleichzeitiger Client-Anfragen, die auch von verschiedenen Clients abgesetzt werden können. Das Zeichen für Proportionalität ($\approx$) zeigt hier die Abhängigkeit der Größen an. Steigt beispielsweise die Anzahl gleichzeitiger Anfragen, so steigt auch jeweils die Verarbeitungsdauer pro Anfrage. Da nur ein Server zur Verfügung steht, ist die Performance bei steigender Anzahl von Anfragen sehr schlecht. Dieses Verhalten ist für *Mobile Cloud-unterstützte Anwendungen* nicht geeignet.

5.2.2 Mehrere Front-Server

Da die Anzahl von VMs bei Verwendung eines IaaS-Anbieters leicht skaliert werden kann, kann die Menge benötigter VMs an die Anzahl von Anfragen angepasst werden. Zudem ist es sinnvoll, die Benutzerverwaltung und Abrechnung auf eine dedizierte Maschine auszulagern, um die konsistente Datenhaltung zu vereinfachen. In diesem Fall kann, wie gefordert, jedem Client eine dedizierte Ressource für Berechnungen bereitgestellt werden. Die Architektur mit mehreren Front-Servern ist in Abbildung 5.3 visualisiert. Ein Client muss jedoch in diesem Fall eine freie Ressource zunächst durch Verbindungsanfragen selbst suchen (Strichlinien). Da VMs bei jedem Start üblicherweise neue IP-Adressen zugewiesen bekommen, ist dies eventuell schwierig. Selbst wenn der Client die Gesamtmenge der Ressourcen kennt, müsste er im schlechtesten Fall alle anfragen, bis er eine freie Ressource findet. Sind alle Ressourcen belegt, kann die Allokation eventuell sogar fehlschlagen. Eine Anfrage würde folgendermaßen durchgeführt:

1. Finden einer freien Ressource,
2. Authentifizierung des Anfragenden,
3. Bearbeiten der Anfrage.

Hierbei wird für jede Ressource ein eigenes Server-Zertifikat benötigt. Durch den Einsatz einer Cloud-Infrastruktur kann hier bei geeigneter Skalierung der Front-Server bereits eine Performancegarantie bezüglich der Verarbeitungsgeschwindigkeit Δt_{work} gegeben werden. Da hier keine zentrale Management-Komponente existiert, ist die Skalierung der benötigten VMs jedoch eventuell schwierig und müsste dezentral erfolgen.

Durch die ausgelagerte Abrechnung und Authentifizierung in eine eigene VM kann die Dauer Δt_{auth} stark verbessert werden. Durch die exklusive Nutzung der Ressourcen der VM können pro Zeiteinheit mehr Authentifizierungsvorgänge erfolgen, ohne dass die Dauer Δt_{auth} stark ansteigt. Dies ist damit zu begründen, dass es eher unwahrscheinlich ist, dass es sehr viele tatsächlich gleichzeitige Authentifizierungsvorgänge gibt. Selbst bei 100000 Anfragen in 12 h (beispielsweise zwischen 8–20 Uhr) müssten pro Sekunde im Durchschnitt nur ca. 2,3 Anfragen bearbeitet werden. In [WSKV08] wird beispielsweise beschrieben, dass ein LDAP-basiertes System bis zu einer Rate von ca. 50 Anfragen pro Sekunde eine konstante Authentifizierungsperformance von unter 100 ms erreicht. Das darin getestete LDAP-Verzeichnis hatte 10000 Einträge (Benutzer) mit je 488 Byte. Bei Datensätzen von 100–1000 Byte Größe pro

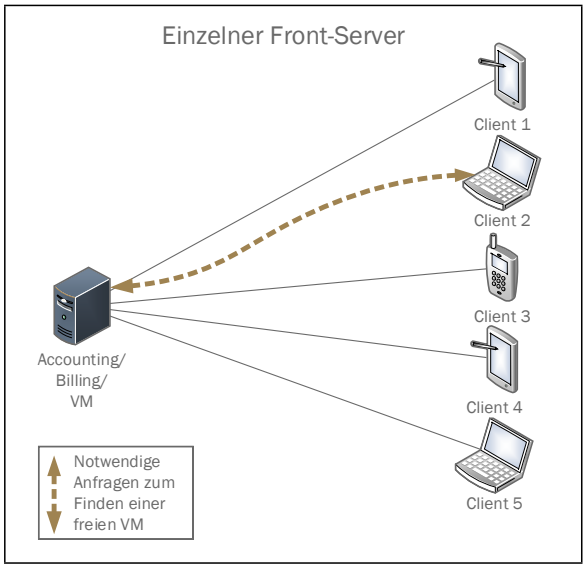


Abbildung 5.2: Allokationsvorgang für eine VM auf einer Architektur mit einem einzelnen Front-Server.

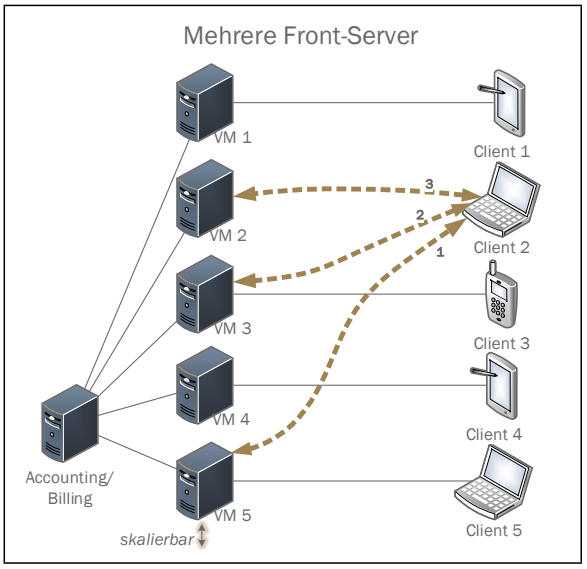


Abbildung 5.3: Allokationsvorgang für eine VM auf einer Architektur mit mehreren Front-Servern und ausgelagerter Benutzerverwaltung und Abrechnung.

Benutzer betrüge die Datenbankgröße ca. 0,1–1 GB für 1 Mio Benutzer. Die Dauer von nur ca. 100 ms für Δt_{auth} ist zudem sehr gering und kann fast immer vernachlässigt werden. Im Folgenden wird also davon ausgegangen, dass nahezu alle Authentifizierungsvorgänge in einer sehr kurzen Dauer Δt_ϵ erfolgen können. Zusammenfassend sind folgende Auswirkungen auf die Summanden in Gleichung (5.1) bzw. (5.5) zu erwarten:

Erste Anfrage	Folgende Anfragen
$\Delta t_{auth} \leq \Delta t_\epsilon$	$\Delta t_{auth} = 0$
$\Delta t_{alloc} \approx \#\,\text{Anfragen}$	$\Delta t_{alloc} = 0$
$\Delta t^*_{comm} = \text{konstant}$	$\Delta t^*_{comm} = \text{konstant}$
$\Delta t^*_{work} = \text{konstant}$	$\Delta t^*_{work} = \text{konstant}$

Auch hier kann bezüglich der entfernten Ausführungszeit keine obere Schranke angegeben werden, da das Finden einer freien Ressource noch von der Anzahl gleichzeitiger Anfragen abhängt. Diese Suche ist besonders im mobilen Umfeld langwierig, da hier der verschlüsselte Verbindungsaufbau aufgrund hoher Latenzen sehr lange dauert.

5.2.3 Broker-Server und Compute-Serverfarm

Im Vergleich zur Variante mit mehreren Front-Servern, wird in dieser Variante ein Broker-Server hinzugefügt, der nur die Authentifizierung übernimmt und dem anfragenden Client eine VM für Berechnungen zuweist (siehe Abbildung 5.4). Dieser Broker-Server hat eine feste IP-Adresse und ist somit immer erreichbar, was die Anzahl von Verbindungen im Vergleich zur Variante ohne Broker-Server stark reduzieren kann. Das eventuell langwierige Suchen einer freien VM entfällt. Auch bei dieser Variante ist die Verarbeitungsgeschwindigkeit bei

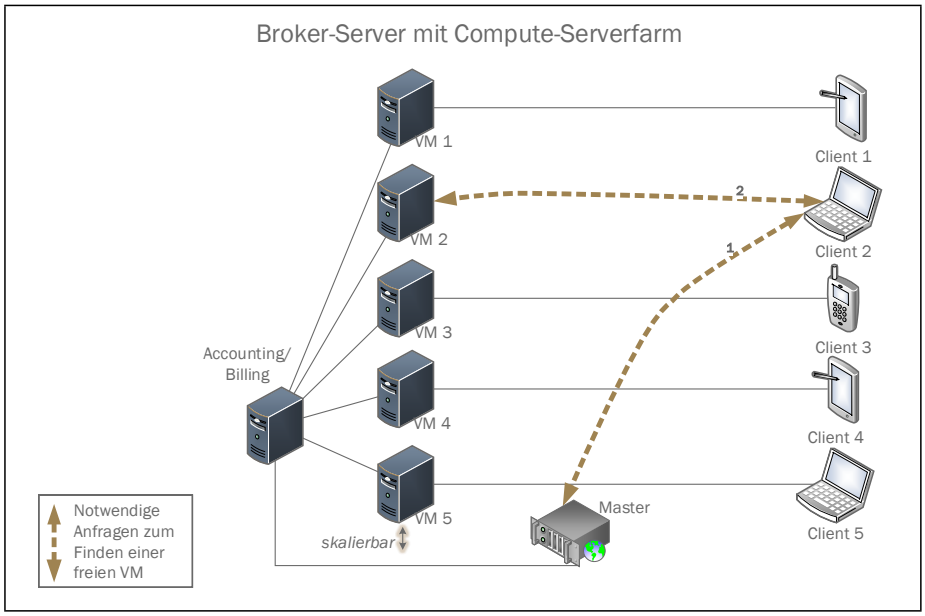


Abbildung 5.4: Allokationsvorgang für eine VM auf einer Architektur mit Broker-Server.

entsprechender Skalierung der Ressourcen konstant. Da die VM jedoch nicht mehr authentifiziert, muss ein Session-Management implementiert werden, damit sich der Client auch mit der zugewiesenen VM verbinden kann. Eine Anfrage wird dann folgendermaßen beantwortet:

1. Authentifizierung des Anfragenden durch den Broker,
2. Allokieren einer VM für die Anfrage,
3. Rücksenden der Serveradresse und der etablierten Session-ID an den Client,
4. Senden der Anfrage zur VM,
5. Beantworten der Anfrage,
6. Explizites Freigeben der VM durch den Client.

Durch den Broker-Server, der alle VM-Instanzen kennt, kann Δt_{alloc} insofern reduziert werden, als dass dafür ein Wert $< \Delta t_{startup}$ angenommen werden kann. Δt_{alloc} ist somit nicht mehr von der Anzahl der Anfragen abhängig und es kann in jedem Fall eine Ressource zugewiesen werden. Diese Architekturvariante wird aufgrund ihrere positiven Eigenschaften für *Mobile Cloud-unterstützte Anwendungen* im Folgenden weiter untersucht. Dazu wird im folgenden Kapitel Δt_{alloc} durch Simulation genauer bestimmt. Der Broker kann hier kaum zum Flaschenhals werden, da er nur für die Authentifizierung zuständig ist, alle weiteren berechnungsintensiven Anfragen werden direkt an die allokierte VM gesendet. Somit steht auch für jeden Client die volle Netzwerkbandbreite zur Verfügung. Zusammenfassend sind folgende Auswirkungen auf die Summanden in Gleichung (5.1) bzw. (5.5) zu erwarten:

Erste Anfrage	Folgende Anfragen
$\Delta t_{auth} \leq \Delta t_{\epsilon}$	$\Delta t_{auth} = 0$
$\Delta t_{alloc} \leq \Delta t_{startup}$	$\Delta t_{alloc} = 0$
Δt^*_{comm} = konstant	Δt^*_{comm} = konstant
Δt^*_{work} = konstant	Δt^*_{work} = konstant

Ein Nachteil dieser Variante ist, wie auch bei der Variante mit mehreren Front-Servern, dass für jede VM ein eigenes Serverzertifikat benötigt wird. Hier muss man also bestenfalls Wildcards in Zertifikaten verwenden oder mit einer eigenen *Trusted Authority* arbeiten, die die Serverzertifikate ausstellt und der der Client vertraut (siehe Kapitel 4.2.2).

5.2.4 Lebenszyklus einer Client-Anfrage

Durch die favorisierte Modellierungsvariante mit dem Broker-Server wird die Implementierung eines Lebenszyklus für eine VM-Nutzung obligatorisch. Dies ist notwendig, um durch die Vermeidung mehrfacher Authentifizierung Zeit einzusparen. Zudem ist der Lebenszyklus auch für die korrekte Abrechnung der Nutzungsdauer essentiell. Als Repräsentation des Lebenszyklus wird nach der Authentifizierung des Client durch den Broker-Server ein *Sessionkey* erstellt. Die Abrechnung endet erst mit Auflösung der Session. Der Nutzer zahlt also für die Vorhaltung der für ihn bereitgestellten VM, unabhängig davon, ob er sie tatsächlich nutzt. Im Gegenzug wird nach der Initiierung einer Session aber auch eine geringere Latenz und eine gleichbleibende Performance der VM sichergestellt.

In Abbildung 5.5 ist der Lebenszyklus einer Client-Anfrage anhand eines Beispiels dargestellt. Ein Client setzt eine Anfrage zur Nutzung einer VM einer bestimmten Qualität via `getInstanceForType()` zum Broker ab. Der Broker prüft zunächst beim Abrechnungsdienst, ob der Nutzer, der sich mit Name und Passwort angemeldet hat, zur Nutzung des gewünschten Dienstes zugelassen ist (`authenticate()`). Falls ja, allokiert der Broker nun die gewünschte Ressource für den Client (`getCloudResourceForType()`). Da der Broker alle laufenden Instanzen aller VMs und deren Qualität und Auslastung kennt, kann er eine laufende Instanz wiederverwenden oder eine neue Instanz starten (`startVM()`). Dies hängt von der verwendeten Allokationsstrategie ab. Anschließend generiert der Broker einen *Sessionkey*, der fortan die Verbindung zwischen dem anfragenden Client und der Cloud-Ressource repräsentiert (`generateKey()`). Diesen *Sessionkey* übermittelt der Broker zur Cloud-Ressource, damit diese den Client akzeptieren kann. Außerdem wird der *Sessionkey* auch zum Abrechnungssystem übermittelt, damit dieses den Vorgang zur späteren Abrechnung speichern kann (`startbilling()`). Abschließend wird dem Client die IP-Adresse und der zu verwendende *Sessionkey* zum Zugriff auf den gewünschten Dienst übermittelt. Der Client kommuniziert danach nur noch direkt mit der zugewiesenen Cloud-Ressource. Er verbindet sich direkt mit der VM unter Angabe des *Sessionkey* (`connect()`). Nun können solange Berechnungen durchgeführt werden, bis der Client die Verbindung trennt (`disconnect()`). Danach meldet die VM selbstständig das Ende der Abrechnung (`reportUsage()`) und meldet ihren aktuellen Status an den Broker (`returnKey()`).

Bei der Implementierung der Session gibt es verschiedene Möglichkeiten der Terminierung. Der übliche Weg sieht vor, dass der Client sich ordnungsgemäß mit der VM verbindet und nach Trennung der Verbindung die Session beendet wird. Aus verschiedenen Gründen könnte es jedoch auch vorkommen, dass ein Client zwar eine VM anfragt und zurückgeliefert bekommt, sich jedoch nie verbindet. Aus diesem Grund wird ein *Sessionkey* mit einer Ver-

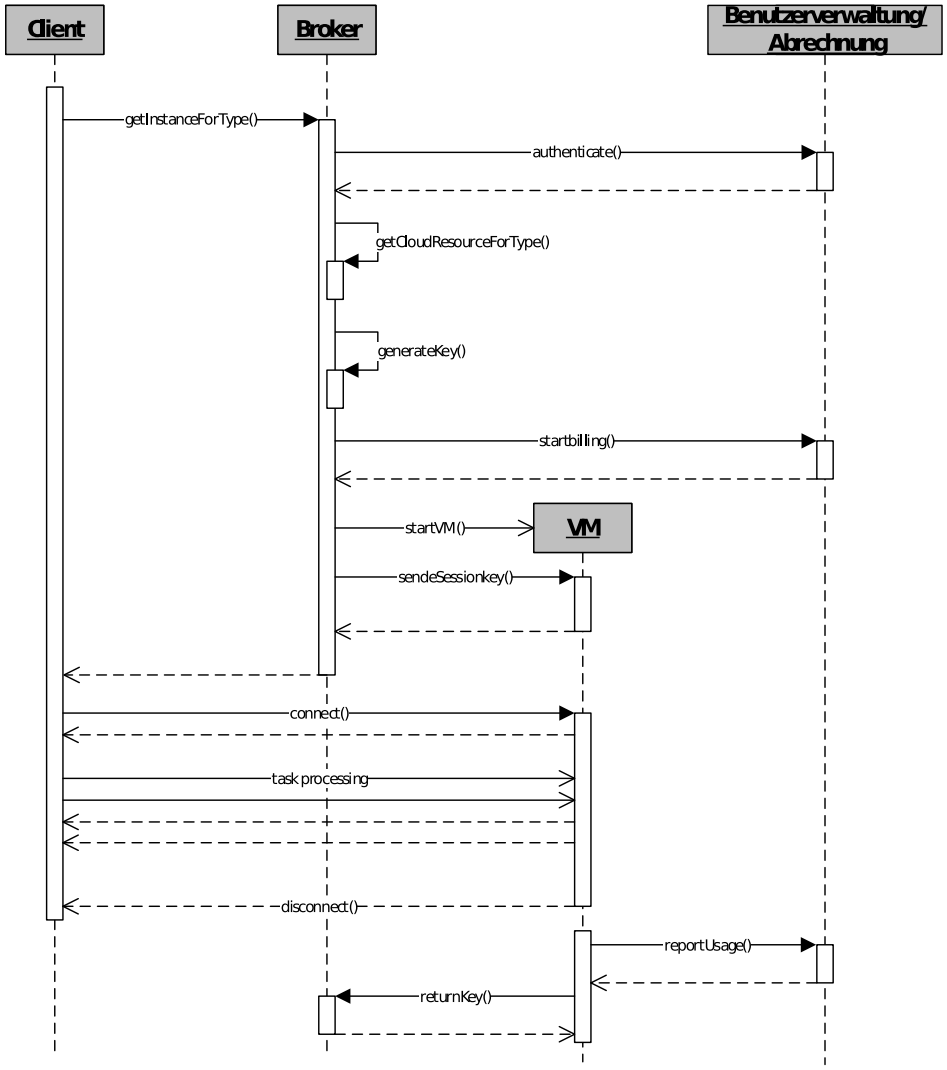


Abbildung 5.5: Illustration des Lebenszyklus einer Client-Anfrage (Sequenzdiagramm) anhand eines Beispiels.

fallsdauer (englisch *Lease*) versehen, die dafür sorgt, dass die Session beim Überschreiten dieser Dauer durch die VM automatisch beendet wird. Danach würde ein Verbindungsversuch durch den Client fehlschlagen und der Client müsste erneut eine VM beim Broker-Server anfragen.

Es sei nochmals darauf hingewiesen, dass die Kommunikation zwischen allen genannten Komponenten (Broker, VM, Benutzerverwaltung/Abrechnung und Client) verschlüsselt werden muss, um die Integrität und Geheimhaltung des *Sessionkey* zu gewährleisten, ohne den die Implementierung des beschriebenen Lebenszyklus nicht möglich wäre.

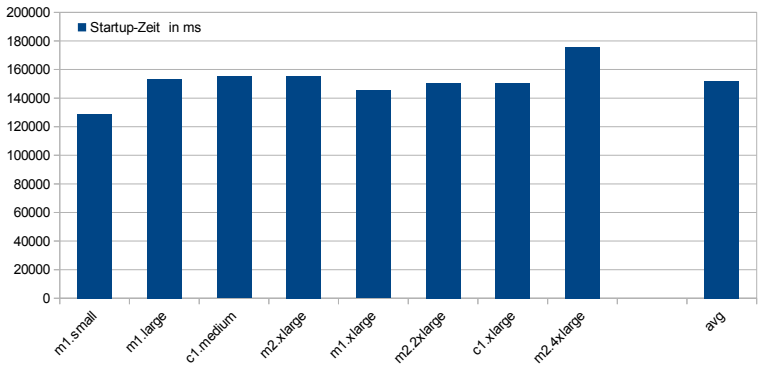


Abbildung 5.6: Durchschnittliche gemessene Startup-Zeiten von EC2 VM-Ressourcen im Amazon Rechenzentrum in Irland.

5.3 Cloud-Ressourcen Startup/Shutdown-Zeiten

Für *Mobile Cloud-unterstützte Anwendungen* wird die Anzahl von Cloud-Ressourcen kontinuierlich dem notwendigen Bedarf angepasst. Dazu muss die Dauer für das Starten einer VM eines IaaS-Anbieters bekannt sein. Um aussagekräftige Simulationen der Allokationszeit für Cloud-Ressourcen durchführen zu können, muss zunächst die tatsächlich zu erwartende Dauer $\Delta t_{startup}$ gemessen werden. Wie schon in Kapitel 4 wurde als Zielumgebung die *EC2* Infrastruktur des Anbieters Amazon im Rechenzentrum in Irland benutzt. Zunächst wurden verfügbare Arbeiten bezügliche Daten zur Startup- und Shutdown-Zeit von *Amazon EC2*-Instanzen untersucht. Die Angaben der Startup-Zeit variieren dabei in der Literatur stark.

In [IOY+11] ist die Startup-Zeit und Release-Zeit für verschiedene VM-Typen von *Amazon EC2* analysiert worden. Als Release-Zeit wird die Zeit betrachtet, die vom Ausschalten der VM bis zur tatsächlichen Freigabe der Ressource durch den IaaS-Anbieter vergeht. Als $\Delta t_{startup}$ wurde hier die Dauer von der Installationsanfrage bis zum abgeschlossenen Bootvorgang betrachtet. Es wurden je 20 aufeinander folgende Messungen für die VM-Typen `m1.small, m1.large, m1.xlarge, c1.medium und c1.xlarge` durchgeführt. Der Mittelwert für $\Delta t_{startup}$ lag dabei bei ca. 73 s. Die durchschnittliche Release-Zeit lag bei ca. 19 s. In welchem Rechenzentrum die Messungen durchgeführt wurden, ist nicht bekannt. In [IYE11] wird die Startup-Zeit des `m1.small`-Instanztyps von *Amazon EC2* über ein Jahr beobachtet und mit durchschnittlich ca. 77 s angegeben. Auch hier wurde die Zeit von der Installationsanfrage bis zum Abschluss des Bootvorgangs gemessen. In welchem Rechenzentrum die Messungen durchgeführt wurden, ist ebenfalls nicht bekannt. In einer weiteren Arbeit [PSS+12] wurde die $\Delta t_{startup}$-Zeit für die VM-Typen `m1.small, m1.large und m1.xlarge` und 10 einzelne Messungen mit durchschnittlich ca. 36 s angegeben. Dabei wurde als Startup-Zeit jedoch nur die Zeit von der Installationsanfrage bis zur Bereitstellung der VM gemessen. Die Zeit für den Bootvorgang ist nicht eingeschlossen. Die Messungen wurden im Rechenzentrum von *Amazon EC2* in Virginia, USA (US-EAST) durchgeführt.

Aufgrund der wenigen und stark divergierenden Messungen, die in der Literatur zu finden waren, wurden eigene Messungen bezüglich $\Delta t_{startup}$ im Amazon Rechenzentrum in Irland (EU-WEST) durchgeführt. Die durchschnittlichen Startup-Zeiten sind in Abbildung 5.6 dar-

gestellt. Es wurden jeweils 10 aufeinander folgende Messungen durchgeführt. Als Startup-Zeit ist hier die Zeit relevant, die von der Installationsanfrage bis zum ersten erfolgreichen Login-Versuch via SSH vergangen ist. Im Mittel wurde hier eine Dauer von 150 s gemessen.

Die stark variierenden Startup-Zeiten sind einerseits durch verschiedene Messmodi (mit/ohne Dauer des Bootvorgangs) zu erklären. Andererseits sind die Startup-Zeiten wahrscheinlich auch in unterschiedlichen Rechenzentren verschieden. Deshalb wird im Weiteren von einer Dauer $\Delta t_{startup}$ von ca. 150 s ausgegangen, wie sie selbst gemessen wurde. Die Release-Zeit ist für die Simulation nicht relevant, da sie sich auf keinen Summanden der Gleichung (5.1) auswirkt. Eine ungenutzte VM sollte stets 5 min vor Beginn der nächsten Abrechnungsperiode heruntergefahren werden. Dadurch sollte genügend Zeit für die Freigabe der VM zur Verfügung stehen. Das Herunterfahren kann automatisiert durch den Broker erfolgen.

5.4 Scheduling und Lastbalancierung seitens des Cloud-Anbieters

Um eine schnelle Bereitstellungszeit und einen geringen Kostenoverhead für *Mobile Cloud-unterstützte Anwendungen* zu erreichen, ist es, wie bereits beschrieben, notwendig, Cloud-Ressourcen wiederzuverwenden. Dazu sind Scheduling- und Lastbalancierungsstrategien zu entwerfen, die die Bereitstellungszeit Δt_{alloc} und die anfallenden Kosten auf einer Menge von in der Anzahl skalierbaren VM-Instanzen eines IaaS-Anbieters minimieren. Um eine gewisse Performance für Client-Anfragen zu garantieren, soll jede Anfrage möglichst exklusiv auf eine dedizierte Serverinstanz (VM) zugewiesen werden. Verschiedene Zuweisungsstrategien, man spricht auch von Ressourcenallokation, haben unterschiedlichen Einfluss auf die Kosten bzw. den Kostenoverhead und die Dienstqualität (Performance). Um diesen Einfluss zu analysieren, wurden zwei verschiedene Allokationsstrategien simuliert. Zudem ist eine Simulation auch sinnvoll, um die Performance, die ein IaaS-Anbieter liefert, die daraus entstehenden Kosten und schließlich dessen Eignung für *Mobile Cloud-unterstützte Anwendungen* vor der Installation der Middleware bereits zu überprüfen. Die nachfolgenden Ausführungen dieses Unterkapitels sind in gekürzter Form in [FRTH12] zu finden.

Simuliert wird die zeitliche Nutzung einer Menge von VM-Instanzen für eine Menge von Client-Anfragen, die über einen Tag verteilt auftreten. Das Modell der Simulation ist in Tabelle 5.1 zusammengefasst und wird nachfolgend beschrieben. Bei der Simulation handelt es sich um eine diskrete Ereignissimulation[1]. Jeder Simulationslauf durchläuft genau einen Tag zwischen 0 und 24 Uhr. Da der Simulation kein stetiger Prozess, sondern einzelne Ereignisse zu Grunde liegen, ist die diskrete Ereignissimulation eine passende Methode zur Durchführung der Simulation. Es werden mehrere Qualitätsattribute betrachtet, die zugleich die Simulationsgrößen darstellen:

1. die Allokationszeit Δt_{alloc},
2. der Overhead der Verarbeitungszeit Δt_{shared} und
3. die Gesamtkosten der VM-Nutzungsdauer.

[1] Eine diskrete Ereignissimulation ist eine zeitdiskrete Simulation, bei der auftretende Ereignisse nach deren Reihenfolge abgearbeitet werden. Das Auftreten eines Ereignisses ändert dabei den Simulationszustand. Bei dieser Art Simulation wird die Tatsache ausgenutzt, dass sich der Simulationszustand zwischen Ereignissen nicht ändert und folglich nicht betrachtet werden muss. Grundlage der hier durchgeführten Simulation ist eine einfache Ereignisverarbeitung (englisch *Event Scheduling*). (siehe [Tys99, S. 17 ff.])

Tabelle 5.1: Beschreibung des Modells zur Simulation der Allokationsstrategien für VMs

Definition	Beschreibung
$Sim(VMs, REQs, S_{init}, S_{final}, F_{alloc})$	Beschreibung einer Simulation
S_{init}, S_{final}	anfänglicher/finaler Simulationszustand
t_{sim}	ein Zeitpunkt in der Simulation
$num_{standby} \geq 1$	$\lvert VM_{free}\rvert$ sollte stets $num_{standby}$ sein
$vm(t_{create}, \Delta t_{startup}, t_{kill}, num_{req})$	Beschreibung einer Cloud-Ressource
$VMs = \{vm_0, \ldots, vm_x \mid x \in \mathbb{N}\}$	Eine Menge von Cloud-Ressourcen
$load(vm)$	gibt num_{req} zum Zeitpunkt t_{sim}zurück
$free(vm) = true$	falls $num_{req} == 0$
$ready(vm) = true$	falls $(t_{create} + \Delta t_{startup}) \leq t_{sim}$
$VM_{free} = \{vm \mid vm \in VMs \wedge free(vm) \wedge ready(vm)\}$	Menge ungenutzter Cloud-Ressourcen
$req(t_{request}, t_{start}, \Delta t_{work}, t_{finish})$	Beschreibung einer Anfrage
$REQs = \{req_0, \ldots, req_x \mid x \in \mathbb{N}\}$	Eine Menge von Anfragen
Δt_{alloc}	$t_{start} - t_{request}$
Δt_{shared}	$t_{finish} - t_{start} - \Delta t_{work}$
$\Delta t_{overhead}$	$\Delta t_{alloc} + \Delta t_{shared}$
$F_{alloc}(req)$	allokiert eine vm zu einer Anfrage req
$F_{assign}(vm, req)$	weist einer Anfrage req eine vm zu
$F_{adjust}(REQs, VMs)$	passt $num_{standby}$ vonVMs an

Δt_{shared} repräsentiert einen Overhead bezüglich Δt_{work}, der durch simultane, also nicht exklusive, Verarbeitung mehrerer Anfragen auf einer VM entsteht, und sich in Gleichung (5.1) durch einen zusätzlichen Term Δt_{shared} widerspiegelt:

$$\Delta t_{remote} \quad = \quad \Delta t_{auth} + \Delta t_{alloc} + \Delta t_{install} + \Delta t^{*}_{comm} + \Delta t_{work} + \Delta t_{shared}. \tag{5.6}$$

Der Zeitaufschlag aus Δt_{shared} und Δt_{alloc} wird zur späteren besseren Vergleichbarkeit der Allokationsstrategien in $\Delta t_{overhead}$ zusammengefasst. Ziel der Simulation ist es, für die vorgestellte Variante der exklusiven VM-Allokation eine Einordnung gegenüber einem bekannten System bezüglich Kosten- und Performanceoverhead zu bekommen.

Es wurde eine exklusive Allokationsstrategie implementiert, die jede Anfrage auf eine dedizierte Ressource zuweist. Außerdem wurde zum Vergleich eine traditionelle Strategie implementiert, die sehr schnell eine Ressource aus dem bestehenden Ressourcenpool aussucht und zuweist, auch wenn diese dann nicht exklusiv genutzt werden kann. Durch Multiplexing kann dann keine Performancegarantie gegeben werden. Die Simulation offenbart, wie häufig Multiplexing dann für verschiedene Anfragen durchgeführt werden musste und wie sich das auf die Simulationsgrößen auswirkt.

5.4.1 Simulationsmodell

Abbildung 5.7 zeigt den generellen Ablauf einer Simulation. Zunächst werden Simulationseingaben generiert. Auf Basis dieser Eingabe werden dann eine Reihe von aufeinander folgenden Simulationsläufen durchgeführt, die solange wiederholt werden, bis sich ein stabiler Zustand einstellt, was man häufig auch als Konvergieren bezeichnet. Hierbei ist jedoch nur bedingt das strenge Zulaufen auf einen Grenzwert gemeint. Die Simulation gilt als konvergiert, wenn

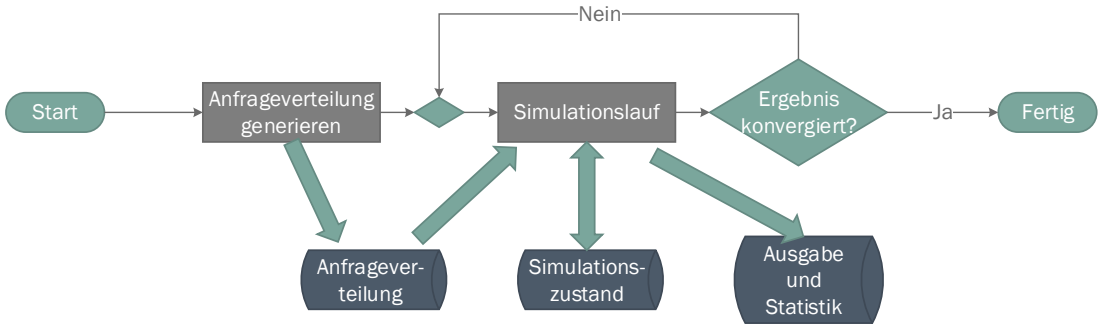


Abbildung 5.7: Schematische Visualisierung des Ablaufs einer Simulation.

aufeinander folgende Simulationsläufe ähnliche Ergebnisse erzielen, die nur um einen kleinen Wert ϵ voneinander abweichen. Die genaue Metrik wird später noch vorgestellt. Abschließend werden die Ergebnisse der Simulation ausgewertet und aufbereitet. Ist die Simulation nach einer Anlaufphase stabil, können fortlaufend mehrere Tage mit unterschiedlichem Anfrageverhalten (beispielsweise am Wochenende) nacheinander simuliert werden.

Eine Cloud-Ressource (in diesem Abschnitt als vm bezeichnet) wird beschrieben durch ihren Erstellungszeitpunkt t_{create}, die Dauer ihrere Startup-Phase $\Delta t_{startup}$, ihres Zerstörungszeitpunktes t_{kill} und ihrer aktuellen Belegung num_{req}. Die Cloud-Ressource hat keine Attribute für Performance, da diese von Anwendung zu Anwendung verschieden sind und allein durch verschiedene Werte von Δt_{work} repräsentiert wird. Eine Dienstanfrage $req \in REQs$ ist charakterisiert durch den Zeitpunkt $t_{request}$, zu dem der Client die Anfrage absetzt, dem Zeipunkt t_{start}, zu dem die Anfrage tatsächlich begonnen wird abzuarbeiten, der Anfragedauer Δt_{work} und dem Zeipunkt t_{finish}, der das Ende der Bearbeitung der Anfrage markiert. Δt_{work} ist dabei die Zeitdauer, die benötigt wird, wenn die vm exklusiv zur Verfügung stünde. Falls die Anfrage req ihre zugewiesene Ressource vm mit einer anderen Anfrage teilen muss, dann ist $\Delta t_{shared} > 0$ und beschreibt den Zeitraum, den die Ressource vm nicht exklusiv zur Verfügung stand.

Es wurde eine diskrete Ereignissimulation durchgeführt, die durch ein 5-Tupel beschrieben wird $Sim(VMs, REQs, S_{init}, S_{final}, F_{alloc})$. Implementierungsdetails der Simulation werden in Abschnitt 5.4.4 genannt. Hier wird zunächst der Ablauf der Simulation beschrieben. Zu Beginn eines Simulationslaufs ist die initiale Menge verwendeter Ressourcen (VMs) und der finale Zustand leer ($VMs = S_{final} = \emptyset$). Die Simulation startet mit dem Wiederherstellen des initialen Zustands S_{init} und dem Einlesen der definierten Anfragen $REQs$ aus der entsprechenden Eingabedatei der Anfrageverteilung. S_{init} enthält eine Menge bei Start bereits ausgeführter VMs und entsprechend auch eine Menge von Anfragen, die auf diesen bei Start schon bearbeitet werden. S_{init} dient somit der Realisierung eines realistischen Simulationsanfangszustands. Während der Simulation wird jede Anfrage req entsprechend ihres Anfragezeitpunktes $t_{request}$ durch die gewählte Funktion zur Allokation F_{alloc} einer vm zugewiesen. Die Verarbeitungszeit Δt_{work} für alle aktuell bearbeiteten Anfragen wird jede Sekunde verringert. Dabei wird die Belegung aller VMs berücksichtigt. Für Anfragen, die exklusiv bearbeitet werden, wird Δt_{work} um 1 s verringert. Für Anfragen, die sich eine vm teilen, wird

Δt_{work} um $\frac{1}{load(vm)}$ s verringert, da in der selben Zeit weniger CPU-Zeit pro Anfrage zur Verfügung stand als bei exklusiver Nutzung. Fertig abgearbeitete Anfragen ($\Delta t_{work} == 0$) werden von der vm entfernt und die vm wird eventuell wieder als frei angesehen, falls keine anderen Anfragen mehr darauf ausgeführt werden. In periodischen Abständen wird auch die die Anzahl $num_{standby}$ von Reserve-VMs durch das Starten oder Beenden von Ressourcen entsprechend angepasst. Nicht mehr benötigte Ressourcen aus VM_{free} werden noch so lange nicht terminiert, wie sie bereits durch die stündliche Abrechnung bezahlt sind. Dies macht es möglich, eventuell weitere Anfragen auf die noch verfügbaren Ressourcen zuzuweisen. Nach Beendigung des Simulationslaufs enthält VMs alle vm, die während der Simulation benutzt wurden einschließlich der durch die Simulation ermittelten Werte für t_{create} und gegebenenfalls t_{kill}. Der Endzustand eines Simulationslaufs S_{final} enthält alle am Ende des Simulationstages (24 Uhr) noch laufenden vm und die noch in Bearbeitung befindlichen Anfragen req. S_{final} kann als Eingabe S_{init} für den nächsten Simulationslauf dienen. Somit ist es möglich, eine fortlaufende Simulation durchzuführen. Außerdem enthält $REQs$ am Ende der Simulation die aktualisierten simulierten Werte für t_{start} und t_{finish} für alle $req \in REQs$.

5.4.2 Ressourcenallokationsstrategie

Das Ziel der hier vorgestellten Allokationsstrategie ist es, stets genügend freie Ressourcen vorzuhalten, um alle eingehenden Anfragen sofort und exklusiv befriedigen zu können. Das Optimum bezüglich der dafür entstehenden Kosten wäre erreicht, wenn stets genau so viele VM-Instanzen aktiv sind, wie Anfragen beantwortet werden müssen. Da Anfragen jedoch nicht vorhersagbar sind, müssen immer einige Reserve-VMs vorgehalten werden, um eine Wartezeit bei der Bearbeitung von Anfragen während des Startens neuer Instanzen zu verhindern. Deren Anzahl wird mit $num_{standby}$ angegeben. Die nachfolgend beschriebenen Simulationsläufe wurden mit verschiedenen Werten für die Anzahl von Reserve-VMs durchgeführt. Dabei wurde ein Wert von 10 % Reserve-VMs im Vergleich zur aktuellen Anfragemenge als ausgewogen ermittelt. Der Wert sollte je nach Anfrageverteilung, Anfragehäufigkeit und Anfragelänge angepasst werden. Für die Simulation ist es wichtig, dass der Wert für alle Simulationsläufe konstant gehalten wird, um die Vergleichbarkeit der Ergebnisse zu gewährleisten. Außerdem läuft zu jedem Zeitpunkt t_{sim} mindestens immer eine vm. $REQ_{t_{sim}}$ beschreibt die Anzahl der zum Zeitpunkt t_{sim} aktiven Anfragen:

$$REQ_{t_{sim}} = \{req \quad \text{mit} \quad req \in REQs \quad \text{und} \quad req.t_{start} \leq t_{sim} < req.t_{finish}\}, \tag{5.7}$$

$$num_{standby} = \begin{cases} 1 & : \text{falls} \quad |REQ_{t_{sim}}| = 0 \\ \lceil |REQ_{t_{sim}}| \cdot 0,1 \rceil & : \text{sonst.} \end{cases} \tag{5.8}$$

Trotzdem kann es passieren, dass zu einem bestimmten Zeitpunkt eine Anfrage gestellt wird, für die keine freie vm mehr zur Verfügung steht. Die beiden vorgestellten Allokationsmethoden F_{alloc} unterscheiden sich in der Strategie, wie sie genau dieses Problem lösen.

Die Strategie $F_{alloc\,exklusive}$ weist nie mehr als eine Anfrage auf eine vm zu. Als Folge kann es vorkommen, dass erst eine neue VM-Instanz gestartet werden muss, um die Anfrage zu bearbeiten. Im schlimmsten Fall dauert es also $\Delta t_{startup}$, bis die Bearbeitung der Anfrage beginnt. Da die Anfrage dann jedoch exklusiv verarbeitet wird, kommt es zu keiner weiteren Verzögerung ($\Delta t_{shared} = 0$). Der maximale zeitliche Overhead für diese Variante ist demnach

Algorithmus 1 $F_{alloc\,exklusive}$ (exklusive Allokationsstrategie)

Require: *req* {Zuweisung einer Anfrage auf eine VM }
  $REQs = REQs \cup req$
  if $|VM_{free}| < 1$ **then**
    $vm_n =$ **new** $vm()$
    $VMs = VMs \cup \{vm_n\}$
    while $|VM_{free}| < 1$ **do**
      wait(1s)
    end while
  end if
  $F_{assign}(vm_x,req)$ mit $vm_x \in VM_{free}$
  $VM_{free} = VM_{free} \setminus vm_x$
  $F_{adjust}(REQs, VMs)$

Algorithmus 2 $F_{alloc\,immediate}$ (unmittelbare Allokationsstrategie)

Require: *req* {Zuweiseung einer Anfrage auf eine VM }
  if $|VM_{free}| < 1$ **then**
    $F_{assign}(vm_n,req)$ with $vm_n = \underset{vm_n \in VMs}{\operatorname{argmin}} \left(load(vm_n)\right)$
  else
    $F_{assign}(vm_x,req)$ with $vm_x \in VM_{free}$
    $VM_{free} = VM_{free} \setminus vm_x$
  end if
  $F_{adjust}(REQs, VMs)$

$MAX(\Delta t_{overhead}) = \Delta t_{startup}$. Der Pseudocode dieser Allokationsmethode $F_{alloc\,exklusive}$ ist in Algorithmus 1 angegeben.

Die Allokationsstrategie $F_{alloc\,immediate}$ weist eine Anfrage immer unverzüglich auf eine *vm* zu. Daraus ergibt sich $\Delta t_{alloc} = 0$. Sollte dabei keine freie *vm* mehr verfügbar sein ($|VM_{free}| = 0$), wird die Anfrage auf die *vm* mit der geringsten Anzahl von aktuell bearbeiteten Anfragen zugewiesen. Durch die Überladung einer *vm* kann jedoch Δt_{shared} eventuell stark ansteigen, wodurch man keine obere Schranke für die Bearbeitungszeit mehr angeben kann. Diese unmittelbare Strategie ist in Algorithmus 2 angegeben. Beide Allokationsstrategien werden im Folgenden durch Simulation miteinander verglichen.

5.4.3 Generierung von Simulationseingaben

Die als Simulationseingabe nötigen Anfragen wurden zufällig über den Simulationszeitraum eines Tages zwischen 0 und 24 Uhr verteilt. Jede Anfrage *req* wird dabei durch den Zeitpunkt des Absetzens der Anfrage $t_{request}$ und die Dauer der Anfrage Δt_{work} charakterisiert. Die Dauer jeder einzelnen Anfrage kann prinzipiell verschieden sein. Um die Ergebnisse verschiedener Verteilungen und Allokationsstrategien besser vergleichen zu können, wurden jedoch für eine generierte Simulationseingabe für alle Anfragen die gleiche Länge gewählt.

Zunächst wurden verschiedene synthetische Verteilungen angenommen, um die Simulation auf Plausibilität und Funktionalität zu überprüfen. Dazu sind in Abbildung 5.8 verschiedene Anfrageverteilungen und die zugehörigen Simulationsergebnisse dargestellt. Es wurden jeweils 100000 Anfragen pro Tag mit je 20 min Anfragelänge benutzt. Zu sehen sind jeweils die Anzahl gleichzeitiger Anfragen (blau gepunktete Linie), die Anzahl gerade in Bearbeitung befindlicher Anfragen (lila gepunktete und rote Linie) und die dazu benötige Anzahl von VM-Ressourcen (grüne gestrichelte und hellblau gestrichelte Linie) für die beiden Allokationsstrategien. Im

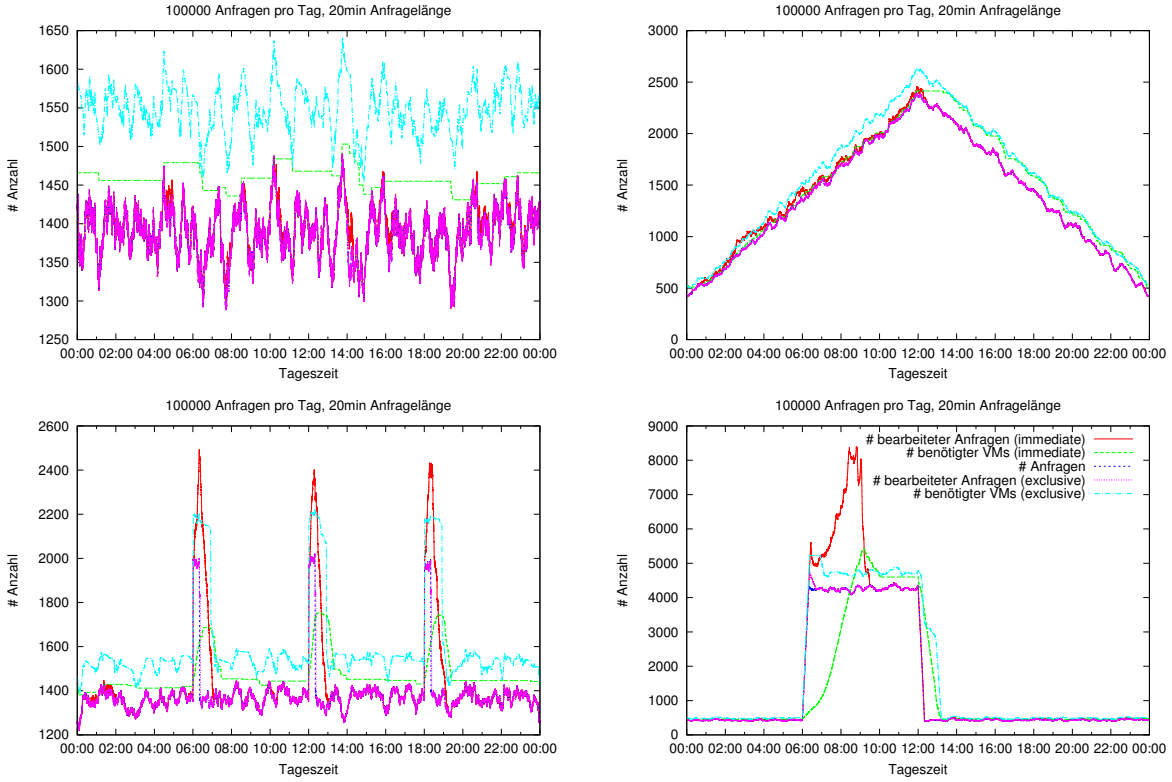


Abbildung 5.8: Visualisierung verschiedener Anfrageverteilungen und zugehörige Simulationsergebnisse.

besten Fall sollten sich die Kurven für die Anzahl gleichzeitiger Anfragen und die Anzahl gerade bearbeiteter Anfragen decken. Dies würde bedeuten, dass keine Verzögerung bei der Bearbeitung von Anfragen auftritt und stets genügend VM-Ressourcen zur Verfügung stehen, um jede Anfrage exklusiv auf eine VM zuzuweisen.

In Abbildung 5.8 (oben links) wurden alle Anfragen mit der gleichen Wahrscheinlichkeit über den ganzen Tag verteilt. Aus der Simulation ergibt sich dann, dass alle Anfragen ohne größeren Overhead bearbeitet werden können. Die exklusive Allokationsstrategie benötigt hier jedoch mehr VM-Ressourcen. In Abbildung 5.8 (oben rechts) wurde die Anfragewahrscheinlichkeit gegen Mittag erhöht und nachmittags spiegelverkehrt wieder abgesenkt. Dadurch ergibt sich eine Spitze um 12 Uhr Mittags. Hier ist links der Spitze zu erkennen, dass die Linie der aktuell in Bearbeitung befindlichen Anfragen für die unmittelbare Allokationsstrategie (rote Linie) häufig leicht oberhalb der Linie der gleichzeitigen Anfragen (blau gepunktete Linie) liegt. Dies liegt daran, dass die unmittelbare Allokationsstrategie beim Allokieren neuer Ressourcen etwas träger reagiert als die exklusive Strategie. Auf der abfallenden Seite ist die Performance beider Strategien nahezu gleich. Um die beobachteten Phänomene beim Anstieg der Anfragehäufigkeit weiter zu untersuchen, wurden in Abbildung 5.8 (unten links) drei kurze Phasen mit einer sprunghaft ansteigenden und abfallenden Anfragehäufigkeit eingebaut. Hier ist deutlich zu erkennen, dass dabei die Bearbeitungsdauer von Anfragen bei der unmittelbaren Allokation stark verzögert wird. Bei der exklusiven Allokation ist dies kaum zu beobachten, jedoch werden bei dieser Strategie sehr viele VM-Ressourcen benötigt, um dem

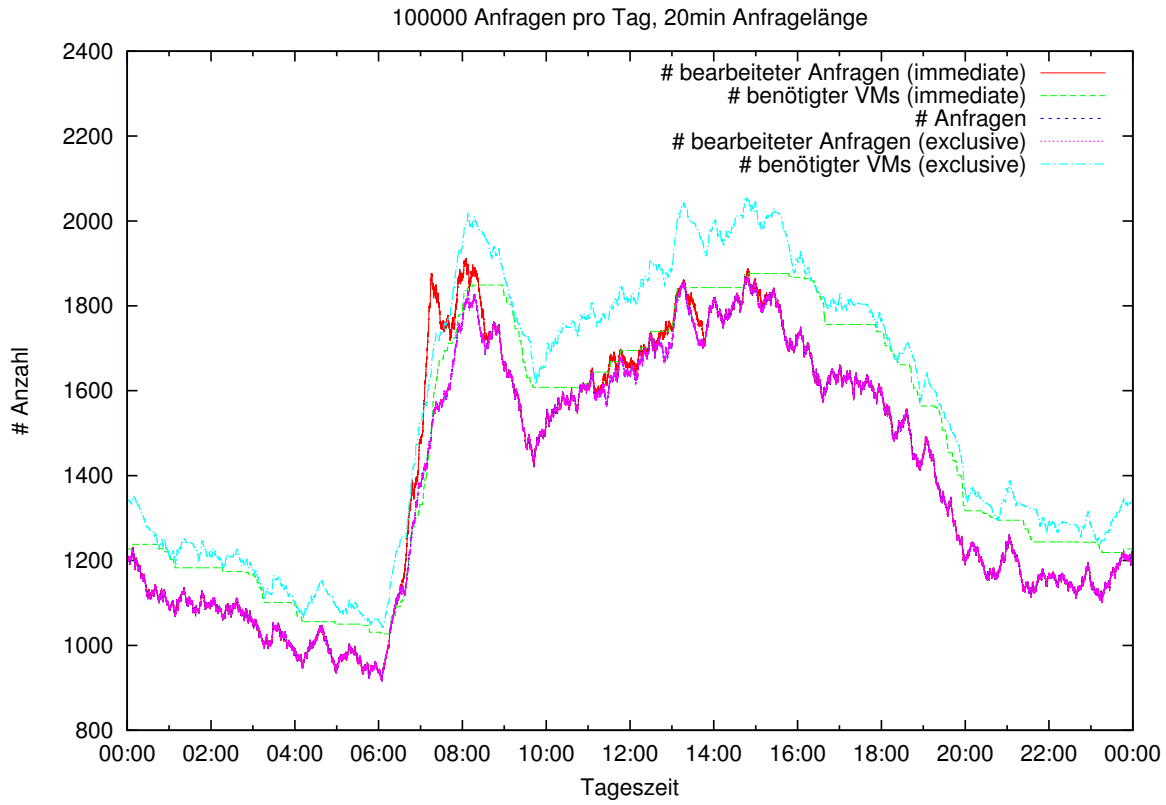


Abbildung 5.9: Verteilung von 100000 simulierten Anfragen und die korrespondierende VM Allokationen für verschiedene Allokationsstrategien.

starken Anfrageanstieg entgegenzuwirken. Noch deutlicher sichtbar wird dieser Effekt in Abbildung 5.8 (unten rechts), in der die kurzen Peaks zu einem langen Plateau zusammengefasst wurden. Dabei lässt sich jedoch auch erkennen, dass nach einer gewissen Zeit die Anpassung der Anzahl von VM-Ressourcen auch für die unmittelbare Allokationsstrategie abgeschlossen ist. Abschließend kann man anhand der synthetischen Fälle das Verhalten der beiden Allokationsstrategien gut nachvollziehen. Die unmittelbare Allokationsstrategie reagiert auf Anstiegen sehr träge, verbraucht im Gegenzug aber auch weniger Ressourcen. Besonders gute Eigenschaften weist die unmittelbare Strategie nur auf, wenn die Anzahl Anfragen nahezu gleichverteilt ist.

Um die zufällige Verteilung realistischer zu machen und Spitzen zu simulieren, wird für die tatsächliche Simulation eine Verteilungsfunktion wie in [SKS+11] verwendet. Hierbei wird beachtet, dass es wahrscheinlicher ist, dass eine Anfrage tagsüber abgesetzt wird als nachts. Eine beispielhafte Verteilung von 100000 Anfragen (Δt_{work}=20 min) ist in Abbildung 5.9 dargestellt. In diesem realistischen Verlauf lassen sich das Verhalten an einem starken Anstieg um ca. 8 Uhr, das Verhalten an einer Spitze um ca. 14 Uhr und die gleichverteilte Platzierung von Anfragen zwischen ca. 20–24 Uhr erkennen. Unter Verwendung dieser realistischen Verteilung werden nachfolgend Simulationen für verschiedene Anzahlen von Anfragen pro Tag und für verschiedene Anfragelängen durchgeführt.

5.4.4 Implementierung des Simulators

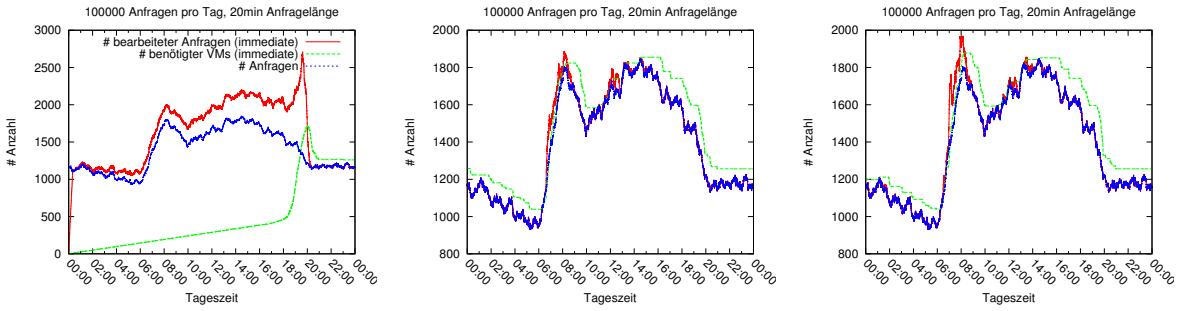
Da für die hier durchgeführten Simulationen kein bereits vorhandener Simulator verwendet wurde, sondern eine Eigeneentwicklung, wird nachfolgend noch auf einige Details des Simulators eingegangen. Der implementierte Simulator ist an das Design von *Javasim* angelehnt [BS96]. Da jedoch auch die parallele Verarbeitung der Simulationsläufe gewünscht war, wurde nicht direkt auf die Quellen zurückgegriffen. Ein sehr leistungsfähiger und erweiterbarer Simulator für Cloud-Umgebungen ist *Cloudsim* [CRB11]. *Cloudsim* ist jedoch dazu entworfen worden, die Prozesse und Architektur innerhalb einer Infrastruktur-Umgebung abzubilden. Dazu zählt beispielsweise das Platzieren von virtuellen Maschinen auf reale Server. Somit agiert *Cloudsim* auf einem anderen Abstraktionslevel. Für das hier dargestellte Simulationsmodell aus Anwendungssicht wird diese feingranulare Modellierung nicht benötigt. Es wirkt sich sogar negativ auf die Performance aus. Zum Vergleich wurde aufbauend auf der Modellierung der *Amazon EC2* Struktur, die durch die Arbeit [Feh12] schon vorhanden war, ebenfalls eine Simulationsvariante mit *Cloudsim 3.0*[2] implementiert. Hierbei stellte sich jedoch heraus, dass die Performance von *Cloudsim* nicht ausreichte. Für die Simulation von 1000 Anfragen pro Tag mit einer Länge von je 20 Min benötigte die *Cloudsim*-basierte Variante bereits ca. 75 s, für 10000 Anfragen pro Tag schon 2 min 8 s. Die gleichen Simulationen wurden von dem selbst implementierten Simulator in 1 s bzw. 2,5 s durchgeführt. Damit ergibt sich ein Gewinn von ca. Faktor 75–100. Somit lassen sich mit der selbst implementierten Variante auch sehr umfangreiche Simulationen in überschaubarer Zeit durchführen.

Der allgemeine Simulationsablauf ist bereits in Abbildung 5.7 dargestellt. Jede Simulation besteht dabei aus mindestens drei einzelnen Simulationsläufen. Für die Simulation wurde ein *Java*-Programm geschrieben, welches auf einer Menge von Dateien arbeitet. Diese Dateien werden zunächst genauer beschrieben.

Anfrageverteilung Die Anfragen für die Simulation werden gemäß der gewählten Wahrscheinlichkeitsfunktion über einen Tag verteilt. Dazu wird jeder Tag in 86400 s eingeteilt. Die Werte der Wahrscheinlichkeitsfunktion liegen zwischen 0 und 1. Um die Verteilung so fair wie möglich zu gestalten, wird zunächst für jede Sekunde gemäß der Gesamtanzahl von Anfragen und der Gesamtzahl Sekunden ausgewürfelt, ob die aktuelle Sekunde überhaupt in die Verteilung einbezogen wird. Ist dies der Fall, so wird im zweiten Schritt gemäß der Wahrscheinlichkeitsfunktion ausgewürfelt, ob eine Anfrage platziert wird. Dieses Verfahren wird solange reihum angewendet, bis die gewünschte Anzahl Anfragen verteilt wurde. Zusätzlich wird zu jeder Anfrage auch deren Dauer vermerkt. Dabei haben alle Anfragen einer Simulationseingabe die gleiche Dauer, um die Vergleichbarkeit der Ergebnisse zu gewährleisten. Diese Datei wird für alle nachfolgenden Simulationsläufe benutzt.

Ausgabedatei Die Ausgabedatei enthält nach der Simulation für jeden der betrachteten 86400 Zeitschritte die aktuelle Anzahl gerade in Bearbeitung befindlicher Anfragen. Dazu zählen auch Anfragen, die zwar schon abgesetzt, aber noch nicht auf eine VM zugewiesen wurden. Zusätzlich enthält jeder Zeitschritt auch die Anzahl aktuell instanziierter VM-Ressourcen. Diese Datei wird für jeden Simulationslauf neu berechnet und dient vornehmlich der Visualisierung der Ergebnisse.

[2] Verfügbar unter Onlineressource `https://code.google.com/p/cloudsim/` (abgerufen am 16.2.2014)



0.Lauf (Abweichung Vorlauf −−) 1.Lauf (Abweichung Vorlauf +193 %) 2.Lauf (Abweichung Vorlauf +0,8 %)

Abbildung 5.10: Annäherung einer Folge von Simulationen unter Verwendung der unmittelbaren Ressourcenallokationsstrategie.

Simulationszustand Um mehrere Simulationsläufe aufeinander folgender Tage durchführen zu können, enthält diese Datei den Endzustand der Simulation als Eingabe für den nächsten Tag. Dies ist wichtig, damit Simulationen mit vielen Anfragen auch konvergieren und stabile Ergebnisse liefern. Diese Datei ist also vor dem ersten Simulationslauf leer und enthält sonst die zum Ende der Simulation noch offenen Anfragen und gestarteten VMs. Dieser Zustand wird zu Beginn des nächsten Simulationslaufs eingelesen und wiederhergestellt.

Statistikausgabe Diese Datei enthält die während eines Simulationslaufes gesammelten Statistikinformationen. Dazu zählen:

- die Gesamtlaufzeit aller benötigten VM-Instanzen,
- die maximale und durchschnittliche Anzahl gleichzeitig instanziierter VMs,
- die Anzahl tatsächlich vollständig bearbeiteter Anfrage,
- die Summe der Ausführungszeit aller vollständig bearbeiteten Anfragen,
- die maximale und durchschnittliche Anfragebearbeitungszeit,
- und die maximale und durchschnittliche VM-Allokationszeit.

Die Simulation wird dadurch bearbeitet, dass auftretende Ereignisse nach ihrer zeitlichen Abfolge durch einen einzelnen Thread bearbeitet werden. Zu diesen Ereignissen zählen beispielsweise das Starten einer Anfrage, das Beenden einer Anfrage und das Beenden einer VM. Ausgangspunkt sind die durch das Einlesen des Simulationszustands und der Anfrageverteilung gespeicherten Ereignisse. Zur Laufzeit kommen dazu je nach Simulationslauf weitere Ereignisse wie das Starten und Beenden von VMs hinzu. Besonders kritisch ist bei diskreten Ereignissimulationen die Abspeicherung und zeitliche Sortierung der Ereignisse, um das nächste Ereignis zu bestimmen. Dazu wird im Allgemeinen ein Fibonacci-Heap verwendet. Auf diesem speziellen Heap sind vor allem die oft benötigte Einfüge- und Entnahmeoperation von konstanter asymptotischer Laufzeit *O(1)*. Weitere Informationen zu dieser Datenstruktur sind beispielsweise in [CLRS01, S. 476ff.] zu finden. Für *Java* gibt es eine Reihe von Implementierungen dieser Struktur. Für den Simulator wurde die Implementierung von *JGraphT*[3] verwendet. Startet man eine neue Simulation, so muss diese zunächst einen stabilen Zustand

[3]Verfügbar unter Onlineressource `http://jgrapht.org/` (abgerufen am 16.2.2014)

Tabelle 5.2: VM-Bedarf für verschiedene Allokationsstrategien (gleichzeitig aktive VMs)

Anfragelänge	2s					10s					60s					300s					1200s				
Anfragen pro Tag	100	1000	1E+4	1E+5	1E+6	100	1000	1E+4	1E+5	1E+6	100	1000	1E+4	1E+5	1E+6	100	1000	1E+4	1E+5	1E+6	100	1000	1E+4	1E+5	1E+6
unmittelbare Allokation (max)	3	4	5	11	80	3	4	9	42	273	3	6	29	162	1022	4	12	67	530	4902	6	30	223	1931	20157
unmittelbare Allokation (avg)	1,81	2,52	3,83	7,69	45,22	1,73	2,81	5,89	22,76	154,19	1,80	3,57	14,62	97,21	752,47	2,07	6,83	44,27	376,24	3674,83	3,00	17,84	152,57	1481,46	14732,23
exklusive Allokation (max)	2	3	5	7	182	3	4	11	49	199	3	8	25	138	1091	4	15	66	547	5311	8	31	235	2079	20768
exklusive Allokation (avg)	1,67	2,15	3,40	6,57	43,28	1,65	2,60	5,73	22,72	151,09	1,68	3,85	15,54	91,39	797,48	2,11	7,58	46,95	398,87	3852,73	3,35	19,72	165,44	1546,87	15330,45

erreichen. Dazu wird die Simulation mehrfach nacheinander durchgeführt und die Ausgabe eines Simulationslaufs als Eingabe des nächsten Simulationslaufs benutzt. Die Simulation wurde beendet, wenn die Ergebnisse von zwei aufeinander folgenden Simulationsläufen sehr ähnlich waren. Als Metrik wurde die Summe der insgesamt verbrauchten VM-Laufzeit gewählt. Als ähnlich gelten die Simulationsläufe in dieser Arbeit, wenn sich die verbrauchte VM-Laufzeit um weniger als einen Wert ϵ von 1 % unterscheidet. Abbildung 5.10 visualisiert das Annähern der einzelnen Simulationsläufe, welches besonders bei der unmittelbaren Allokationsstrategie mit vielen Anfragen lange dauern kann, da weniger häufig neue Maschinen gestartet werden. Ein ϵ von 1 % führte zum zuverlässigen konvergieren, gemeint ist hier der Abbruch der Simulation durch das Unterschreiten von ϵ, aller in dieser Arbeit durchgeführten Simulationen. Die Allokationsstrategien sind so implementiert, dass sie stets versuchen, die Anzahl von VM-Instanzen an die Anzahl konfigurierter Anfragen anzupassen. Für einige Simulationsläufe führt dies dazu, dass die Ausgabe der Simulation wieder exakt der Eingabe entspricht, beispielsweise bei gleichbleibenden Anfrageanzahlen über einen ganzen Tag. Somit wären auch aufeinander folgende Simulationsläufe exakt gleich. Für sehr viele Anfragen ist es jedoch unwahrscheinlich, dass die Ausgabe erneut der Eingabe entspricht. Somit weichen aufeinander folgende Simulationsläufe immer wieder etwas ab. Darum wird die Simulation in dieser Arbeit als konvergiert betrachtet, wenn einzelne Simulationsläufe nicht mehr als den Wert ϵ voneinander abweichen.

Jeder Simulationslauf ist in *Java* als ausführbares Objekt (`Runnable`) gekapselt. Somit können unabhängige Simulationen, beispielsweise für verschiedene Anfrageanzahlen parallel verarbeitet werden. Das einmalige Ausführen aller in Tabelle 5.2 visualisierten Simulationskonfigurationen konnte so auf dem Dell Latitude Laptop (siehe Anhang A) unter Ausnutzung von 8 Threads in ca. 28 min fertiggestellt werden.

5.4.5 Simulationsergebnisse

Es wurden Simulationen mit 100 bis 1000000 Anfragen pro Tag ($\frac{\#_{Anfragen}}{Tag}$) durchgeführt. Außerdem wurde dabei Δt_{work} der Anfragen $REQs$ zwischen 2 s bis zu 20 min variiert. Das Variieren von Δt_{work} kann dabei nicht nur als Anfragen unterschiedlicher Länge verstanden werden, sondern es kann auch verschiedene VM-Typen repräsentieren, die sich in der Geschwindigkeit unterscheiden. Ein kürzeres Δt_{work} kann also auch die gleiche Anfrage auf einer schnelleren Maschine repräsentieren. Jede Simulation wurde mehrfach durchgeführt. Dabei wurde S_{final} für die nachfolgende Simulation als S_{init} benutzt. Die Simulationen wurden zunächst so oft durchgeführt, bis sich die Simulation stabilisierte. Danach wurden 10 weitere Läufe nach der gleichen Vorgehensweise gestartet. Die hier präsentierten Ergebnisse zeigen die durchschnittlichen Werte dieser 10 Simulationsläufe.

Abbildung 5.9 zeigt die Simulationsergebnisse für 100000 $\frac{\#_{req}}{day}$ und Δt_{work} von 20 min. Das Diagramm stellt, wie auch in Abbildung 5.8, die Summe aller gleichzeitig bearbeiteten Anfragen REQ und der gleichzeitig aktiven virtuellen Maschinen VMs über den Zeitraum eines Tages für beide Allokationsstrategien dar. Dabei bedeutet der Abstand zwischen der Simula-

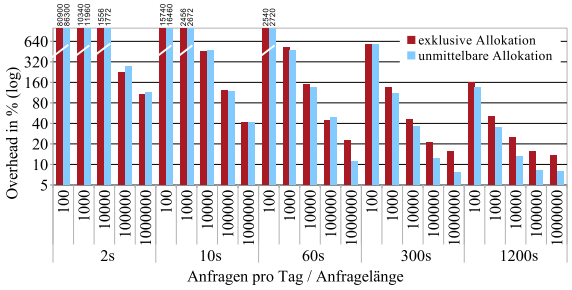


Abbildung 5.11: Ungenutzte VM-Laufzeit (Kostenoverhead)

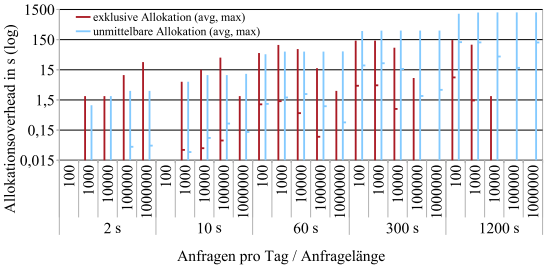


Abbildung 5.12: Zeitlicher Allokationsoverhead $\Delta t_{overhead}$

tionseingabe der Anfragen (dunkelblaue Linie) und den verwendeten VMs (grüne/hellblaue Linie) den Overhead an Ressourcen für die jeweilige Allokationsstrategie. Es wird sichtbar, dass die Linie der simulierten Anfragen für die exklusive Allokationsstrategie (violette Linie) beinahe mit der Linie der Simulationseingaben identisch ist, was eine sehr gute Qualität der Bearbeitung der Anfragen bedeutet. Allerdings werden dazu auch etwas mehr Ressourcen verwendet als bei der unmittelbaren Strategie. Die Linie der simulierten Anfragen für die unmittelbare Allokationsstrategie (rote Linie) ist hingegen oft über der Line der Simulationseingaben, was für einzelne Anfragen auf ein deutlich messbares $\Delta t_{overhead}$ schließen lässt. Dies ist vor allem der Fall, wenn die Anzahl der Anfragen in kurzer Zeit stark ansteigt, beispielsweise zwischen 6 und 10 Uhr. Das bessere Reaktionsverhalten von $F_{alloc\,exklusive}$ ist vor allem darauf zurückzuführen, dass hier im Vergleich zu $F_{alloc\,immediate}$ nicht nur periodisch die Anzahl der Ressourcen erhöht wird, sondern auch jedes Mal, wenn keine Ressourcen mehr zur Verfügung stehen. Dadurch reagiert $F_{alloc\,exklusive}$ schneller auf große Spitzen in der Anfrageanzahl, nutzt zur Beantwortung jedoch auch mehr Ressourcen. Weitere Visualisierungen der durchgeführten Simulation mit anderen Parametern sind in Anhang C dargestellt.

Tabelle 5.2 zeigt die durchschnittliche und maximale Anzahl von gleichzeitig aktiven VMs für beide Allokationsstrategien unter verschiedenen Bedingungen für $\frac{\#_{Anfragen}}{Tag}$ und Δt_{work}. Beide Strategien haben relativ ähnliche Anforderungen. Abbildung 5.11 zeigt, dass der Overhead in Bezug auf ungenutzte VM-Laufzeit für viele Anfragen pro Tag ($\frac{\#_{Anfragen}}{Tag}$) und lange Anfragen ($\Delta t_{work}$) am Kleinsten ist. Der Overhead reflektiert auch die Kosteneffizienz der Strategie, da ungenutzte VM-Laufzeit trotzdem bezahlt werden muss, jedoch nicht auf den Anwendungs-Nutzer umgelegt werden kann. Weniger ungenutzte VM-Laufzeit bedeutet also einen geringeren Kostenoverhead, was zu einem geringeren Preis pro Anfragesekunde führt.

Ein wesentlicher Vorteil der exklusiven Allokationsstrategie wird in Abbildung 5.12 deutlich. Der Overhead $\Delta t_{overhead}$ kann mit dieser Strategie auf $\Delta t_{startup}$=150 s begrenzt werden. Dies bedeutet, dass obwohl ein Nutzer eventuell kurz warten muss, bis die Bearbeitung seiner Anfrage beginnt, die Anfragezeit durch eine ober Schranke begrenzt wird. Dadurch wird die erwartete Anfragezeit vorhersehbarer. Außerdem lässt sich in Abbildung 5.12 erkennen, dass der Overhead bei steigender Anzahl an Anfragen pro Tag und steigender Anfragedauer abnimmt. Erreicht wird dies durch eine bessere VM-Wiederverwendungswahrscheinlichkeit, da mehr VMs aktiv sind. Außerdem ist absolut auch die Anzahl $num_{standby}$ von Reserve-VMs höher. Wie bereits in Abbildung 5.9 dargestellt wurde, hat die unmittelbare Allokationsstrategie Probleme, die Anzahl Standbyinstanzen zu regulieren, wenn die Anzahl von Anfragen stark ansteigt. Ein hoher Overhead $\Delta t_{overhead}$ ist die Folge. Diesen Effekt kann

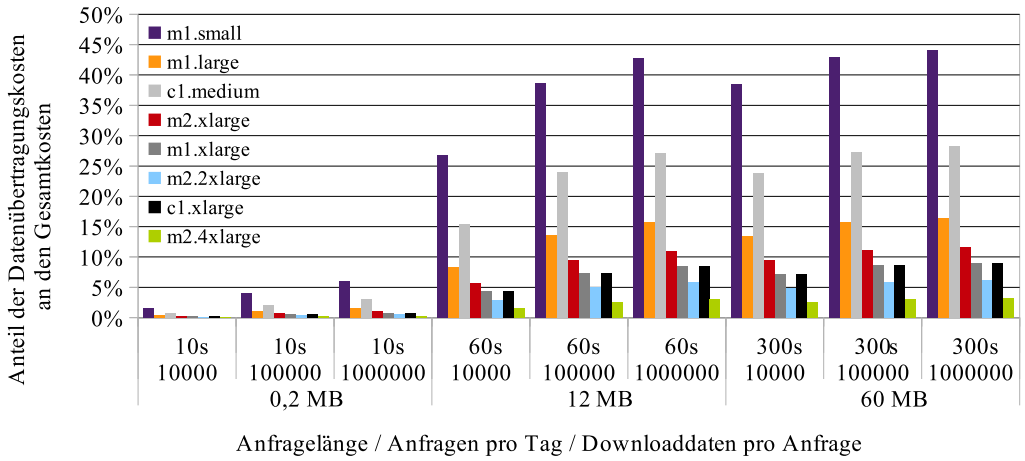


Abbildung 5.13: Datenübertragungskosten im Vergleich zu gesamten Cloud-Nutzungskosten.

man auch an den maximalen Werten in Abbildung 5.12 wiedererkennen. Trotzdem lässt sich auch feststellen, dass die durchschnittlichen Werte für den Overhead viel niedriger sind als die Maximalwerte und dass die durchschnittlichen Werte beider Strategien recht ähnlich sind.

5.4.6 Datenübertragungskosten

Bei der Nutzung von Infrastrukturdiensten wie VMs fallen nicht nur Kosten für die Nutzung der eigentlichen Ressourcen an, auch die Datenübertragungskosten sind zu berücksichtigen. Nachdem der Kostenoverhead für die Allokationsstrategien bestimmt wurde, ist zudem interessant, in welchem Verhältnis zur eigentlichen Nutzungsgebühr die Datenübertragungskosten stehen. Dazu wird bezüglich der Kosten exemplarisch die Situation im *Amazon EC2* Rechenzentrum in Irland betrachtet. Datenübertragungskosten werden quasi doppelt abgerechnet. Einerseits finden sie sich bereits in der VM-Nutzungsdauer wieder, da während Daten zur VM übertragen werden, diese auch aktiv sein muss und somit Geld kostet. Zusätzlich wird jedes Byte, welches aus dem Amazon Rechenzentrum heruntergeladen wird, in verschiedenen Stufen, je nach Menge der Gesamtübertragungsdaten im Monat, abgerechnet. Der Upload von Daten ist kostenlos. Zur Vereinfachung wurde angenommen, dass für jedes angefangene GigaByte 0,12 $ berechnet werden. Diese Kosten reflektieren die pessimistischste Variante. Je nach tatsächlicher Nutzung kann es jedoch nur günstiger werden. Die genauen Abrechnungsmodalitäten für Datenübertragungen zu verschiedenen Cloud-Rechenzentren sind in Anhang B angegeben.

Abbildung 5.13 zeigt die Datenübertragungskosten relativ zu den entstehenden VM-Kosten für verschiedene Simulationseingaben. Da längere Berechnungen auch etwas mehr an Download Traffic erzeugen, wurde die Datenmenge der Nutzungsdauer angepasst. Das Diagramm zeigt Datenübertragungen von 0,2 bis 60 MB. Für die kurze Übertragungsdauer sind die prozentualen Übertragungskosten besonders niedrig. Da die Datenübertragungskosten für alle verschiedenen VM-Instanztypen konstant sind, reflektiert das Diagramm jedoch auch den Kostenoverhead, wenn teurere Instanztypen verwendet werden. Weniger performante Instanz-

typen sind günstiger und kosteneffizienter, wodurch sich der relative Anteil der Datenübertragungskosten an den Gesamtkosten erhöht. Ebenso ist der Overhead für wenige Anfragen pro Tag sehr hoch, wie bereits in Abbildung 5.11 zu sehen war, wodurch die Datenübertragungskosten relativ gesehen in diesem Szenario geringer werden.

Zusätzliche Kosten, die für die Datenspeicherung anfallen, wurden nicht mit in die Berechnungen inbegriffen, da diese nur 0,0024 US cent pro Anfrage ausmachen würden. Zusammenfassend ist zu sagen, dass die Datenübertragungskosten eher einen geringen Anteil an den Gesamtkosten haben, jedoch im Gegensatz zu den Kosten für die Datenspeicherung nicht zu vernachlässigen sind. Sie werden deshalb im Folgenden bei der Berechnung der Nutzungskosten auch weiter betrachtet.

5.5 Fazit

In diesem Kapitel wurde untersucht, wie die Forderung nach einer exklusiven Ressourcenallokation für jeden Client den Kostenoverhead für die VM-Nutzung beeinflusst. Dabei wurde die Wiederverwendung von VM-Ressourcen zur Kostenreduktion als essentiell eingeführt. Danach wurde eine Architektur vorgestellt, die VM-Ressourcen über einen Broker-Server an die einzelnen Clients zuweist. In einer Simulation wurde die durchschnittliche VM-Allokationsdauer im Vergleich zu einer traditionellen Strategie mit Multiplexing von Anfragen verglichen.

Gegenüber der traditionellen unmittelbaren Ressourcenallokation stellt die exklusive Allokationsstrategie eine obere Schranke bezüglich der Allokationsdauer für eine VM nahezu sicher. Trotz dieser Verbesserung offenbarte eine Simulation, dass die exklusive Allokationsstrategie in Bezug auf den Kostenoverhead ein ähnliches Verhalten zeigt wie die unmittelbare Allokationsstrategie, also für die meisten simulierten Fälle nicht teurer ist. Für die exklusive Allokationsstrategie dauert die Allokation für die simulierten Fälle im Mittel nicht länger als ca. 10 s. Für die unmittelbare Allokationsstrategie war der Aufschlag auf die Ausführungszeit hingegen nicht nur im Mittel, sondern auch absolut, wesentlich höher.

Für die nachfolgenden Betrachtungen lässt sich im Ergebnis festhalten, dass eine VM zur Nutzung durch eine *Mobile Cloud-unterstützte Anwendung* exklusiv zugewiesen werden kann und dass die dafür notwendige exklusive Ressourcenallokation mit abschätzbarem Overhead realisierbar ist. Somit können nachfolgend Vorhersagealgorithmen über die Ausführungszeit von Applikationen entwickelt werden, die die exklusive VM-Nutzung voraussetzen.

Als Einschränkung bezüglich der Verbesserung der Granularität der Abrechnungsperiode stellte sich heraus, dass für wenige kurze Anfragen über einen Tag verteilt kaum eine Verbesserung erreicht werden kann. Erst ab ca. 10000 Anfragen/Tag, die jeweils ca. 1 min dauern, kann der Overhead auf das Doppelte begrenzt werden. Das heißt, dass eine so abgerechnete Minute Berechnungszeit circa der Nutzungsgebühr für zwei Minuten Berechnungszeit entspricht. Im Ergebnis kann nun zwar eine sekundengenaue Erfassung der Nutzungsdauer für *Mobile Cloud-unterstützte Anwendungen* implementiert werden, der Abrechnungspreis muss jedoch je nach tatsächlichem Nutzungsverhalten in der Vergangenheit angepasst werden.

Aufgrund der Wiederverwendung von VMs für verschiedene Anwendungen, muss der Programmcode durch den Client zur Laufzeit installiert werden. Dies führt zu der Einschränkung, dass die Installation und Datenübertragung des Programmcodes in angemessenem Verhältnis zur Ausführung der eigentlichen Applikation stehen muss. Abschließend wurde festgestellt, dass auch die Datenübertragungskosten nicht zu vernachlässigen sind und somit nachfolgend in die Kostenvorhersage integriert werden müssen.

6 Auswahl geeigneter Cloud-Ressourcen zur Laufzeit

Mobile Cloud-unterstützte Anwendungen nutzen Cloud-Ressourcen zur entfernten Ausführung von berechnungsintensiven Programmmodulen. Dieses Kapitel widmet sich der Auswahl der Qualität verfügbarer Cloud-Ressourcen (VMs). Das Treffen einer Auswahl ist nötig, da seitens des IaaS-Anbieters üblicherweise verschiedene Ressourcenkonfigurationen zur Verfügung stehen. Nicht alle können jedoch für eine konkrete Situation zur Laufzeit sinnvoll verwendet werden, da sie eventuell keine schnellere Ausführung ermöglichen oder die Kostenbeschränkung verletzen, die in Kapitel 3 definiert wurde. Darüber hinaus hat jeder Nutzer eventuell eigene Vorgaben bezüglich der Qualitätsauswahl. So könnte ein Nutzer etwa die bestmögliche Performance pro Geldeinheit als Ziel haben und nicht die maximal mögliche Performance.

Die Auswahl hängt von der momentan verfügbaren Netzwerkperformance ab, deren Bestimmung in Kapitel 4 behandelt wurde. Es ist beispielsweise unökonomisch, eine performante und teure Ausführungsressource auszuwählen, wenn diese dann aufgrund einer zu geringen Netzwerkperformance nicht schnell genug mit Daten versorgt werden kann. Die Ausführungsgeschwindigkeit der eigentlichen Applikation auf verschiedenen VM-Typen ist demnach auch wichtig für die Ressourcenauswahl.

In diesem Kapitel wird deshalb der Einfluss des Applikationsverhaltens auf die Auswahl der zur Verfügung stehenden Cloud-Ressourcen untersucht. Ziel ist es, eine Methode zu entwickeln, die das Applikationsverhalten approximiert und es so ermöglicht, für eine spezifische Eingabe die Laufzeit vorherzusagen. Da nicht jedes Applikationsverhalten vorhersagbar modelliert werden kann, werden auch Grenzen des Verfahrens beschrieben und Applikationen genannt, die mit dem hier beschriebenen Verfahren nicht erfasst werden können. Nachdem in Kapitel 5 die Anwendbarkeit eines exklusiven Allokationsmechanismus gezeigt wurde, wird in diesem Kapitel davon ausgegangen, dass für jede entfernte Ausführung stets eine Ressource exklusiv zur Verfügung steht. Dadurch sind bei der Ausführung einer *Mobilen Cloud-unterstützten Anwendung* Seiteneffekte durch andere gleichzeitig ausgeführte Anwendungen ausgeschlossen. Außerdem spielen die Kosten für die Nutzung von Cloud-Ressourcen und damit auch der Kostenoverhead nur eine untergeordnete Rolle. Es wird davon ausgegangen, dass VMs eines IaaS-Anbieters sekundengenau abgerechnet werden können. Somit lassen sich neben der Laufzeit auch die Kosten vorhersagen.

6.1 Auswahlstrategien

Zur Modellierung des Entscheidungsmodells zur Ressourcenauswahl ist es notwendig, *Mobile Cloud-unterstützte Anwendungen* zu kategorisieren und die Applikationen zu identifizieren, für die eine Auswahl zur Laufzeit nicht vorgenommen werden kann. Für die vom Modell erfassten Applikationen wird anschließend analysiert, wie deren Verhalten für verschiedene Eingaben und auf verschiedenen Ressourcentypen realistisch approximiert werden kann.

6.1.1 Klassifikation von Mobilen Cloud-unterstützten Anwendungen

Zunächst wird die Berechnung der Laufzeit und die allgemeine Funktionsweise von *Mobilen Cloud-unterstützten Anwendung* kurz wiederholt (siehe Kapitel 3).

Zur Repräsentation der Ausführungszeit eines auf einem entfernten Server bearbeiteten Programmabschnitts wurde in Gleichung (3.13) die Variable Δt_{work} eingeführt. Sie erhält ihren Wert durch Berechnung der Laufzeitfunktion $lauf(R, p_i.e_i)$. Die Ausführungszeit ist demnach von der Ressource R, dem Applikationscode p_i und der eigentlichen Eingabe e_i abhängig. Da das Laufzeitverhalten jedoch für verschiedenen Programmcode auf verschiedenen Ressourcen unterschiedlich sein kann, wird intern für jede Kombination aus R und p_i eine separate Laufzeitfunktion $lauf_{R,p_i}(e_i)$ benutzt, die nur von der Eingabe abhängt. Will man $lauf(...)$ approximieren und zur Vorhersage der Laufzeit benutzen, so ist zunächst für jede Kombination aus Programmmodul p_i und Ressource R eine separate Laufzeitfunktion $lauf_{R,p_i}(e_i)$ zu erstellen. Hierzu kann die Methode der Regressionsanalyse eingesetzt werden, auf die später noch eingegangen wird. Als Ressourcen R werden die verfügbaren VMs aus dem Portfolio des verwendeten IaaS-Anbieters eingesetzt. Anforderungen an diese Ressourcen werden später beschrieben. Die in Kapitel 4 vorgeschlagene Kommunikation mittels Pipelining auf einer persistenten TCP-Verbindung impliziert die Kapselung des entfernt auszuführenden Programmcodes in einzelne Tasks. Je nach Interpretation der Laufzeitfunktion $lauf(...)$ kann diese für jede Task einzeln (p_i) oder als Komposition für die Gesamtausführung aller Tasks als Programm P betrachtet werden (siehe Kapitel 3.1.1).

Nicht für jedes taskbasierte Programm kann eine Laufzeitvorhersage realisiert werden. Deshalb werden in diesem Kapitel taskbasierte Applikationen weiter differenziert. Das Applikationsverhalten kann unterschieden werden in *benutzerterminierte* und *eingabeterminierte* Applikationen. Eine eingabeterminierte Applikation endet, wenn die Eingabe verarbeitet wurde. Die Eingabe ist hier also vollständig bekannt und die Laufzeit somit vorhersagbar. Ein Beispiel ist die Ausführung einer Gesichtserkennungssoftware auf einem einzelnen Foto. Für eingabeterminierte Programme wird die Laufzeitfunktion $lauf_{single}(\dots)$ verwendet, welche eine Laufzeit in Sekunden angibt. Führt man die Gesichtserkennungssoftware jedoch auf sehr vielen Bildern oder auf einem laufenden Kamerabilddatenstrom aus, so terminiert nicht die Eingabe, sondern der Benutzer die Applikation, wenn er abbrechen möchte. Dabei handelt es sich um eine benutzerterminierte Applikation. Die vollständige Eingabe ist nicht bekannt. Nur für einzelne Tasks p_i kann die Laufzeit approximiert werden. Eine Laufzeitvorhersage ist hier nur für einen fortlaufenden Strom von Tasks gleicher Art und gleicher Eingabe als Rate möglich. Für benutzerterminierte Applikationen ist die Laufzeitvorhersage demnach nur eingeschränkt möglich. Für mehrere einzelne Tasks wird die Laufzeitfunktion $rate_{multi}(R, p_i, e_i)$ verwendet, welche die Anzahl von pro Sekunde ausgeführter Tasks als Rate angibt. Als Schlussfolgerung muss $lauf(\dots)$ für jede Art Applikation verschieden berechnet werden. Es wird also zunächst unterschieden in:

$$lauf_{single}(R, p_i, e_i) \quad \widehat{=} \quad lauf(R, p_i.e_i), \tag{6.1}$$

$$rate_{multi}(R, p_i, e_i) \quad \widehat{=} \quad \frac{1}{lauf(R, p_i, e_i)}. \tag{6.2}$$

Daraus resultiert auch, dass die Auswahl geeigneter Ressourcen zur Laufzeit verschiedenen Ansprüchen genügt und somit nicht gleichartig für beide Fälle durchgeführt werden kann. Diese Erkenntnis wird später beim Entwurf von Algorithmen zur Ressourcenauswahl berücksichtigt.

Tabelle 6.1: Kategorisierung von Tasks in 9 Kategorien nach deren Eigenschaften

Ausführungsmodus	**Einzelprozessortasks**	**gemischte Tasks**	**Multiprozessortasks**
unabhängig/simultan	1	8	7
teils sequentiell	2	9	6
sequentiell	3	4	5

werden für die Laufzeitvorhersage unterstützt

Die Zuordnung einer Applikation zu den beschriebenen Fällen kann aufgrund der Taskeigenschaften der Applikation erfolgen. Dazu werden die Eigenschaften der Tasks genauer untersucht. Es kann eine Unterscheidung bezüglich der Nutzung mehrerer CPU-Kerne für einzelne Tasks vorgenommen werden. Eine Applikation kann somit aus *Einzelprozessortasks*, aus *Multiprozessortasks* oder aus beiden Varianten gleichzeitig bestehen. Zudem können Tasks bezüglich ihrer Datenabhängigkeiten in die Ausführungsmodi *sequentiell* und *simultan* unterteilt werden. Bezüglich der Datenabhängigkeit der Tasks und deren Multithreadfähigkeit gibt es neun verschiedene Fälle, die in Tabelle 6.1 zusammengefasst sind. Datenabhängigkeiten in der Ausführungsstruktur verschiedener Tasks können als Taskgraphen (*Directed Acyclic Graph* (DAG)) modelliert werden [Hun09]. Existieren keine Datenabhängigkeiten zwischen Tasks, sind sie also unabhängig, so können diese simultan, also parallel, ausgeführt werden. Außerdem kann es auch Applikationen geben, bei denen die Ergebnisse einzelner Tasks aufeinander aufbauen. Die Tasks müssen dann sequentiell, also der Reihe nach, ausgeführt werden. Sequentielle Verarbeitung bedeutet, dass, bevor ein neuer Task abgesetzt wird, zunächst die Bearbeitung des Vorangegangenen abgewartet werden muss. Als dritte Variante gibt es zudem Applikationen, bei denen Tasks beider Ausführungsmodi simultan vorhanden sind.

Es können jedoch nicht alle Fälle aus Tabelle 6.1 durch die Laufzeitfunktionen $lauf_{single}$ und $rate_{multi}$ repräsentiert werden. Nachfolgend wird anhand der Nummerierung erläutert, welche Fälle aus welchem Grund unterstützt oder nicht unterstützt werden können.

- Die Kombinationen 6–9 entfallen. Die Middleware besitzt keine Schedulingkomponente, die Tasks mit verschiedener Prozessorzahl sinnvoll gleichzeitig unter Minimierung der Gesamtlaufzeit auf einer VM ausführen kann. Solche Algorithmen sind komplex und Gegenstand der aktuellen Forschung [IOY+11]. Da diese Taskanordnung aber für *Mobile Cloud-unterstützte Anwendungen* unwahrscheinlich ist und eher im wissenschaftlichen Rechnen vorkommt, sind solche komplexen Schedulingalgorithmen nicht Gegenstand dieser Arbeit. Komplexe Taskanordnungen (Taskgraphen) müssen gemäß der Datenabhängigkeiten in ihrer Ausführungsreihenfolge sequentialisiert werden, sodass sie sich auf eine der Kategorien 3–5 abbilden lassen. Dies kann beispielsweise durch eine topologische Sortierung erreicht werden.

- Die Kombination 2 entfällt. Die Laufzeitfunktionen $lauf_{single}(\dots)$ und $rate_{multi}(\dots)$ sind nur in der Lage, für sequentielle und für gleichartige simultan ausgeführte Tasks die Laufzeit vorherzusagen. Das Mischen wird nicht unterstützt. Die Vorhersage kann dennoch ermöglicht werden, wenn die Ausführungsreihenfolge sequentialisiert wird.

- Die Kombination 1 wird von $rate_{multi}(\dots)$ unterstützt. Bei dieser Variante können so viele Einzelprozessortasks simultan ausgeführt werden wie CPUs vorhanden sind.

- Die Kombinationen 3–5 werden von $lauf_{single}(\dots)$ unterstützt.

In Tabelle 6.1 sind alle Fälle, die ohne Einschränkungen durch eine Laufzeitfunktion beschrieben werden können, grau markiert. Es wird darauf hingewiesen, dass *Mobile Cloud-unterstützte Anwendungen* aller Kombinationen auf der entwickelten Middleware lauffähig sind. Jedoch kann die für die bedarfsgerechte Auswahl von Ressourcen notwendige Laufzeitvorhersage nur für die beschriebene Untermenge durchgeführt werden.

6.1.2 Laufzeitprognose von Tasks

Die Klassifikation von *Mobile Cloud-unterstützte Anwendungen* bezüglich deren Laufzeitvorhersagbarkeit hat ergeben, dass nur sequentielle Ausführungsfolgen (eingabeterminierte Applikationen) und die simultane Ausführung gleichartiger Einzelprozessortasks (benutzerterminierte Applikationen) zur Laufzeitvorhersage geeignet sind.

Für benutzerterminierte Applikationen wird eine Rate $f_{work\,multi}$ angegeben, die die Taskverarbeitungen pro Sekunde angibt. Da jeweils nur ein CPU-Kern der VM pro Task genutzt wird, kann die Bestimmung von $rate_{multi}(R, p_i, e_i)$ um die simultane CPU-Kern-Nutzung ergänzt werden. Die Verbesserung wird im Vergleich zu Gleichung (6.2) wie folgt angegeben:

$$f_{work\,multi} = rate_{multi}(R, p_i, e_i) \quad = \quad \frac{cores(R)}{lauf(R, p_i, e_i)}. \tag{6.3}$$

Dabei liefert $cores(R)$ die Anzahl verfügbarer CPU-Kerne der Ressource R zurück. Es wird davon ausgegangen, dass die tatsächliche Anzahl ausgeführter Tasks hierbei wesentlich größer ist als $cores(R)$. Für eingabeterminierte Programme muss $lauf(\dots)$ demnach nur für einzelne Tasks bestimmt werden. Anschließend wird $rate_{multi}(\dots)$ je nach CPU-Kernanzahl der Ressource R berechnet. Dies bedeutet auch, dass für VM-Typen mit gleichgroßer Einzelprozessorgeschwindigkeit (beispielsweise `c1.medium` und `c1.xlarge` (siehe Anhang Tabelle B.3)) die gleiche Funktion $lauf(\dots)$ verwendet werden kann. Die schnellere Taskverarbeitung kann jedoch aufgrund der höheren Kommunikationsanforderungen zur Datenübertragung eventuell nicht erreicht werden. Dies wird später bei der Auswahl der Cloud-Optionen berücksichtigt und wird innerhalb der Laufzeitfunktion $rate_{multi}(\dots)$ nicht betrachtet.

$lauf_{single}(R, L)$ kann die komplette Dauer eines Programmlaufs L unter Verwendung einer Menge von Programmmodulen p_n und deren zugehörigen Eingaben e_n beschreiben (siehe Kapitel 3.1.1). Somit setzt sich die Laufzeit aus mehreren Einzellaufzeiten wie folgt zusammen:

$$\Delta t_{work\,single} = lauf_{single}(R, L) \quad = \quad \sum_{n=1}^{|L|} lauf_{single}(R, p_n, e_n) \quad mit \quad (p_n, e_n) \in L. \tag{6.4}$$

6.1.3 Auswahl einer geeigneten Ressource

Wenn nun in einem konkreten Fall eine geeignete VM zur Ausführung ausgewählt werden soll, so muss für jede konfigurierte VM_x geprüft werden, ob sie die Bedingungen (3.12) und (3.18) erfüllt. Die Bedingungen stellen sicher, dass die entfernte Ausführung tatsächlich zu einer Laufzeitverkürzung führt. Außerdem darf dabei das festgelegte Kostenlimit nicht überschritten werden. Als Ergebnis erhält man eine Menge von Cloud-Optionen, unter denen der Nutzer dann wählen kann. Diese Menge von Cloud-Optionen wird als Mc bezeichnet. Eine Cloud-Option C_x wird für eingabeterminierte Applikationen beschrieben durch ein 3er-Tupel $C_x = (\mathrm{VM}_x, \Delta t_{work}, cost_x)$ bestehend aus dem VM-Typ VM_x, der entfernten Laufzeit Δt_{work}

und den dafür entstehenden Kosten $cost_x$. Für benutzerterminierte Applikationen enthält C_x anstatt Δt_{work} eine Rate $f_{work\,k}$. Nachfolgend wird beschrieben, wie die Mengen Mc_{single} für eingabeterminierte Programme und die Menge $Mc_{multiple}$ für benutzerterminierte Programme zur Laufzeit berechnet werden.

Berechnung von Cloud-Optionen für eingabeterminierte Programme

Die entfernte Ausführungszeit setzt sich zusammen aus der Zeit für die Datenübertragung Δt_{comm} und der eigentlichen Berechnungszeit Δt_{work}. Diese Zusammensetzung lässt sich auch erkennen, wenn man die Kriterien (3.12) und (3.18) betrachtet und die Variablen entsprechend für eingabeterminierte Programme anpasst. Dies ist nachfolgend dargestellt. Δt_{mobil}, Δt_{comm} und Δt_{work} können dabei aufgrund der sequentiellen Ausführung auch Summen der Verarbeitungs- und Kommunikationszeit mehrerer Tasks repräsentieren (siehe Kapitel 3.1.2 und 3.2).

$$\begin{aligned} \text{(schnellere entfernte Ausführung)} \qquad \Delta t_{mobil} &\geq \Delta t_{remote}, \\ \Delta t_{mobil} &\geq \Delta t_{comm} + \Delta t_{work}, \end{aligned} \tag{6.5}$$

$$\begin{aligned} \text{(Einhaltung der Kostenbeschränkung)} \qquad cost_{limit} &\geq cost_x, \\ cost_{limit} &\geq cost_{per\,s}(\mathrm{VM}_x) \cdot ceil(\Delta t_{remote}). \end{aligned} \tag{6.6}$$

Die Funktion $cost_{per\,s}(\mathrm{VM}_x)$ liefert die Mietkosten für die Ressource VM_x pro Sekunde zurück. $ceil(\Delta t_{remote})$ liefert die auf volle Sekunden aufgerundete Laufzeitabschätzung zurück. Für eingabeterminierte Programme, die aus einer Sequenz von Ausführungen zusammengesetzt sind, gilt zudem die Vereinfachung, dass nur die Summe der entfernten Gesamtlaufzeit kleiner als die Summe der mobilen Laufzeit sein muss. Somit ist das Kriterium (3.12) nicht mehr für alle Einzeltasks einzuhalten. Außerdem wurde in Kapitel 5.1 gezeigt, dass die Nutzungsdauer durch Wiederverwendung anstatt auf Stundenbasis auch auf Sekundenbasis abgerechnet werden kann. Diese Verbesserung wurde in Bedingung (6.6) im Vergleich zu Bedingung (3.18) berücksichtigt.

Unter Berücksichtigung der angepassten Kriterien (6.5) und (6.6) generiert Algorithmus 3 die Menge Mc aller 3er-Tupel $C_x = (\mathrm{VM}_x, \Delta t_{work}, cost_x)$ für eingabeterminierte Programme. Zunächst werden für alle verfügbaren VM-Typen die erwartete Ausführungszeit und die

Algorithmus 3 Generierung von Mc_{single}

Require: $\Delta t_{mobil}, \Delta t_{comm}, L, cost_{per\,s}$ {berechne C_x für alle VM_x des IaaS-Anbieters}
  $Mc = \{\}$
  for all $\mathrm{VM}_x \in$ Portfolio des IaaS-Anbieters **do**
    $\Delta t_{work} = lauf_{single}(\mathrm{VM}_x, L)$
    $cost_x = cost_{per\,s}(\mathrm{VM}_x) \cdot ceil(\Delta t_{comm} + \Delta t_{work})$
    if $(\Delta t_{mobil} \geq \Delta t_{comm} + \Delta t_{work}) \wedge (cost_{limit} \geq cost_x)$ **then**
      $C_x = (\mathrm{VM}_x, \Delta t_{work}, cost_x)$
      $Mc = Mc \cup \{C_x\}$
    end if
  end for

daraus entstehenden Kosten berechnet. Anschließend werden alle Optionen ausgefiltert, die nicht den Bedingungen (6.5) und (6.6) entsprechen. Übrig bleibt die Menge Mc aller möglichen Cloud-Optionen.

Berechnung von Cloud-Optionen für benutzerterminierte Programme
Für die Berechnung der Menge $Mc_{multiple}$ ergibt sich eine leicht angepasste Variante der Berechnung unter Verwendung der Taskrate f. $Mc_{multiple}$ enthält die Cloud-Optionen für die fortlaufende Ausführung eines Tasks k, charakterisiert durch (p_k, e_k), unter Verwendung der immer gleichen Eingabegröße e_k.

$$\begin{aligned} f_{mobil\,k} &\leq f_{remote\,k}, \\ f_{mobil\,k} &\leq MIN(f_{comm\,k}, f_{work\,k}), \end{aligned} \tag{6.7}$$

$$\begin{aligned} cost_{limit} &\geq cost_k, \\ cost_{limit} &\geq cost_{per\,s}(\mathrm{VM}_x). \end{aligned} \tag{6.8}$$

Algorithmus 4 generiert $Mc_{multiple}$ unter Berücksichtigung der für benutzerterminierte Programme angepassten Kriterien (6.7) und (6.8). Zunächst wird durch $rate_{multi}(\dots)$ die Rate

Algorithmus 4 Generierung von $Mc_{multiple}$

Require: $f_{mobil\,k}, f_{comm\,k}, p_k, e_k, cost_{per\,s}$ {berechne C_x für alle VM_x des IaaS-Anbieters}
  $Mc = \{\}$
  for all $\mathrm{VM}_x \in$ Portfolio des IaaS-Anbieters **do**
    $f_{work\,k} = rate_{multi}(\mathrm{VM}_x, p_k, e_k)$
    $cost_k = cost_{per\,s}(\mathrm{VM}_x)$
    if $(f_{mobil\,k} \leq MIN(f_{comm\,k}, f_{work\,k}) \wedge (cost_{limit} \geq cost_k)$ **then**
      $C_x = (\mathrm{VM}_x, MIN(f_{comm\,k}, f_{work\,k}), cost_k)$
      $Mc = Mc \cup \{C_x\}$
    end if
  end for

der Taskverarbeitung pro Sekunde für jeden verfügbaren VM-Typ berechnet. Danach wird geprüft, ob der errechnete Wert höher ist als die Taskrate, die durch die Netzwerkperformance limitiert ist. Falls nicht, wird weiter geprüft, ob die Bedingungen (6.7) und (6.8) nicht verletzt werden. Zu beachten ist hierbei, dass das Kostenlimit auch pro Sekunde zu verstehen ist, da die Gesamtlaufzeit unbekannt ist. Die nicht ausgefilterten Cloud-Optionen werden anschließend in der Menge Mc zusammengefasst und zurückgeliefert.

Für die beschriebenen Auswahlstrategien wird im Folgenden die Laufzeitfunktion $lauf(\dots)$ zur Laufzeitvorhersage bestimmt. Die weiteren Betrachtungen beschränken sich demnach auf einzelne Tasks, die durch eine einzelne Laufzeitfunktion $lauf(...)$ repräsentiert sind. Außerdem sind je nach Approximationsqualität von $lauf(\dots)$ auch die zur Auswahl stehenden VM-Typen vm_x einzuschränken, um dem Endnutzer sinnvoll unterscheidbare Cloud-Optionen anzubieten.

6.2 Regressionsanalyse zur Laufzeiterfassung

Für die Laufzeitvorhersage der Tasktypen, die in Kapitel 6.1.1 vorgestellt wurden, muss ein Zusammenhang zwischen der Eingabe und der Laufzeit hergestellt werden. Das Herstellen eines Zusammenhangs setzt voraus, dass die Funktionsweise des zu untersuchenden Tasks hinreichend bekannt ist. Die Laufzeitfunktion $lauf(...)$ wird über Messungen für verschiedene Eingaben auf allen verfügbaren VM-Typen konstruiert. Dafür kommt die Methode der Regressionsanalyse zum Einsatz.

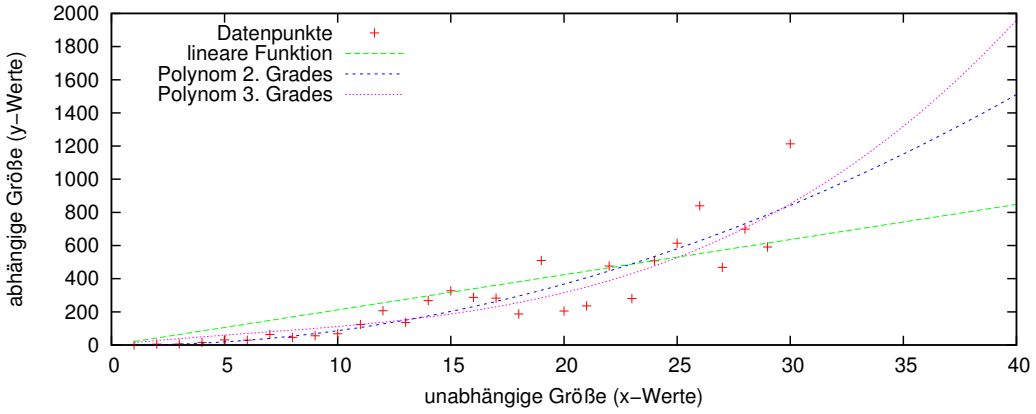


Abbildung 6.1: Beispiel einer Regressionsanalyse für drei verschiedene Kurventypen. Die Messwerte (Datenpunkte) wurden generiert und dabei zufällig entlang einer Parabel durch den Nullpunkt verteilt. Außerhalb der Messpunkte gehen die Werte der Approximationsfunktionen weit auseinander.

Im Allgemeinen dient eine Regresionsanalyse dazu, eine Vermutung über den Zusammenhang einer bestimmten Art, beispielsweise einen linearen Zusammenhang, zu überprüfen. Dabei wird die abhängige Variable durch eine oder mehrere unabhängige Variablen bestimmt. Es können nur die Parameter eines vermuteten Funktionstyps bestimmt werden, nicht jedoch die Art des Zusammenhangs selbst. Zur Durchführung werden viele aussagekräftige Messwerte benötigt, auf deren Qualität später noch eingegangen wird. Ein Messwert umfasst zugehörige Werte der unabhängigen Variablen (X-Wert, eventuell auch X_1,X_2,usw.) und den resultierenden Wert der abhängigen Variable (Y-Wert). Die Regressionsanalyse muss für jeden Typ von Task und auf jedem VM-Typ separat durchgeführt werden, da für jede Applikation und jeden VM-Typ unterschiedliche Verhaltensweisen möglich sind.

Ziel der Regressionsanalyse ist es, die Parameter einer vorgegeben Funktion zu bestimmen, von der die Abweichungen der Messpunkte minimal sind. Es wird davon ausgegangen, dass die so konstruierte Funktion auch für weitere Werte der unabhängigen Größe gute Näherungen für die abhängige Größe errechnet [Nie08, S. 114ff.]. Im Allgemeinen versucht man, die approximierte Funktion so einfach wie möglich zu gestalten, also beispielsweise als Gerade. Verwendet man hingegen eine polynomielle Funktion, können die Messpunkte eventuell besser verbunden werden, da die Funktion besser angepasst werden kann. Je spezieller die Funktion jedoch wird, desto mehr geht ihr Vorhersagecharakter verloren. Man spricht in diesem Fall von Überanpassung. Da die Verwendung von Polynomen schnell zur Überanpassung führen kann, wird bei Unkenntnis des Kurventyps in dieser Arbeit ausschließlich mit linearen Funktionen gearbeitet. Abbildung 6.1 visualisiert einige zufällig entlang einer quadratischen Funktion verteilte Datenpunkte. Zudem enthält die Abbildung drei mögliche Approximationskurven verschiedenen Typs. Es ist gut erkennbar, dass die Kurven über die Spannweite der Messpunkte eine brauchbare Approximation liefern können. Für Werte außerhalb der Spannweite (siehe Abbildung 6.1 (rechte Seite)) weichen die Kurven jedoch sehr stark voneinander ab, sodass deren Vorhersagecharakter verloren geht. Generell ist die approximierte Funktion zunächst nur für die Spannweite, also die Werte zwischen dem minimalen und dem maximalen Messpunkt, gültig. Vor allem für Polynome ist diese Einschränkung notwendig. Hat

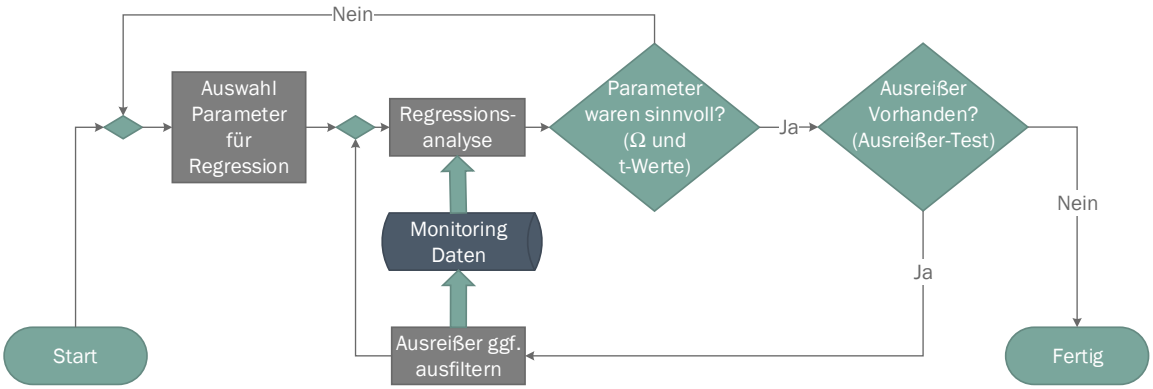


Abbildung 6.2: Schematische Darstellung des Ablaufs einer Regressionsanalyse.

man jedoch Kenntnis über die Art der Abhängigkeit, so kann man anhand des Funktionstyps eventuell auch über die Spannweite hinaus Aussagen treffen. Weiterführende Literatur zum Thema Regressionsanalyse ist in [FKL09, Nie08, Bra13] zu finden. Nachfolgend werden nur für diese Arbeit wichtige Aspekte näher erläutert.

6.2.1 Voraussetzungen

Im Allgemeinen wird davon ausgegangen, dass die Abweichung der Messpunkte von der zu approximierenden Kurve normalverteilt ist (siehe [Nie08, S.39]). Dazu ist es notwendig, dass mindestens 30 unabhängige Messungen vorliegen, damit diese Annahme auch hinreichend gestützt wird. Um einen guten Vorhersagecharakter der Approximationsfunktion sicherzustellen, sollten die verwendeten Messpunkte zudem eine möglichst große Spannweite (Abstand zwischen minimalem und maximalem Messwert) abdecken. Die Gesamtheit der betrachteten Messwerte/Messpunkte wird auch als Stichprobe bezeichnet. Wichtig ist, dass sich Median und Mittelwert der Stichprobe nicht deutlich unterscheiden. Andernfalls muss davon ausgegangen werden, dass es starke Ausreißer in der Stichprobe gibt, die auf Plausibilität überprüft werden müssen. Besonders für sehr kurze Ausführungsdauern im Millisekundenbereich, die bei *Mobilen Cloud-unterstützten Anwendungen* üblicherweise nicht auftreten, existieren häufig endlastige Verteilungen der Messwerte [TTH05]. Hierbei treten vereinzelt starke Ausreißer nach oben auf, die den Mittelwert verzerren. Für solche Verteilungen kann jedoch eventuell trotzdem eine Regressionsanalyse durchgeführt werden. Mögliche Vorgehensweisen dazu sind in Abschnitt 6.2.4 beschrieben.

Die abhängige Variable wird aufgrund von Fremdeinflüssen oft nicht nur von der unabhängigen Variable bestimmt. Man kann Fremdeinflüsse jedoch manchmal nicht spezifizieren und somit nicht ausschließen. Treten also bei verschiedenen Messungen für die gleiche unabhängige Variable deutlich verschiedene Ergebnisse auf, so muss der Grund für das Verhalten näher analysiert werden. Dass Fremdeinflüsse gerade auch bei *Java*-Programmen auftreten können, wird später in Kapitel 6.4 noch näher erläutert. Kommen erklärbare starke Schwankungen nur selten vor, kann man sie oft ausfiltern. Weiterhin ist es auch möglich, aus mehreren Messwerten für die gleiche unabhängige Variable den Mittelwert zu bilden, ohne dass dabei der Vorhersagecharakter verloren geht.

Es wird außerdem in dieser Arbeit eingeschränkt, dass die X-Werte der Messpunkte (unabhängige Variablen) keinem Fehler unterliegen, da diese für die Messung explizit festgelegt werden können. Dadurch vereinfacht sich die Regression.

Sind die Voraussetzungen bezüglich der Qualität der Datenpunkte erfüllt, so kann die eigentliche Bestimmung der Parameter der Regressionsfunktion beginnen. Da die Parameter linear kombiniert werden, spricht man von linearer Regression. Hierbei liegt der Schätzfunktion eine unabhängige Variable zu Grunde. Liegt der Schätzfunktion hingegen mehr als eine unabhängige Variable zu Grunde, so spricht man von multipler Regression. Nachfolgend wird zunächst die einfache Methode erläutert, um ein Verständnis für die Idee der Regressionsanalyse zu schaffen. Anschließend wird die Methode auf mehrere Variablen erweitert. Der schematische Ablauf einer Regressionsanalyse mit Vor- und Nachbereitung ist in Abbildung 6.2 dargestellt.

6.2.2 Durchführung einer einfachen Regression

Eine oft verwendete Schätzmethode ist die Kleinste-Quadrate-Schätzung [Nie08, S.129 ff.]. Für eine lineare Funktion wird diese folgendermaßen angewendet. Zunächst wird die Funktion allgemein aufgestellt:

$$\hat{y} \quad = \quad f(\hat{x}) = m \cdot \hat{x} + n. \tag{6.9}$$

Dabei ist $\hat{y}$ der approximierte Wert und $\hat{x}$ sind die Abszissenwerte der tatsächlichen Messwerte. Für eine Menge von k Messwerten (x_i, y_i) mit $i = 1, \ldots, k$ wird nun die Summe der quadrierten Abweichungen der tatsächlichen Messwerte y von der Funktion $f(\hat{x})$ wie folgt berechnet:

$$V \quad = \quad \sum_{i=1}^{k} (y_i - \hat{y}_i)^2. \tag{6.10}$$

Gesucht ist nun eine Wahl der Parameter m und n, sodass der Gesamtfehler V minimal wird. Dazu wird zunächst Gleichung (6.9) in (6.10) eingesetzt:

$$V(m, n) \quad = \quad \sum_{i=1}^{k} (y_i - m \cdot \hat{x}_i - n)^2. \tag{6.11}$$

Um nun das Minimum bezüglich $V(m, n)$ zu bestimmen, werden weiterhin die partiellen Ableitungen $\frac{\partial V(m,n)}{\partial n}$ und $\frac{\partial V(m,n)}{\partial m}$ gebildet [BSMM12, S.999 f.]. Man erhält:

$$\frac{\partial V(m, n)}{\partial m} \quad = \quad 2 \cdot \sum_{i=1}^{k} (y_i - m \cdot \hat{x}_i - n) \cdot (-\hat{x}_i), \tag{6.12}$$

$$\frac{\partial V(m, n)}{\partial n} \quad = \quad 2 \cdot \sum_{i=1}^{k} (y_i - m \cdot \hat{x}_i - n) \cdot (-1). \tag{6.13}$$

Diese werden nun zur Bestimmung des Minimums Null gesetzt. Es handelt sich hierbei stets um eine Minimum, da gezeigt werden kann, dass die Hesse-Matrix der partiellen Ableitungen

der 2. Ordnung stets positiv definit ist [Tou02, S.47 ff.]. Man erhält:

$$\sum_{i=1}^{k}(\hat{x}_i y_i) = m \cdot \sum_{i=1}^{k}\hat{x}_i^2 + n \cdot \sum_{i=1}^{k}\hat{x}_i, \tag{6.14}$$

$$\sum_{i=1}^{k} y_i = m \cdot \sum_{i=1}^{k}\hat{x}_i + n \cdot k. \tag{6.15}$$

Anschließend wird das gesamte Gleichungssystem durch die Variable k geteilt, was der Ersetzung der Summenformeln durch die entsprechenden Mittelwerte der Datenpaare entspricht:

$$\overline{\hat{x}_i y_i} = \frac{\sum_{i=1}^{k}(\hat{x}_i y_i)}{k}, \quad \overline{\hat{x}_i^2} = \frac{\sum_{i=1}^{k}\hat{x}_i^2}{k}, \quad \overline{\hat{x}_i} = \frac{\sum_{i=1}^{k}\hat{x}_i}{k}, \quad \overline{y_i} = \frac{\sum_{i=1}^{k} y_i}{k}.$$

Man erhält:

$$\overline{\hat{x}_i y_i} = m \cdot \overline{\hat{x}_i^2} + n \cdot \overline{\hat{x}_i}, \tag{6.16}$$

$$\overline{y_i} = m \cdot \overline{\hat{x}_i} + n. \tag{6.17}$$

Aus diesem System lassen sich anschließend beispielsweise durch Anwendung des Einsetzungsverfahrens die Werte für m und n berechnen.

Weitere Verfahren zur Bestimmung nichtlinearer Zusammenhänge beispielsweise für Polynome sind in [Nie08, BSMM12] beschrieben.

6.2.3 Durchführung einer multiplen Regression

Die einfache Regression ist ein Spezialfall der multiplen Regression. Im Gegensatz zur einfachen Regression wird die abhängige Variable $\hat{y}$ aus mehreren unabhängigen Variablen $\hat{x}_1, ..., \hat{x}_q$ $(q \in \mathbb{Z}, q > 1)$ gebildet. Somit ergeben sich für jede Variable $\hat{x}_1, ..., \hat{x}_q$ auch eigene Koeffizienten $m_1, ..., m_q$. Allgemein schreibt man [Nie08, S. 389f.]:

$$\hat{y} = f(\hat{x}_1, \ldots, \hat{x}_q) = n + m_1 \cdot \hat{x}_1 + \cdots + m_q \cdot \hat{x}_q. \tag{6.18}$$

Zur Bestimmung der Koeffizienten bietet sich bei der multiplen Regression die Matrixschreibweise an, um alle k Messpunkte darzustellen.

$$\underbrace{\begin{pmatrix}\hat{y}_1\\ \vdots\\ \hat{y}_k\end{pmatrix}}_{\hat{Y}} = \underbrace{\begin{pmatrix}1 & \hat{x}_{11} & \ldots & \hat{x}_{q1}\\ \vdots & \vdots & & \vdots\\ 1 & \hat{x}_{1k} & \ldots & \hat{x}_{qk}\end{pmatrix}}_{\hat{X}} \cdot \underbrace{\begin{pmatrix}n\\ m_1\\ \vdots\\ m_q\end{pmatrix}}_{m} \tag{6.19}$$

Es muss gelten $k >> q$. Somit ist das Gleichungssystem $\hat{Y} = \hat{X} \cdot m$ überbestimmt. Ziel ist, wie auch bei der einfachen Regression, die Summe der Fehlerquadrate zu minimieren. Das zu lösende Minimierungsproblem lautet demnach:

$$\left\| \hat{X} \cdot m - \hat{Y} \right\|^2 \rightarrow min. \tag{6.20}$$

Für die Bestimmung des Minimums muss zunächst der Gradient dieses Terms bestimmt und Null gesetzt werden. Als Zwischenbemerkung sei erwähnt, dass der Gradient einer Funktion $g(x) = ||f(x)||^2$ für beliebiges $f : \mathbb{R}^n \to \mathbb{R}^n$ durch $\nabla g(x) = 2Df(x)^T f(x)$ gegeben ist [Grü08, S. 7], wobei $Df(x)^T$ die Differentiation von $f(x)^T$ darstellt. Mit diesem Hinweis lässt sich das Minimierungsproblem Null setzen und wie folgt schreiben:

$$\hat{X}^T\hat{Y} \quad = \quad \hat{X}^T\hat{X}m. \tag{6.21}$$

Der Vektor m löst das Minimierungsproblem genau dann, wenn die Normalengleichungen in (6.21) erfüllt sind. Da $\hat{X}$ vollen Spaltenrang besitzt, ist die zweite Ableitung

$$D^2\left(\left\|\hat{X}\cdot m - \hat{Y}\right\|^2\right) \quad = \quad 2\hat{X}^T\hat{X}$$

stets positiv definit. Somit liegt als Lösung stets ein Minimum vor.

Diese Arbeit beschränkt sich auf die multiple Regression mit genau zwei unabhängigen Variablen. Räumlich betrachtet wird also im Gegensatz zur einfachen Regression nicht mehr nach einer Schätzfunktion in der Ebene, sondern nach einer Schätzfunktion im Raum gesucht. Dabei ist diese Funktion bei gänzlich unabhängigen $\hat{x}_1$ und $\hat{x}_2$ als Ebene beschrieben, bei sich beeinflussenden Variablen kann diese Ebene zusätzlich verformt sein. Die Beschaffenheit der Funktion $f(\hat{x}_1, \hat{x}_2)$ hängt also auch davon ab, ob sich die unabhängigen Variablen gegenseitig beeinflussen. Da dieser Fall im Rahmen der Arbeit vorkommt, kann man ihn berücksichtigen, indem man zusätzlich einen Summanden hinzufügt, der die Wechselwirkung der Variablen beschreibt. Bei etwa $\hat{x}_1$ und $\hat{x}_2$ fügt man den Summand $(m_{12} \cdot \hat{x}_1\hat{x}_2)$ hinzu. Für den speziellen Fall mit zwei Variablen wird folgende Schätzfunktion aufgestellt:

$$\hat{y} \quad = \quad f(\hat{x}_1, \hat{x}_2) = n + m_1 \cdot \hat{x}_1 + m_2 \cdot \hat{x}_2 + m_{12} \cdot \hat{x_1}\hat{x_2}. \tag{6.22}$$

Zur Bestimmung der Koeffizienten bietet sich bei der multiplen Regression die Matrixschreibweise an, um alle k Messpunkte darzustellen.

$$\underbrace{\begin{pmatrix}\hat{y}_1\\ \vdots \\ \hat{y}_k\end{pmatrix}}_{\hat{Y}} \quad = \quad \underbrace{\begin{pmatrix}1 & \hat{x}_{11} & \hat{x}_{21} & (\hat{x}_{11}\hat{x}_{21})\\ \vdots & \vdots & \vdots & \vdots \\ 1 & \hat{x}_{1k} & \hat{x}_{2k} & (\hat{x}_{1k}\hat{x}_{2k})\end{pmatrix}}_{\hat{X}} \cdot \underbrace{\begin{pmatrix}n\\ m_1\\ m_2\\ m_{12}\end{pmatrix}}_{m} \tag{6.23}$$

Das zu lösende Minimierungsproblem wird dann durch nachfolgende Normalengleichungen repräsentiert:

$$\underbrace{\begin{pmatrix}1 & \cdots & 1\\ \hat{x}_{11} & \cdots & \hat{x}_{1k}\\ \hat{x}_{21} & \cdots & \hat{x}_{2k}\\ (\hat{x}_{11}\cdot\hat{x}_{21}) & \cdots & (\hat{x}_{1k}\cdot\hat{x}_{2k})\end{pmatrix}\cdot\begin{pmatrix}\hat{y}_1\\ \vdots\\ \hat{y}_k\end{pmatrix}}_{\hat{X}^T\hat{Y}} \quad = \quad \underbrace{\begin{pmatrix}k & \sum_{i=1}^{k}\hat{x}_{1i} & \sum_{i=1}^{k}\hat{x}_{2i} & \sum_{i=1}^{k}(\hat{x}_{1i}\cdot\hat{x}_{2i})\\ \sum_{i=1}^{k}\hat{x}_{1i} & \sum_{i=1}^{k}\hat{x}_{1i}^2 & \sum_{i=1}^{k}\hat{x}_{1i}\hat{x}_{2i} & \sum_{i=1}^{k}\hat{x}_{1i}^2\hat{x}_{2i}\\ \sum_{i=1}^{k}\hat{x}_{2i} & \sum_{i=1}^{k}\hat{x}_{1i}\hat{x}_{2i} & \sum_{i=1}^{k}\hat{x}_{2i}^2 & \sum_{i=1}^{k}\hat{x}_{1i}\hat{x}_{2i}^2\\ \sum_{i=1}^{k}(\hat{x}_{1i}\cdot\hat{x}_{2i}) & \sum_{i=1}^{k}\hat{x}_{1i}^2\hat{x}_{2i} & \sum_{i=1}^{k}\hat{x}_{1i}\hat{x}_{2i}^2 & \sum_{i=1}^{k}(\hat{x}_{1i}\cdot\hat{x}_{2i})^2\end{pmatrix}}_{\hat{X}^T\hat{X}}\cdot\underbrace{\begin{pmatrix}n\\ m_1\\ m_2\\ m_{12}\end{pmatrix}}_{m} \tag{6.24}$$

Das Gleichungssystem ist immer eindeutig bestimmt und der Koeffizientenvektor m kann durch geeignete Verfahren, beispielsweise Gaußelimination, bestimmt werden.

Schließlich kann für jeden Koeffizienten aus m bestimmt werden, wie stark er zur Erklärung der Funktion $\hat{y}$ beiträgt. Dazu wird der t-Wert berechnet [Nie08, S. 398]. Ist dieser sehr klein, so kann die Größe auch aus der Vorhersage eliminiert werden, da sie den vorhergesagten Wert eventuell nur sehr wenig verbessert. Die Regression muss ohne diesen Koeffizienten noch einmal durchgeführt werden (siehe Abbildung 6.2). Durch das Weglassen vereinfacht sich jedoch eventuell die Schätzfunktion und die Summe der quadratischen Fehler wird häufig kleiner.

6.2.4 Fehlerbehandlung

Als Ergebnis der Approximation weichen einzelne Messpunkte von der Approximationsfunktion ab. Dieser Abstand wird Residuum genannt. Besteht tatsächlich ein funktionaler Zusammenhang, sollten die Abweichungen einer Normalverteilung folgen. Ob eine Normalverteilung der Fehler vorliegt, lässt sich beispielsweise mit dem $\mathcal{X}^2$-Test bestimmen [BSMM12, S.848 f.]. Ist dies nicht der Fall, wurde eventuell der falsche Funktionstyp approximiert oder man muss zu dem Schluss kommen, dass kein sinnvoller Zusammenhang zwischen der abhängigen und den unabhängigen Variablen hergestellt werden kann.

Liegt wenigstens näherungsweise eine Normalverteilung vor, können weitere Metriken die Qualität der Schätzung charakterisieren. Oft wird das Bestimmtheitsmaß Ω verwendet [Nie08, S.395 ff.]. Es liegt zwischen 0 und 1 und gibt an, wie groß der prozentuale Anteil der Y-Werte ist, die durch die approximierte Funktion erklärt werden. Eine hoher Wert von Ω ($> 0,9$) ist also anzustreben. Da sich Ω nur auf die verfügbaren Messwerte bezieht, sagt es jedoch nichts über den Vorhersagecharakter der approximierten Funktion aus. Insbesondere für Polynome sollte visuell überprüft werden, ob nicht eine Überanpassung stattgefunden hat. Liefert das Bestimmtheitsmaß keinen guten Wert, so müssen die einzelnen Fehler näher analysiert werden.

Aus statistischer Sicht ist relevant, ob alle Messungen den gleichen Fehlerquellen in gleicher Ausprägung unterliegen, oder ob es bestimmte Messungen gibt, die Fehler beinhalten können, die für andere Messungen nicht relevant sind. Ein Problem der Kleinste-Quadrate-Schätzung ist insbesondere die starke Beeinflussung des Ergebnisses durch sogenannte Ausreißer. Als Ausreißer werden Werte bezeichnet, die gegenüber den anderen Werten eine deutliche Abweichung vom restlichen Verhalten der Messpunkte zeigen. In der Statistik wird ein Wert üblicherweise als Ausreißer bezeichnet, wenn er um mehr als das 2,5 fache der Standardabweichung einer Stichprobe vom Erwartungswert, also von der konstruierten Schätzfunktion, abweicht [GB09, S.57 ff.]. Ausreißer entstehen oft durch spezielle Einflussgrößen (beispielsweise durch den *Java Classloader*, siehe Kapitel 6.4). Vor der Konstruktion der Approximationsfunktion sind die Ausreißer jedoch schwer zu ermitteln, da ein Referenzwert fehlt. Nach der Durchführung einer ersten Regressionsanalyse hingegen können Werte, die mit einem großen Fehler behaftet sind, in begründeten Fällen ausgefiltert werden. Danach wird die Regressionsanalyse erneut durchgeführt. Sind für einen X-Wert verschiedene Y-Werte vorhanden, so muss der Mittelwert der Y-Werte benutzt werden. Bei sehr vielen Y-Werten können zudem zur Mittelwertbildung ausschließlich Werte aus dem Interquartilsabstand ($\pm 25\,\%$ der Anzahl der Werte um den Median) aller dieser Y-Werte benutzt werden. Dadurch wird die Verzerrung des Mittelwerts durch Ausreißer minimiert [GB09, S. 92f.].

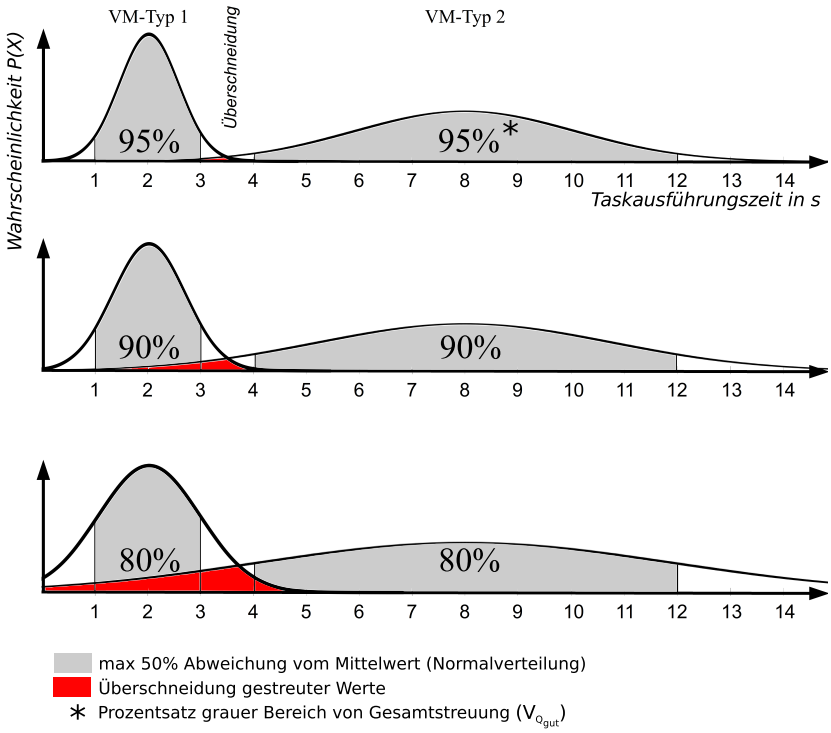


Abbildung 6.3: Darstellung der Streuung vorhergesagter fiktiver Tasklaufzeiten einer Applikation auf je zwei verschiedenen VM-Typen. Der Überschneidungsbereich der Streuung bei einem Performanceunterschied der VMs von Faktor 4 (zu erkennen an den Mittelwerten der Tasklaufzeiten) für verschiedene $V_{Q_{gut}}$ ist jeweils rot markiert.

6.2.5 Vorhersagetauglichkeit

Fraglich bleibt noch, ab wann eine Approximation unbrauchbar ist und nicht zur Vorhersage verwendet werden kann. Eine Antwort hierauf kann gegeben werden, wenn man beachtet, dass nur diskrete Werte für die Geschwindigkeit der VM Ressourcen, also auch für die Laufzeit der Tasks, vorliegen. Ziel ist, dass sich die vorhergesagten Laufzeitverteilungen für verschiedene VM-Typen unter Beachtung des Vorhersagefehlers möglichst wenig überschneiden. Eine Überschneidung könnte bedeuten, dass für die gleiche Performance auch eine andere, womöglich günstigere Ressource, hätte gewählt werden können. Von Bedeutung für eine belastbare Laufzeitvorhersage ist also je nach Güte der Streuung um den Mittelwert auch die Wahl der verfügbaren VM-Typen für eine *Mobile Cloud-unterstützte Anwendung*.

Zur Bewertung der Schätzung der Ausführungszeit wird das Maß Güte verwendet, das die Abweichung vom Schätzwert beschreibt. Die Güte Q berechnet sich wie folgt:

$$Q = \frac{Schätzwert}{Messwert}. \tag{6.25}$$

Zudem wird festgelegt, dass eine spezielle Güte Q_{gut} genau dann vorliegt, wenn der Messwert nicht mehr als $\pm 50\,\%$ vom Schätzwert abweicht.

Abbildung 6.3 verdeutlicht die ideal normalverteilte Laufzeitverteilung von unendlich vielen fiktiven Messwerten einer Applikation auf je zwei verschiedenen VM-Typen für drei Konfi-

gurationen. Dabei unterscheiden sich die Mittelwerte der Tasklaufzeiten in der Abbildung um den Faktor 4. Konfiguriert man etwa aus dem Portfolio des Cloud-Anbieters *Amazon EC2* die VM-Typen `m1.small`, `c1.medium` und `c1.xlarge`, so unterscheiden sich diese in ihrem Performance-Rating um den Faktor 4–5. Die drei Diagramme visualisieren jeweils eine Streuung zwischen $\pm 50\,\%$ um den Mittelwert (Q_{gut}). Dabei liegen 95%, 90% oder 80% der Messwerte innerhalb dieses Streuungsbereichs. Dazu wird das Verhältnis $V_{Q_{gut}}$ definiert:

$$V_{Q_{gut}} = \frac{|Messwerte\ mit\ Q_{gut}|}{|Messwerte|}. \tag{6.26}$$

Es ergibt sich jeweils eine leichte Überschneidung der tatsächlich auftretenden Werte zwischen den Streuungsbereichen der beiden VM-Typen (zwischen den X-Werten 3–4 und darüber hinaus). Für $V_{Q_{gut}} = 0,95$ ist die Häufigkeit des Auftretens von Messungen in dieser Überschneidung sehr gering. Je geringer die Überschneidung ist, desto weniger wahrscheinlich ist es, dass die andere VM Ergebnisse im gleichen Bereich liefert. Die Wahrscheinlichkeit, den falschen VM-Typ gewählt zu haben, ist geringer. Diese Wahrscheinlichkeit nimmt jedoch in den beiden anderen Varianten zu. Für diese Arbeit wird definiert, dass es gerade noch vertretbar ist, wenn sich bei einem Performanceunterschied der gewählten Ressourcen von etwa Faktor 4 ca. 80% der tatsächlich auftretenden Messwerte innerhalb des $\pm 50\,\%$-Bereichs um den Mittelwert befinden. Es wird deshalb das Gütekriterium $K_{Q_{gut}}$ definiert, welches besagt, dass mindestens 80% der Messwerte eine Güte Q_{gut} aufweisen müssen. Andernfalls gilt $K_{Q_{gut}}$ als nicht eingehalten.

$$K_{Q_{gut}} \quad \Leftrightarrow \quad V_{Q_{gut}} \geq 0,8. \tag{6.27}$$

6.2.6 Toolunterstützung

Um die Performance einer Applikation beziehungsweise einzelner Tasks mittels Regressionsanalyse in einer Schätzfunktion zu repräsentieren, bedarf es Werkzeugunterstützung in zwei Bereichen. Erstens muss ein Monitoring implementiert werden, welches alle nötigen Parameter der Taskausführung aufnehmen kann. Auf Basis dieser Messwerte wird anschließend die Regressionsanalyse durchgeführt. Zweitens wird für die Regressionsanalyse selbst auch Programmunterstützung benötigt, da diese für eine große Anzahl von Messwerten nicht mehr händisch durchgeführt werden kann.

Die zu entwickelnde Middleware auf Serverseite muss demnach für jeden Task p_k die Größe der Eingabedaten e_k und die jeweilige Ausführungszeit auf Serverseite $\Delta t_{work\,k}$ erfassen. Dies kann schwierig sein, da die tatsächliche Eingabegröße, die als unabhängige Größe für die Schätzfunktion identifiziert wurde, für jeden Typ p_k von Task verschieden sein kann. Für Bildverarbeitungstasks ist es eventuell die Pixelanzahl, für Tonverarbeitung aber eventuell die Samplerate, etc. Somit muss die Middleware ein konfigurierbares Logging bereitstellen.

Für die hier verwendete Regressionsanalyse über lineare Regression gibt es viele verschiedene Software-Werkzeuge. Zunächst ist es wichtig, die erhaltenen Messwerte zu visualisieren.

Dazu kann beispielsweise *MATLAB*[1] oder das freie Werkzeug *GNU R*[2] verwendet werden. *GNU R* ist auch in der Lage, verschiedene Kurventypen an die Messwerte anzunähern, und verwendet dazu die Methode der Kleinste-Quadrate-Schätzung [SH07, S. 92]. Zu beachten ist jeweils, dass die Schätzfunktion auch beschränkt sein kann, beispielsweise dürfen eventuell keine negativen Werte entstehen. Dies ist bei der Spezifikation des Kurventyps anzugeben.

Abschließend ist zu prüfen, ob das geforderte Gütekriterium $K_{Q_{gut}}$ für die approximierte Funktion bezüglich der Messwerte erreicht wird. Da es sich bei diesem Qualitätstest um keinen Standardtest handelt, muss die Qualität mit Hilfe eines Tabellenkalkulationsprogramms auf Basis der Messwerte und der Schätzfunktion selbst berechnet werden.

6.2.7 Zusammenfassung Regressionsanalyse

Für eine Regressionsanalyse sollte zunächst analysiert werden, von welchen Variablen die Laufzeit der Tasks der *Mobilen Cloud-unterstützten Anwendung* abhängt. Bezüglich der Qualität der Messdaten muss eine große Menge und eine möglichst große Spannweite an Messdaten vorliegen. Erklärbare Ausreißer sollten ausgefiltert werden. Für jeden VM-Typ muss eine separate Regressionsanalyse durchgeführt werden. Nach der Regressionsanalyse muss das Ergebnis auf jedem VM-Typ das Gütekriterium $K_{Q_{gut}}$ erfüllen. Die Analyse kann mit Hilfe verschiedener verfügbarer Tools durchgeführt werden, die zudem auch wichtige Parameter der Regression automatisch bestimmen. Im nachfolgenden Abschnitt wird untersucht, welche Performanceschwankungen durch die Middleware selbst und durch Applikationen zu erwarten sind.

6.3 Cloud-Performance Modellierung und Bewertung

Cloud-Ressourcen eines IaaS-Anbieters sind virtuelle Maschinen und diese unterscheiden sich, wie auch dedizierte Server, im Allgemeinen in: (siehe Anhang B)

- der Geschwindigkeit der einzelnen CPU-Kerne,
- der Anzahl der einzelnen CPU-Kerne,
- der Größe des verfügbaren Hauptspeichers,
- und in den dafür verlangten Kosten pro Zeiteinheit.

[1] *„MATLAB ist eine höhere Programmiersprache und interaktive Umgebung für numerische Berechnungen, Visualisierung und Programmierung. MATLAB dient zur Datenanalyse, Algorithmen-Entwicklung und zur Erstellung von Modellen und Anwendungen. Mit der Programmiersprache, den Tools und den integrierten mathematischen Funktionen können Sie verschiedene Ansätze ausprobieren und schneller zu einer Lösung gelangen als mit Tabellenkalkulationen oder herkömmlichen Programmiersprachen wie C/C++ oder Java.“* (Onlineressource `http://www.mathworks.de/products/MATLAB/` (abgerufen am 12.9.2013))

[2] *„R ist eine freie Programmiersprache für statistisches Rechnen und statistische Grafiken. Sie ist in Anlehnung an die Programmiersprache S entstanden und weitgehend mit dieser kompatibel. Außerdem orientierten sich die Entwickler an der Programmiersprache Scheme. R ist Teil des GNU-Projekts und auf vielen Plattformen verfügbar. [...] R wurde 1992 von Ross Ihaka und Robert Gentleman an der Universität Auckland entwickelt. [...] 1993 wurde die Software erstmals öffentlich vorgestellt, seit Juni 1995 steht R unter der GNU General Public License. “* (Onlineressource `http://de.wikipedia.org/wiki/R_(Programmiersprache)`(abgerufen am 30.10.2013))

Auf diesen virtuellen Maschinen können verschiedene Betriebssysteme gestartet werden. Nicht von jedem IaaS-Anbieter wird das Starten beliebiger Betriebssysteme unterstützt. Weiterhin ist das Benutzen verschiedener Betriebssysteme verschieden teuer. Beispielsweise wird durch *Amazon EC2* für Windows-Instanzen etwas mehr berechnet als für Linux-Instanzen. Für die für *Mobile Cloud-unterstützte Anwendungen* benötigte *Java*-Umgebung sind jedoch schlicht konfigurierte Linux-Instanzen ausreichend, was später in Kapitel 7.1 zur Implementierung noch näher erläutert wird. In den nachfolgenden Abschnitten werden die besonderen Eigenschaften bei der Performancemodellierung virtualisierter Maschinen untersucht.

6.3.1 Performance-Rating

Um die Berechnungsperformance ihrer Angebote zu kategorisieren, verwenden IaaS-Anbieter oft ein Performance-Rating. Dabei wird häufig nur die Anzahl der CPU-Kerne und deren Geschwindigkeitsbewertung (oft als Rating bezeichnet) multipliziert und um einen Faktor ergänzt. Anhand dieser Angabe werden im Allgemeinen die Kosten für die Abrechnung der Nutzungsdauer berechnet. *Amazon EC2* etwa stellt ein ECU-Rating[3] zur Verfügung (siehe Anhang B). Dieses Rating kann jedoch trügerisch sein und reflektiert nicht notwendigerweise die tatsächliche Performanceeinordnung verschiedener VM-Typen zueinander. Aus Amdahls Gesetz lässt sich ableiten, dass der parallele Speedup *de facto* niemals 100 % betragen kann [RR00, S.313], da ein Programm in der Praxis immer auch eine geringen Anteil sequentiell auszuführender Instruktionen umfasst, die sich nicht parallelisieren lassen. Ein ECU-Rating von 2 kann auf einem Dual-Core System gebildet werden aus $2\,ECU = 2 \cdot 1\,GHz$ und auf einem Single-Core System aus $2\,ECU = 1 \cdot 2\,GHz$. Nur ein paralleles Programm, welches einen parallelen Speedup von 100% aufweist, würde auf beiden Konfigurationen tatsächlich die gleiche Performance erzielen. Das Performance-Rating ist demnach nur bedingt geeignet, um die Performance einer Ressource zu beschreiben.

Eine andere Möglichkeit, die Performance zu beschreiben, ist die Einteilung der verfügbaren Ressourcen in Gruppen, die jeweils die gleiche Einzelprozessor-Performance aufweisen. Da viele berechnungsintensive Programme recht proportional mit der CPU-Kernfrequenz des Prozessors skalieren, ist die Performance der Gruppeneinteilung wenigstens für Einzelprozessor-Anwendungen vergleichbar. Einige von *Amazon EC2* angebotenen VM-Typen sind beispielsweise in den folgenden Konfigurationen verfügbar:

2 ECU pro CPU-Kern (`m1.small,m1.large,m1.xlarge`)

2,5 ECU pro CPU-Kern (`c1.medium,c1.xlarge`)

3,25 ECU pro CPU-Kern (`m2.xlarge,m2.2xlarge,m2.4xlarge`)

[3] *„Amazon EC2 setzt verschiedene Maßnahmen ein, um jede Instance mit einer gleichbleibenden und vorhersehbaren Menge an CPU-Kapazität zu versorgen. Um den Entwicklern den Vergleich zwischen den CPU-Kapazitäten der verschiedenen Instance-Typen zu erleichtern, haben wir eine Amazon EC2 Compute Unit definiert. Die einer bestimmten Instance zugewiesene CPU-Menge wird in diesen EC2 Compute Units ausgedrückt. Um die Konsistenz und Vorhersehbarkeit der Leistung einer EC2 Compute Unit zu verwalten, werden verschiedene Benchmarks und Tests eingesetzt. Eine EC2 Compute Unit bietet die entsprechende CPU-Kapazität eines 1,0- bis 1,2 GHz-Opteron oder -Xeon Prozessors von 2007. Dies entspricht außerdem einem 1,7-GHz-Xeon-Prozessor von Anfang 2006, auf den in unserer Originaldokumentation verwiesen wird. Mit der Zeit werden wir möglicherweise weitere Größenwerte zur Definition einer EC2 Compute Unit hinzufügen oder ersetzen, sofern Kennzahlen gefunden werden, die Ihnen die Rechenkapazität besser verdeutlichen.“* (Onlineressource `http://aws.amazon.com/de/ec2/faqs/` (abgerufen am 6.9.2013))

Innerhalb der Gruppen gibt es dann Ressourcen mit verschieden vielen CPU-Kernen. Man kann also je nach parallelem Speedup der Applikation eine weitere Beschleunigung erreichen.

Zusammenfassend wird festgehalten, dass das Performance-Rating im Allgemeinen nicht verwendet werden kann, um die Applikationsperformance auf einer Ressource vorherzusagen. Somit muss für jeden VM-Typ die Applikationsperformance für verschiedene Eingaben separat bestimmt werden. Nur für Einzelprozessorapplikationen können eventuell Vereinfachungen ausgenutzt werden. Bisher wurde stets davon ausgegangen, dass die Verarbeitungsgeschwindigkeit für identisch konfigurierte VM-Instanzen, wie bei physikalischen Servern, annähernd gleich hoch ist. Ob diese Annahme tatsächlich auch für virtualisierte Server getroffen werden kann, wird in Kapitel 6.3.3 untersucht.

6.3.2 Speicheranforderungen

Die Menge des von einer Applikation verwendeten Hauptspeichers hängt unter anderem von dem verwendeten Algorithmus und der Beschaffenheit der Eingabedaten ab und lässt sich im Allgemeinen schwierig bestimmen [BFGY08]. Dies gilt insbesondere für *Java*-Anwendungen, die keine explizite Speicherverwaltung ermöglichen, da das Laufzeitsystem diese übernimmt. Hier haben auch Einstellparameter des Laufzeitsystems Einfluss auf den Speicherbedarf. Dennoch ist der Speicherbedarf eine wichtige Größe. Bei der Auswahl eines VM-Typs mit zu wenig Speicher kann eine Applikation nicht mehr ausgeführt werden und bricht ab. Im Gegensatz dazu beeinflusst die Auswahl von zu viel Speicher die Ausführungszeit im Allgemeinen kaum positiv. Bezüglich der Speicherkonfiguration einer Ressource muss somit nur darauf geachtet werden, dass ausreichend Hauptspeicher vorhanden ist.

Somit muss die Maximalmenge von benötigtem Hauptspeicher für jede Applikation *a priori* ermittelt werden. Aufgrund dieser Schätzung können anschließend nur VM-Typen für die Nutzung konfiguriert werden, die die gewünschte Menge an Speicher auch bereitstellen. Sind also beispielsweise mindestens 2 GB Hauptspeicher gefordert, so können die Instanztypen `m1.small, c1.medium` und `t1.micro` vom *Amazon EC2* nicht konfiguriert werden, da diese weniger Hauptspeicher zur Verfügung haben (siehe Anhang Tabelle B.3).

Java bietet zur Erfassung des momentan verwendeten Arbeitsspeichers sowie des verfügbaren Speichers eingebaute Methoden. Die Informationen lassen sich im Programm selbst durch entsprechende *Java*-API-Aufrufe beschaffen. Zudem gibt es grafische Werkzeuge, die Laufzeitinformationen von *Java*-Programmen darstellen können. Hierbei sind die Werkzeuge `VisualVM`[4] und `JConsole`[5] hervorzuheben, da sie Bestandteil der *Java*-Installation sind. Durch die Middleware ist sichergestellt, dass auf jeder VM nur eine Serverinstanz und demzufolge auch nur eine JVM ausgeführt wird. Diese erhält den überwiegenden Teil des verfügbaren Arbeitsspeichers zugewiesen.

6.3.3 Volatilität der Performance virtueller Maschinen

An dieser Stelle wird untersucht, welche Auswirkungen durch den virtualisierten Softwarestack auf die Schwankung der Performance von VMs zu erwarten sind. Der für die Midd-

[4] `VisualVM` ist Bestandteil der *Oracle Java*-Installation. Das Programm bietet Profiling und Monitoring für Entwicklungs- und Produktionsumgebungen an. Das Programm ist Open Source und kann durch Plugins erweitert werden. (siehe [Ull12])

[5] `JConsole` ist Teil des *Java Development Kits* (JDK). Es bietet die Möglichkeit, Statusinformationen von lokal und entfernt ausgeführten Programmen anzuzeigen. (siehe [Ull12])

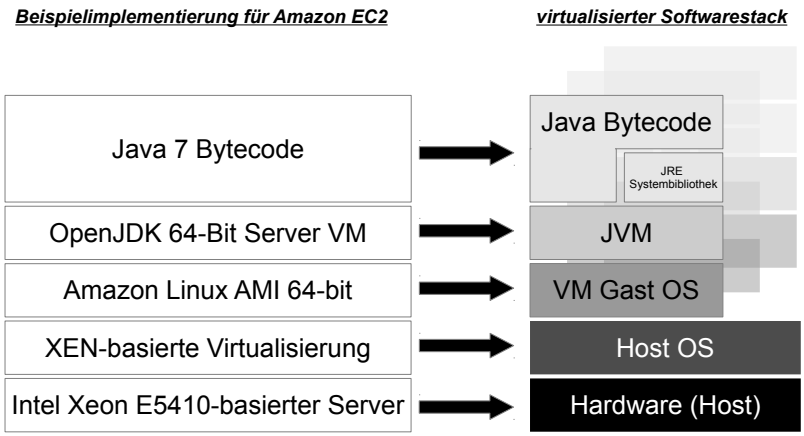


Abbildung 6.4: Softwarestack für Java-Programme, die in einer virtuellen Maschine ausgeführt werden. Zusätzlich ist eine konkrete Instantiierung des Stacks für den Cloud-Anbieter *Amazon EC2* und die in dieser Arbeit verwendete *Java*-Implementierung *OpenJDK* angegeben.

leware relevante Softwarestack ist in Abbildung 6.4 visualisiert. Dabei wird Virtualisierung auf mehreren Schichten eingesetzt. Die Cloud-Ressource selbst ist eine virtuelle Maschine, welche als Gast auf einem physikalischen Host neben weiteren virtuellen Maschinen residiert. Der Applikationscode wird wiederum nicht direkt, sondern von einer JVM ausgeführt. Dazu liegt *Java*-Code als Bytecode vor. Beide Formen der Virtualisierung haben unterschiedlichen Einfluss auf die Performance von Anwendungen, die auf diesem Softwarestack implementiert werden. Um den Einfluss der Virtualisierung auf die Konstanz der Applikationsperformance auf dem verwendeten Softwarestack zu bestimmen, wird nachfolgend anhand von Literatur und eigenen Messungen eine Analyse durchgeführt.

IaaS-Ressourcen

Im Vergleich zu einem dedizierten, nichtvirtualisierten Server ist es üblich, dass mehrere VMs parallel zueinander auf einem physikalischen Host ausgeführt werden. Dies kann im Vergleich zu nichtvirtualisierten Servern dazu führen, dass die VM-Performance nicht zu jedem Zeitpunkt konstant hoch ist. Als Grund für eine solche Schwankung kann beispielsweise angeführt werden, dass sich alle auf dem gleichen Host ausgeführten VMs die verfügbare Hauptspeicherbandbreite teilen. Bei gleichzeitiger Nutzung des Hauptspeichers durch mehrere VMs kann es zu Engpässen kommen, die die Ausführungsgeschwindigkeit einer Applikation beeinflussen können. Weiterhin ist es möglich, dass man bei mehrfachen Anforderungen eines gewissen VM-Types vom IaaS-Anbieter jeweils Maschinen mit verschiedener darunter liegender Hardware (beispielsweise Prozessortypen der Hersteller *Intel* oder *AMD*, siehe Tabelle 6.2) erhält. Dies kann die Applikationsperformance ebenfalls beeinflussen, da sich die Architekturen in Speicher/Cache-Hierarchie oder dem Aufbau der CPU unterscheiden (beispielsweise Anzahl der ALU/FPU Einheiten). Derlei Geschwindigkeitsvolatilität lässt sich nicht verhindern und sollte im Applikationsentwurf berücksichtigt werden. Eine umfangreiche Analyse der Ausführungsgeschwindigkeit von Cloud-Ressourcen ist in einschlägiger Literatur zu finden. In [LML+11, LYKZ10, IOY+11, IYE11] werden einige Anbieter von IaaS-Diensten verglichen.

Tabelle 6.2: Auswertung von Benchmarks auf Cloud-Ressourcen

Mögliche CPU-Typen von VM-Instanzen und Häufigkeitsangaben

	EC2 Small	*EC2 Medium*	*EC2 Large*
CPU Typ			
AMD Opteron 2374 HE	-	-	0,54%
Intel Xeon E5430	67,10%	-	99,46%
Intel Xeon E5410	-	87,50%	-
Intel Xeon E5345	-	12,50%	-
AMD Opteron 2218 HE	32,90%	-	-

Schwankungsbreite von Benchmarkläufen auf VM-Typen

Anzahl Messwerte	462	72	186
Ø Variationskoeffizient	7,70%	4,00%	11,60%

Quelle:[LML+11]

In [LML+11] werden Benchmarks mehrfach auf verschiedenen Ressourcen von *Amazon EC2* durchgeführt und dabei die Schwankungen gemessen. Dabei wurden 11 Programme aus verschiedensten Bereichen als Benchmarks benutzt. Tabelle 6.2 fasst die Ergebnisse der Messungen bezüglich Schwankungsbreite für die drei benutzten *Amazon EC2*-Instanztypen zusammen. Die exakten Instanztypen wurden nicht genannt. Verglichen mit den Eigenschaften der verfügbaren Instanztypen (siehe Anhang Tabelle B.3), entsprechen sie jedoch wahrscheinlich den Typen `m1.small`, `m1.medium` und `m1.large`. Die absolute Performance der Instanztypen ist hier auch nicht von besonderer Bedeutung, da zunächst nur ein Eindruck bezüglich der Stabilität der Performance gewonnen werden soll. Der Variationskoeffizient liegt bei maximal 11,6 %. Das bedeutet bei einer angenommenen Normalverteilung der Messergebnisse, dass ca. 68,3 % der Messergebnisse um maximal 11,6 % vom Mittelwert abweichen. In [LML+11] wurden keine Vergleichsergebnisse für nichtvirtualisierte Server angegeben. Absolut gesehen ist die Abweichung für exakt gleichartige Benchmarks zwar hoch, jedoch nicht unbrauchbar hoch. Daraus kann abgeleitet werden, dass ein Fehler bei der Vorhersage der Performance einer Applikation zu einem geringen Teil auch durch die Schwankung der VM-Performance verursacht werden kann. Es kann jedoch davon ausgegangen werden, dass sich diese im unteren zweistelligen Bereich bewegt.

In [IYE11] wurde eine Langzeitstudie über ein Jahr auf der *Amazon EC2* und *Google AppEngine* Infrastruktur durchgeführt. Hauptsächlich wurde die Schwankung der Performance für Datenbankanfragen und Speicherdienste überwacht. Als Ergebnis wurde festgestellt, dass es Perioden stabiler Performance, aber auch monatliche und tägliche deutliche Schwankungen gibt. In [IOY+11] werden verschiedene Cloud-Anbieter bezüglich der Performance ihrer Ressourcen miteinander verglichen. Das liefert nur bedingt Aussagen über die Variabilität der Performance, zeigt aber, dass auch andere Cloud-Anbieter ähnlich performante Angebote wie *Amazon EC2* zur Verfügung stellen. Zusätzlich wurde explizit die Performanceschwankung einer Instanz des Typs `m1.xlarge` untersucht. Genaue Zahlen sind nicht beschrieben, jedoch lässt sich aus dem abgebildeten Diagramm erkennen, dass die Gesamtschwankungsbreite bei ca. ± 20 % liegt. Dies ist im Gegensatz zu den anderen getesteten IaaS-Anbietern *GoGrid*, *Elastic Hosts* und *Mosso* ein sehr geringer Wert.

In [LYKZ10] werden Metriken vorgestellt, die dazu geeignet sind, die Performance verschie-

dener Angebote von Cloud-Ressourcen miteinander zu vergleichen. Obwohl versucht wurde zu anonymisieren, konnten die Angebote von *Amazon EC2* identifiziert werden. Auch die Schwankungsbreite wurde angegeben. Es wurde außerdem auch zwischen CPU-, Speicher- und Disk-I/O-intensiven Applikationen unterschieden. Dabei wurde deutlich, dass die maximale Schwankungsbreite bei CPU-intensiven Applikationen gering ist (einstelliger Prozentbereich), bei Speicher-intensiven Applikationen höher ist (unterer zweistelliger Prozentbereich) und bei Disk-I/O-intensiven Applikationen sehr hoch ist (bis in den dreistelligen Prozentbereich). Die untersuchten Anbieter *Amazon AWS*, *Microsoft Azure*, *Google AppEngine* und *Rackspace CloudServers* lieferten bezüglich Schwankungsbreite ähnliche Ergebnisse, einzig bei Disk-I/O-intensiven Applikationen schnitt *Amazon AWS* besonders schlecht ab.

Im Gesamteindruck liefern die Instanztypen von *Amazon EC2* die stabilste Performance für CPU-intensive Applikationen. Dennoch muss davon ausgegangen werden, dass die Performance von berechnungsintensiven Applikationen im unteren zweistelligen Prozentbereich um einen Mittelwert schwankt. Dies liegt jedoch weit unterhalb der geforderten maximalen Schwankung von ca. 50% für mindestens 80% der Messwerte und ist somit tolerierbar. Da für speicherintensive und I/O-intensive Applikationen noch weit höhere Schwankungen zu erwarten sind, kann für diese Anwendungen keine sinnvolle Prognose der Verarbeitungsgeschwindigkeit erstellt werden. Auch für berechnungsintensive Applikationen sollte die Prognose stets angepasst werden, da sich die Host-Hardware ändern kann oder es zu dauerhaften Veränderungen durch beispielsweise geändertes Scheduling von VM-Instanzen im Rechenzentrum kommen kann.

Java Virtual Machine – JVM

Bei *Java* handelt es sich um eine interpretierte Sprache, bei der der *Java*-Quellcode in einem Vorkompilierungsschritt zunächst in einen Bytecode übersetzt wird, der dann durch eine JVM ausgeführt wird. Für *Java* gilt zunächst die Vermutung, dass verschiedene JVM-Implementierungen nicht die gleiche Applikationsperformance liefern. Zudem haben weitere *Java*-spezifische Eigenschaften wie *Garbage Collection* Einfluss auf die Performance. Dies hängt unter anderem von der verwendeten Konfiguration der JVM ab. Die *Java*-Sprachspezifikation [GJS+13], wie auch die Spezifikation des Bytecode und der JVM-API [LYBB13], sind öffentlich verfügbar. Es ist jedem gestattet, eine kompatible JVM[6] zu entwickeln, die Bytecode ausführen kann. Somit gibt es am Markt verschiedene JVM-Implementierungen für unterschiedliche Zwecke. Ein ausführlicher Vergleich verschiedener JVM-Implementierungen ist in [Fri12] zu finden.

Aufgrund der sich schnell entwickelnden JVMs verschiedener Hersteller ist das Vergleichen verschiedener JVMs bezüglich der Performance nur schlecht möglich. In der nächsten Version der JVM könnte sich die Geschwindigkeit bereits schon verändert haben. Seit 2011 bildet *Java 7* das aktuelle Ende der Entwicklung des *Java*-Softwaresystems. Die Veröffentlichung der nächsten *Java*-Entwicklungsstufe, *Java 8*, ist für 2014 geplant. Zudem findet im Moment eine Bereinigung des Marktes statt. Einige Hersteller haben die Entwicklung eigener JVMs aufgegeben und beteiligen sich stattdessen im *OpenJDK*-Projekt[7] an der Entwicklung

[6] JVM ist dabei ein geschütztes Label, welches von *Oracle* geprüft wird, bevor eine Virtuelle Maschine für Java als JVM bezeichnet werden darf. Nicht zertifizierte Implementierungen werden als Research Virtual Machine (RVM) bezeichnet. (siehe [Fri12])

[7] Das *OpenJDK* ist ein Projekt, welches vom damaligen Java-Eigentümer *Sun Microsystems* 2006 geschaffen wurde, um den Quellcode der Sun-Implemetierung von Java einer breiten Gemeinschaft zu übergeben und fortan durch diese weiterzuentwickeln. Nachdem *Sun Microsystems* durch *Oracle* übernommen wurde,

Tabelle 6.3: Performancevergleich verschiedener JVM (64-bit)

Benchmark	Leistungs-unterschied	verglichene JVMs	System
SciMark	3,30%[1]	Oracle HotSpot(TM) 1.7.0_17 / OpenJDK 1.7.0_6	2x Intel Xeon E5-2643 @3.3GHz (Linux 3.2.0 64-bit)
DaCapo	0,70%	Oracle HotSpot(TM) 1.7.0_25 / OpenJDK 1.7.0_25	Dell Latitude 6530 (siehe Anhang A)
SciMark	0,20%	Oracle HotSpot(TM) 1.7.0_25 / OpenJDK 1.7.0_25	Dell Latitude 6530 (siehe Anhang A)

1 = Quelle:`http://dior.ics.muni.cz/~makub/java/speed.html` (abgerufen am 02.09.2013)

einer gemeinsamen Open Source JVM. Das *OpenJDK* umfasst seinerseits wiederum den von *Oracle* freigegeben Quellcode der *Oracle HotSpot* JVM mit einigen Einschränkungen. Da für GNU/Linux-Derivate nur freie Software verwendet werden darf, wurde das *IcedTea*-Projekt gegründet, welches als Ziel hat, die Einschränkungen der *OpenJDK*-Quellen durch freie Software zu komplettieren. Dadurch steht nun für nahezu alle Linux-Systeme eine *OpenJDK Java*-Laufzeitumgebung unter Verwendung der *IcedTea* JVM zur Verfügung.

Somit ist für die beiden sehr häufig verwendeten JVM-Implementierungen *Oracle HotSpot* JVM und die *OpenJDK IcedTea*-Implementierung kaum ein Performanceunterschied zu erwarten. Nachfolgend werden nur diese Implementierungen für *Java SE 7* verglichen. Für Performancevergleiche sind einschlägige Benchmark-Anwendungen für *Java* verfügbar. Oft verwendet wird beispielsweise der *DaCapo* Benchmark [BGH+06]. Darin werden eine Reihe verschiedenster *Java*-Anwendungen aus unterschiedlichen Bereichen verwendet. Ein weiterer Benchmark ist der *SciMark 2.0* Benchmark[8], welcher die Geschwindigkeit numerischer Berechnungen als Einzelprozessoranwendung bestimmt. Tabelle 6.3 fasst eigene und fremde Ergebnisse einiger Benchmarkläufe zusammen.

Aus den Benchmarkergebnissen ist abzulesen, dass sich die Verarbeitungsgeschwindigkeiten der beiden betrachteten JVM Implementierungen nur im Rahmen der Abweichung durch allgemeine Messtoleranz unterscheiden und kein deutlicher Unterschied zu erkennen ist. Dies bedeutet, dass die Wahl der JVM keinen deutlichen Einfluss auf die Schwankung der Verarbeitungsgeschwindigkeit hat. Obwohl im Folgenden aufgrund der breiteren und einfacheren Unterstützung der *OpenJDK* JVM ausschließlich diese verwendet wird, sollten alle weiteren Messergebnisse mit sehr geringen Abweichungen auch für die *Oracle* JVM gültig sein. Da im Moment kaum weitere relevante *Java*-Implementierungen neben der *Oracle*- und der *OpenJDK*-Implementierung existieren, werden performancerelevante Einflüsse aufgrund verschiedener JVM-Implementierungen nicht weiter betrachtet. Weitere *Java*-spezifische performancerelevante Eigenschaften, wie *Garbage Collection* oder Just-in-Time Compiler, werden später in Kapitel 6.4 anhand der *OpenJDK*-Implementierung genauer untersucht.

wurde dieser Prozess fortgesetzt, sodass *Oracle* auch heute noch großen Anteil an diesem Open Source-Projekt hat. Heute stellt das *OpenJDK* die Referenzimplementierung von Java dar. (siehe [Fri12])

[8] Verfügbar unter Onlineressource `http://math.nist.gov/scimark2` (Abgerufen am 20.3.2014)

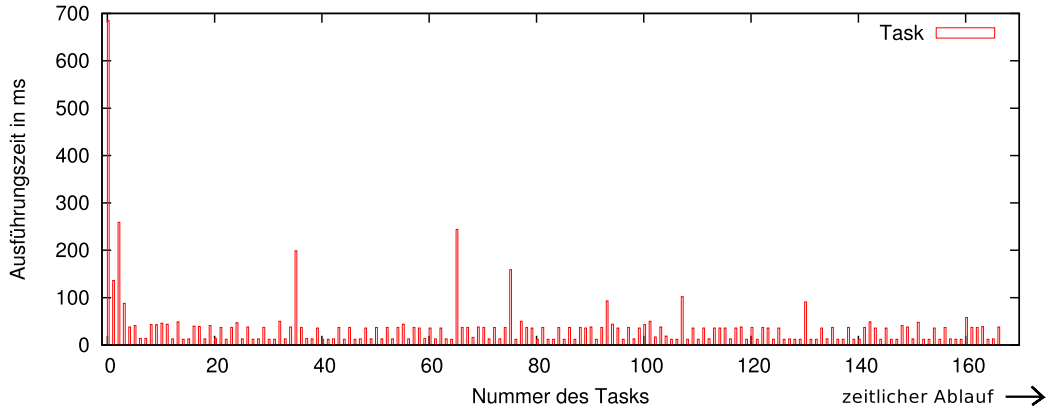


Abbildung 6.5: Ausführungszeiten einer Sequenz von gleichen Tasks innerhalb einer JVM-Instanz auf einem nichtvirtualisierten System. Einzelne Tasklaufzeiten weichen von der Mehrheit deutlich nach oben ab.

6.4 Java-spezifische Einflussparameter auf die Ausführungsperformance

Die Ausführungszeit eines *Java*-Programms wird durch verschiedene *Java*-spezifische Eigenarten des *Java*-Laufzeitsystems beeinflusst. Drei bedeutsame Eigenarten werden nachfolgend genauer betrachtet, der *Classloader* zum Laden der Bytecodedateien, der *Just-In-Time Compiler* zur laufzeitmäßigen Codeoptimierung und der *Garbage Collector* zur Freigabe nicht benötigten Arbeitsspeichers. Diese Einflüsse können vom Programmierer oft nur bedingt beeinflusst werden. Sie führen jedoch in Bezug auf die Regressionsanalyse und die Bestimmung der Laufzeit eines *Java*-Programms zu Schwankungen und Ausreißern in den Messwerten. Abbildung 6.5 zeigt dazu die Laufzeiten einiger gleichartiger sequentiell ausgeführter Tasks innerhalb der gleichen JVM, die die gleiche oder nur eine leicht verschiedene Eingabe verarbeitet haben. Es wurden Tasks der Kantenerkennung genutzt, die in Kapitel 8.3 vorgestellt wird. Somit ist p_k konstant und e_k schwankt nur sehr wenig. Als Ausführungsplattform wurde hier der Dell Latitude Laptop genutzt (siehe Anhang A). Deutlich sichtbar ist, dass einige Laufzeiten deutlich von der Mehrheit abweichen. Diese Effekte werden nachfolgend erklärt.

6.4.1 Classloader

In *Java* werden Klassen erst dann geladen, wenn sie im Programmlauf gebraucht werden. Dazu ist jede Klasse in einer eigenen Datei gespeichert. Das Laden der Klassen übernimmt ein sogenannter *Classloader*. Im Ergebnis kommt es durch diese Implementierung des Ladens von Klassen zu einer Verzögerung beim ersten Verwenden eines Objektes der Klasse, da zunächst die entsprechende Klassendatei von der Festplatte geladen und gelesen werden muss. Je nach Umfang der Klasse kann dieser Vorgang einige Millisekunden dauern. Als Verwendung zählt hier schon der Import der Klasse zu Beginn einer Klassendeklaration. Ist die Klassendatei einmal geladen, steht sie bei der nächsten Verwendung aus dem Klassencache zur Verfügung und es fallen nur noch minimale Kosten für das Laden an.

Dieses Verhalten ist in Abbildung 6.5 für Task 0 deutlich sichtbar. Da dieser Aufschlag auf

die Verarbeitungszeit nur bei diesem ersten Task zu erwarten ist, sollte man diese Laufzeit für die Konstruktion der Laufzeitfunktion $rate_{multi}(\dots)$ ausfiltern, da sie dazu führen könnte, dass die Laufzeit der restlichen Tasks deutlich mit einem Fehler behaftet würde. Die ebenfalls hohe Ausführungszeit der folgenden drei Tasks ist nicht durch den *Classloader* zu erklären, sondern durch den später noch vorgestellten JIT-Compiler.

6.4.2 Garbage Collection

In *Java* werden dynamisch allokierte Variablen, die durch den `new`-Operator erzeugt werden, auf dem Heap-Speicher abgelegt. Der Speicherbereich dieser Variablen muss im Gegensatz zu statischen Variablen, die auf dem Programmstack abgelegt werden, explizit wieder freigegeben werden. In *Java* fehlt jedoch eine solche Möglichkeit der expliziten Freigabe. Solche Speicherbereiche werden durch das *Java*-Laufzeitsystem überwacht und automatisch freigegeben, wenn sie nicht mehr benötigt werden. Der Vorgang des automatischen Freigebens wird als *Garbage Collection* (GC) bezeichnet.*Garbage Collection* beeinflusst die Laufzeit eines Programms, da das Freigeben der Speicherbereiche je nach Anzahl und Art der Verwendung Programmlaufzeit und Arbeitsspeicher in Anspruch nimmt.

Da *Garbage Collection* nicht Teil der *Java Language Specification* ist, implementiert jede *Java*-Laufzeitumgebung eventuell verschiedene Verfahren. Deshalb soll in diesem Abschnitt nur auf problematische Konstellationen bei der Speicherbereinigung eingegangen werden. Es wird kurz beschrieben, welche Möglichkeiten der *Garbage Collection* es gibt, um diese Konstellationen bestmöglich zu verarbeiten, und wie sich diese auf die Programmlaufzeit auswirken. Für weiterführende Informationen sei auf [Nag09, Fri12] verwiesen.

Grundlegend muss für den Vorgang der Speicherbereinigung zunächst für jedes Objekt bestimmt werden, ob es noch erreichbar ist. Dazu muss es im Programm mindestens noch eine gültige Referenz auf das Objekt geben. Auch alle von einem erreichbaren Objekt erreichbaren weiteren Objekte sind erreichbar, jedoch eventuell nur indirekt. Um die Erreichbarkeit festzustellen, müssen alle Referenzen gezählt werden. Dieser Zählvorgang, genannt *Reference Counting*, kann zur Laufzeit simultan stattfinden und verteilt sich somit gleichmäßig über die gesamte Programmlaufzeit, jedoch werden durch diese Variante nicht alle nicht mehr erreichbaren Objekte identifiziert. Zusätzlich muss demnach von Zeit zu Zeit das Programm angehalten werden, um die Referenzen für alle Objekte explizit zu bestimmen (*Mark and Sweep*). Nicht mehr erreichbare Speicherbereiche werden dann wieder freigegeben. Wie bei kompiliertem Programmcode auch, treten beim Speichermanagement in *Java* verschiedene Probleme auf, von denen zwei nachfolgend etwas näher betrachtet werden.

Viele kurzlebige Objekte implizieren auch das häufige Ausführen des *Garbage Collectors*. Dabei kann es zur Speicherfragmentierung kommen, die im Falle einer neuen Speicheranforderung dann dazu führt, dass kein zusammenhängender Speicherbereich für die Anfrage mehr gefunden werden kann, obwohl insgesamt genügend freier Speicher vorhanden wäre. Dieses Problem kann nur dadurch gelöst werden, dass nach der Ausführung des *Garbage Collectors* zusätzlich noch die übrigen Speicherbereiche zusammenkopiert (defragmentiert) werden, sodass wieder große freie Lücken enstehen, die für neue Speicheranforderungen genutzt werden können. Entsprechend sind die Varianten *Mark and Compact* und *Stop and Copy* des *Garbage Collectors* vorhanden. Dieses häufige Zusammenkopieren kostet jedoch Zeit und kann in vielen Fällen vermieden werden, wie nachfolgend beschrieben wird.

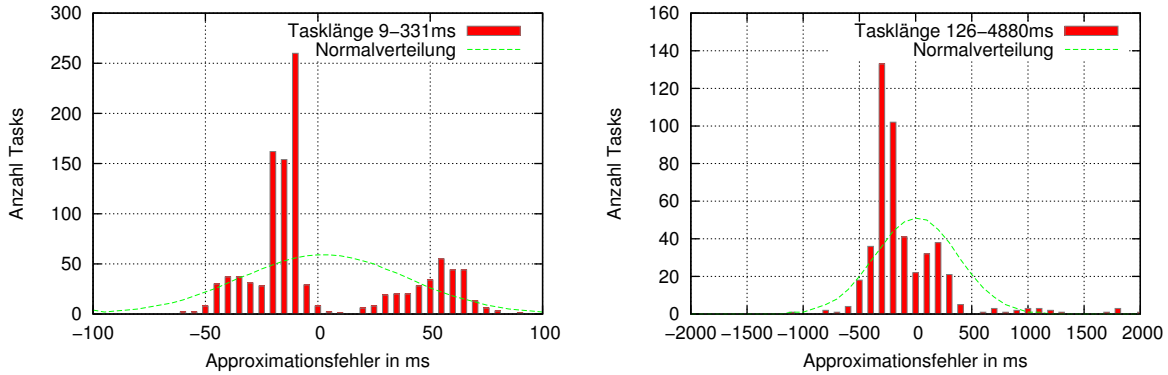


Abbildung 6.6: Häufigkeitsverteilung der Abweichung der Laufzeit einzelner Tasks der Kantenerkennung vom approximierten Wert. *Links:* geringe Bildauflösung, kurze Tasklaufzeiten *Rechts:* hohe Bildauflösung, längere Tasklaufzeiten

Objekte mit sehr unterschiedlicher Lebensdauer sind oftmals in einem Programmlauf gleichzeitig anzutreffen. Diese Tatsache kann man ausnutzen, indem man ein generationelles Speichermanagement implementiert. Dabei wird der Heap in einen *jungen* und einen *alten* Bereich aufgeteilt. Neue Objekte werden im *jungen* Bereich angelegt. Beim Zusammenkopieren werden ältere Objekte in den *alten* Speicherbereich kopiert, da es wahrscheinlich ist, dass diese weiter aktiv bleiben. Demnach werden die älteren Objekte beim recht häufigen Defragmentieren des *jungen* Bereichs nicht mehr betrachtet, was Zeit einspart. Der *alte* Bereich muss nur selten defragmentiert werden.

Insgesamt wirkt sich die Speicherbereinigung stärker auf die Programmlaufzeit aus, je mehr Objekte erstellt und gelöscht werden müssen. Deshalb sollte auch in *Java* mit der Speicherallokation sorgsam umgegangen werden und es sollte nach Möglichkeit Speicher wiederverwendet werden. Für die in dieser Arbeit favorisierte Ausführung des Programmcodes als Tasks wirkt sich der *Garbage Collector* auf sehr kurzlebige Tasks aus, deren Verarbeitungsdauer unterhalb der Ausführungsdauer eines *Garbage Collector* Laufes liegt. Für die getesteten VM-Typen dauert ein Lauf des *Garbage Collectors* typischerweise zwischen 10–100 ms. Einzelne kurzlebige Tasks können dann, sollten sie durch eine Lauf des *Garbage Collectors* (GC-Lauf) unterbrochen werden, plötzlich eine vielfach längere Laufzeit aufweisen. Dieses Phänomen wird dann deutlich in der Regressionsanalyse sichtbar. In Abbildung 6.5 ist ein GC-Lauf an den wenigen deutlich längeren Tasklaufzeiten zu erkennen, die sich über die gesamte Laufzeit verteilen. Dabei handelt es sich jedoch nicht um einen Messfehler, den man einfach ausfiltern könnte. Da die Beeinflussung jedes Task durch einen GC-Lauf gleichwahrscheinlich ist, müssen in diesem Fall mehrere Messungen gemacht werden, aus welchen dann der Mittelwert bestimmt wird.

Abbildung 6.6 stellt die Fehlerhäufigkeiten nach absoluter Abweichung von der Schätzfunktion für relativ kurze Tasks und für etwas längere Tasks dar. Die Messpunkte stammen auch hier von der in Kapitel 8.3 vorgestellten Applikation zur Kantenerkennung. Zusätzlich wurde die erwartete Normalverteilung aufgrund der gemessenen Standardabweichung der Messwerte visualisiert. Zu erkennen ist, dass die Approximationsfehler der längeren Tasks eher einer Normalverteilung entsprechen als die Fehler der kurzen Tasks. Zur Erklärung der Häufigkeitsverteilung der kürzeren Tasklaufzeiten sei noch einmal auf Abbildung 6.5 verwiesen. Dort ist

zu erkennen, dass einige Tasklaufzeiten viel länger sind, als die Mehrheit. Es existieren also zwei Gruppen von Laufzeiten die vom Mittelwert getrennt werden. In Abbildung 6.6 (links) sind diese beiden Gruppen als starke Häufigkeitsansammlung unterhalb der Mitte und oberhalb der Mitte zu erkennen. Die untere Ansammlung reflektiert unbeeinflusste Tasklaufzeiten und die obere Ansammlung reflektiert durch einen GC-Lauf verlängerte Laufzeiten. Aus statistischer Sicht ist eine Normalverteilung für kurze Tasks also nicht mehr gegeben. Praktisch tritt dieser Fall jedoch nur für sehr kurze Tasks auf. Es muss demnach festgehalten werden, dass die Prognosequalität mit der durchschnittlichen Länge der Tasks zunimmt. Für sehr kurze Tasks ist jedoch auch die Übertragungsdauer der Taskdaten eventuell schon höher, sodass die ungenaue Prognose der Tasklaufzeit relativiert wird (siehe Kriterium (6.5) bzw. Kriterium (6.7)).

6.4.3 Just-In-Time Compiler

Java-Bytecode wird interpretiert, um ein *Java*-Programm auf allen Systemen ausführen zu können, die einen *Java*-Interpreter bereitstellen. Diese breite Ausführbarkeit zieht jedoch eine teilweise geringe Verarbeitungsgeschwindigkeit nach sich. Vor allem berechnungsintensive Programme, die viele Programmschleifen enthalten, werden davon stark beeinträchtigt. Um dieses Problem zu beheben, müsste der komplette Bytecode zu Beginn der Ausführung in entsprechenden systemspezifischen Maschinencode kompiliert werden. Dieser Kompilierungsvorgang würde jedoch je nach Programmumfang und Komplexität sehr lange dauern. Um die Programmstartverzögerung zu vermeiden und dennoch berechnungsintensive Programmteile in Maschinencode zu übersetzen, wird im *OpenJDK* ein sogenannter *Just-In-Time Compiler* (JIT-Compiler) eingesetzt, der Programmcode zur Laufzeit selektiv übersetzt. Danach wird interpretierter Bytecode und übersetzter Maschinencode durch die JVM gemeinsam ausgeführt. Man spricht deshalb auch von gemischtem Code (englisch *mixed-mode execution*).

Bei dieser Art der Kompilierung wird ausgenutzt, dass nur ein kleiner Teil des gesamten Programmcodes im größten Teil der Programmlaufzeit genutzt wird. Solche Verteilungen kommen häufig vor und sie sind auch bei der Code-Optimierung in der Informatik anwendbar [ACGS04]. Somit ist es nicht nötig, das gesamte Programm zu kompilieren, um einen deutlichen Speedup zu erzielen. Es reicht, einen kleinen Teil zu kompilieren, um die Performance stark zu verbessern. Es muss jedoch bestimmt werden, welcher Teil zu kompilieren ist.

Um die berechnungsintensiven Programmteile zu identifizieren, werden durch das Laufzeitsystem umfangreiche Statistikinformationen gesammelt. Wird beispielsweise eine Programmschleife sehr oft durchlaufen, so wird nach Erreichen eines Schwellwertes die gesamte umgebende Methode kompiliert. Dieser Kompilierungsvorgang erfolgt sogar nach verschiedenen Optimierungsstufen inkrementell. Wird die Schleife also weiterhin oft durchlaufen, so wird eventuell nach Erreichen eines weiteren Schwellwertes ein erneuter Kompilierungsvorgang gestartet, der eine höhere Optimierungsstufe nutzt. Zusätzlich zur Kompilierung können aufgrund der Statstikinformationen auch weitere Optimierungen, teilweise auf dem Bytecode, zur Laufzeit vorgenommen werden.

Mögliche Optimierungen sind beispielsweise:

- Inlining von Methoden; dabei wird der Programmcode der Methode direkt ausgeführt, ohne einen Methodenaufruf zu veranlassen;
- Eliminierung von Locks, falls diese nur von einem Thread erreicht werden können;

- Übergehen der Interface-Methode beim Methodenaufruf, falls es nur eine einzige Implementierung gibt;
- Verschieben von nicht-volatilen Speichervariablen auf den Stack;
- Löschen von nicht erreichbarem Code.

Die Optimierungen des Just-In-Time Compilers verändern offensichtlich die Laufzeit eines Programms, nicht nur weil die Performance sich bei häufigen Schleifendurchläufen verbessert, sondern auch weil die Kompilierung auch Programmlaufzeit in Anspruch nimmt. Konkret ist zu erwarten, dass berechnungsintensive Programme während der Laufzeit ihre Performance steigern können. Dies geschieht in mehreren Stufen. Durch die Performancesteigerung kann es bei der Vergrößerung der Eingabe zu einem superlinearen Speedup kommen, obwohl der eigentliche mathematische Zusammenhang zwischen Eingabegröße und Ausführungszeit linear ist. Ob und wie stark optimiert wurde, kann man durch eine spezielle Ausgabe der JVM zur Laufzeit nachvollziehen. Dazu sind die Aufrufparameter `-XX:+PrintCompilation` für Informationen über die erfolgten Kompilierungsvorgänge oder `-XX:+CITime` für Informationen über die dafür benötigte Zeit an den `java` Kommandozeilenaufruf anzuhängen. Die benötigte Zeit wird dabei nur kumuliert bei der Terminierung des Programms ausgegeben. Es ist somit nicht möglich, die Dauer für einzelne Kompilierungsvorgänge zu bestimmen.

In Abbildung 6.5 könnte die erhöhte Ausführungszeit der ersten fünf Tasks darauf hindeuten, dass diese durch JIT-Compiler Aktivitäten beeinflusst wurden. Die Auswertung der Logdaten ergab jedoch, dass zwar Taskcode kompiliert wurde, die Gesamtlaufzeit lag jedoch nur bei ca. 500 ms. Somit kann die Laufzeit des ersten Task nicht nur durch den JIT-Compiler beeinflusst worden sein. Die Auswirkungen auf die Laufzeit sind in Abbildung 6.5 nicht von denen des *Garbage Collectors* unterscheidbar. Im Allgemeinen ist der Einfluss der JIT-Compiler als gering zu bewerten. Dennoch kann die Performance zu Beginn der Ausführung leicht beeinflusst werden. Es sollte für jede Taskart geprüft werden, ob über das Ausfiltern der ersten Tasklaufzeit aufgrund erhöhter Ladezeiten der Klassendateien auch noch weitere Tasklaufzeiten aufgrund der erhöhten Aktivität des JIT-Compiler auszufiltern sind. Im vorliegenden Beispiel wurde nicht ausgefiltert, da der Einfluss nicht als überproportional eingeschätzt wurde. Vor allem beim Wechsel der JVM sollte jedoch der Einfluss des JIT-Compiler erneut untersucht werden.

6.5 Fazit

Ziel dieses Kapitels war es, eine Technik zu entwickeln, die für *Mobile Cloud-unterstützte Anwendungen* passende VM-Typen zur Laufzeit anhand bestimmter Parameter auswählen kann. Dazu wird die Ausführungsperformance vorhergesagt. Zunächst wurden mögliche taskbasierte Applikationen kategorisiert und es wurden für die Auswahl zur Laufzeit abschließend nur

- aus unabhängigen Einzeltasks bestehende benutzerterminierte Applikationen
- und aus sequentiellen Tasks bestehende eingabeterminierte Applikationen

zugelassen. Andere Arten von Applikationen können aber durch entsprechende Einschränkungen auf eine der Kategorien abgebildet werden. Darauf aufbauend wurden Algorithmen zur Bestimmung von Cloud-Optionen zur Laufzeit vorgestellt, anhand derer ein Benutzer sinnvolle Ausführungsoptionen wählen kann. Als Ergebnis wurde festgestellt, dass die Qualität der

Vorhersage der Tasklaufzeit und die Wahl der verfügbaren VM-Typen einen großen Einfluss auf die Bestimmung von Cloud-Optionen haben. Diese Aspekte wurden genauer untersucht.

Für die Vorhersage der Tasklaufzeiten der beiden unterstützten Applikationsarten wurden zunächst durch Regressionsanlyse Vorhersagefunktionen bestimmt. Für die Regressionsanalyse müssen für jede Art Task auf allen verfügbaren Ressourcentypen aussagekräftige Messreihen der Laufzeit unter Verwendung verschiedener Eingaben durchgeführt werden. Die Eingaben sollten dabei die maximale Spannweite aller möglichen Eingaben abdecken, um später auch für diese Werte gute Vorhersagen treffen zu können. Es ist jedoch von Bedeutung, in welchem Ausmaß die tatsächlichen Messwerte um die Vorhersage schwanken. Aus der jeweiligen Schwankungsbreite der Tasklaufzeiten ergibt sich auch, welche Ressourcentypen im Verhältnis zueinander konfiguriert werden sollten. Schwanken die Werte stark, so sollte die Performance der verfügbaren VM-Typen weiter auseinander liegen, als wenn die Werte nur leicht schwanken. Um dies zu analysieren, wurden VM-Typen, die *Java*-Laufzeitumgebung und die Applikationen selbst auf deren Verhalten untersucht.

Zunächst wurden verfügbare VM-Typen und insbesondere Ressourcentypen von *Amazon EC2* auf deren Performanceschwankung untersucht. Es wurde ermittelt, dass diese im unteren zweistelligen Prozentbereich liegt. Performanceschwankungen verschiedener JVM-Implementierungen konnten als unbedeutend vernachlässigt werden. Weiterhin wurde unter anderem auch untersucht, wie die Schwankungsbreite von Tasklaufzeiten durch spezielle Eigenschaften der *Java*-Laufzeitumgebung beeinflusst wird. Dazu zählen *Garbage Collection*, der *Classloader* und der *Just-in-Time* Compiler. Es wurde festgestellt, dass der Einfluss des *Garbage Collectors* für kurze Tasklaufzeiten groß ist und mit steigender Laufzeit abnimmt. Außerdem ist es oft ratsam, die ersten Tasklaufzeiten einer Reihe von Tasks für die Regressionsanalyse auszufiltern, da diese stark durch verzögertes Klassenladen und Aktivitäten des *Just-in-Time* Compilers beeinflusst werden.

Abschließend wurde festgelegt, dass der Approximationsfehler aller Messdaten für eine Approximationsfunktion ein Gütekriterium $K_{Q_{gut}}$ einhalten muss (siehe Abschnitt 6.2.4). Bei einer angenommenen Normalverteilung der Fehler entspricht dies etwa der Forderung, dass mindestens 80% aller Werte um weniger als 50% vom Wert der Schätzfunktion abweichen. Wird diese Forderung eingehalten, dann können jeweils VM-Typen ausgewählt werden, deren Performance um ca. den Faktor 4 verschieden ist. Wird dies nicht erreicht, so kann es entweder zu großen Überschneidungen in der Schwankungsbreite für die verschiedenen VM-Typen kommen, was dann gehäuft zur falschen Auswahl des VM-Typs führen kann. Andererseits kann auch durch Vergrößerung des Performanceabstands der VM-Typen versucht werden, eine Fehlauswahl zu vermeiden, jedoch stehen dann eventuell nur eine oder sehr wenige VM-Typen zur Auswahl.

Für andere IaaS-Anbieter wäre die Einschränkung bezüglich des Gütekriteriums $K_{Q_{gut}}$ ebenso zu übernehmen. Je nach Schwankungsbreite der VM-Verarbeitungsgeschwindigkeit reagiert die entsprechende Applikation jedoch anders. Für weitere IaaS-Anbieter wären also ebenso die Tasklaufzeiten auf verschiedenen Ressourcen zu messen, zu analysieren und entsprechend verschiedene Ressourcentypen zu konfigurieren.

7 Middleware-Implementierung für Java

In den vorangegangenen Kapiteln wurden Modellierungs- und Performanceaspekte für *Mobile Cloud-unterstützte Anwendungen* untersucht. Aufbauend auf den Ergebnissen wird in diesem Kapitel eine Middleware-Implementierung vorgestellt, die das Erstellen solcher Anwendungen in der Programmiersprache *Java* ermöglicht. Die Middleware ist für *Java* implementiert, da *Java* für viele Mobilgeräte sowie für fast alle Desktop- und Server-Betriebssysteme verfügbar ist (siehe Kapitel 2.4). Die Middleware soll Benutzer und Entwickler bei der Erstellung und Nutzung von *Mobilen Cloud-unterstützten Anwendungen* unterstützen und dazu die erarbeiteten Algorithmen und Techniken implementieren. Dazu zählen:

- die effiziente Kommunikations- und Taskverarbeitung in kabellosen Rechnernetzwerken,
- die bedarfsgerechte Auswahl und Bereitstellung von VMs zur Laufzeit
- und die Überwachung der *Mobilen Cloud-unterstützten Anwendungen* zum Zweck der Verbesserung der Auswahlstrategie.

Die Middleware besteht aus verschiedenen Komponenten in Form von Servern, die auf virtuellen Maschinen in einem Cloud-Rechenzentrum eines IaaS-Anbieters ausgeführt werden. Eine vorteilhafte Architektur für die Middleware wurde in Kapitel 5.2 erarbeitet. Sie besteht neben den VMs aus einem Front-Server und Servern für Benutzerverwaltung und Abrechnung. Die Komponenten stellen über ihre APIs Funktionalitäten zur Implementierung von *Mobilen Cloud-unterstützten Anwendungen* zur Verfügung. Zudem ist als zweiter Bestandteil eine Programmbibliothek Teil der Middleware, die die Benutzung der Server-APIs durch *Java*-Anwendungen vereinfacht und abstrahiert. Zur Entwicklung von *Mobilen Cloud-unterstützten Anwendungen* wird ausschließlich diese Programmbibliothek benötigt. Vom Entwickler wird nur das clientseitige Programm erstellt, welches dann unter Verwendung der Programmbibliothek in der Lage ist, VMs zur Unterstützung bei berechnungsintensiven Aufgaben hinzuzuziehen. Serverseitiger Programmcode muss nur zur Installation des Applikationsprofils erstellt werden, welches zur bedarfsgerechten Auswahl von VM-Ressourcen zur Laufzeit benötigt wird.

Die Anwendungsschnittstelle der Middleware ist im Cloud-Schichtenmodell auf Plattform-Ebene angesiedelt. Die Implementierungsdetails der darunter liegenden Infrastruktur-Schicht bleiben dabei verborgen (siehe Kapitel 1.1.2). Im Unterschied zu Plattform-APIs anderer Cloud-Anbieter ist es jedoch hier möglich, den zu verwendenden VM-Typ der Infrastruktur-Schicht zu beeinflussen. Dadurch wird der Nutzer in die Lage versetzt, die Verarbeitungsgeschwindigkeit der entfernt ausgeführten Programmmodule proaktiv zu beeinflussen und den aktuellen Bedürfnissen anzupassen. Weitere Anforderungen an die Middleware sind die flexible Erfassung und Abrechnung der verwendeten Ressourcen/Netzwerktraffic/etc. und die Implementierung einer Lastbalancierungsstrategie, die die effiziente VM-Nutzung sicherstellt (siehe Kapitel 5.4). Seitens des IaaS-Anbieters wird dabei nur eine API vorausgesetzt, die es ermöglicht, zu jedem Zeitpunkt VM-Images auf VM-Typen zu starten und zu beenden.

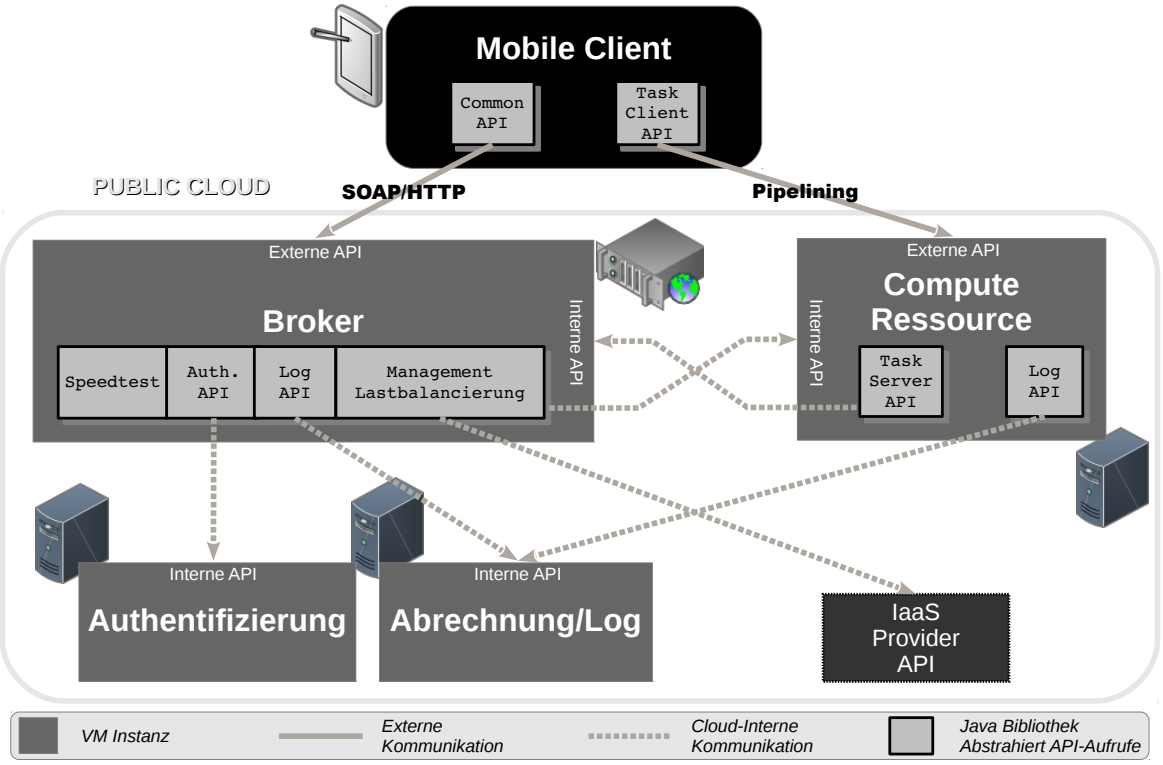


Abbildung 7.1: Illustration der Middleware-Komponenten und deren Interaktion.

Das Kapitel ist wie folgt organisiert: Zunächst werden die einzelnen Komponenten und deren APIs sowie die zugehörigen Programmbibliotheken vorgestellt. Anschließend wird anhand von Beispielen die Verwendung der Programmbibliothek zur Implementierung von *Mobilen Cloud-unterstützten Anwendungen* illustriert.

7.1 Middleware-Komponenten

Zur Unterstützung des Client wurde ein Aufbau der Middleware aus vier Komponenten vorgeschlagen. Die Middleware besteht aus Broker, Compute-Ressource (VM), Authentifizierung und Abrechnung. Der Aufbau ist in Abbildung 7.1 dargestellt. Zusätzlich sind auch die entwickelten *Java*-Programmbibliotheken dargestellt, auf deren Aufgaben und Implementierung im nachfolgenden Abschnitt näher eingegangen wird. Jede Komponente wird dabei auf eine dedizierte virtuelle Maschine innerhalb eines Cloud-Rechenzentrums abgebildet. Für alle Komponenten wird ein spezielles VM-Image bereitgestellt, welches bereits eine Implementierung des notwendigen Softwarestacks enthält und nur gestartet werden muss. Als Betriebssystem für alle VM-Instanzen wird ein *Amazon Linux*[1] benutzt, welches neben der

[1] *„Das Amazon Linux AMI (Amazon Machine Image) ist ein von Amazon Web Services bereitgestelltes, unterstütztes und gepflegtes Linux-Image zur Verwendung auf Amazon EC2 (Amazon Elastic Compute Cloud). Es bietet eine stabile, sichere und High-Performance-Ausführungsumgebung für Anwendungen auf Amazon EC2. Außerdem enthält es Pakete für die einfache Integration mit AWS, darunter Startkonfigurations-Tools und viele weitverbreitete AWS-Bibliotheken und -Tools. Amazon Web Services bieten laufend Sicherheits- und Wartungsaktualisierungen für alle Instances, auf denen das Amazon Li-*

Tabelle 7.1: Kategorisierung der API-Methoden der Middleware-Komponenten

	intern	**extern**	
Broker	Sessionmanagement	Ressourcenallokation	Ressourcenauswahl
	`returnKey`	`getInstanceForType`	`deploy`
			`undeploy`
			`calculateMCsingle`
			`calculateMCmulti`
			SPEEDTEST (Web)
VM	Sessionmanagement	Taskverarbeitung	
	`sendeSessionkey`	`pipelining`	
Authentifizierung	Benutzerverwaltung		
	`authenticate`	Bedeutung:	`SOAP-API`
Log/Abrechnung	Logging		`Socket-API`
	`writeLogEntry`		`proprietäre-API`

notwendigen *Java*-Laufzeitumgebung auch Kommandozeilenwerkzeuge zur Verwendung der *Amazon EC2* Provider-API bereitstellt. Es wurde auf eine minimale Softwareausstattung der VM-Instanzen geachtet, damit Performance und Sicherheit kaum beeinträchtigt werden.

Authentifizierung und Abrechnung sind durch Datenbanken realisiert. Der Broker ist eine zentrale Komponente, die als Frontend fungiert und Nutzeranfragen von außen direkt entgegennimmt und beantwortet. Authentifizierung und Abrechnung sind von außen nicht erreichbar und werden nur von Komponenten innerhalb des Cloud-Rechenzentrums verwendet.

Jede Komponente bietet eine interne Schnittstelle an, die von anderen Komponenten genutzt werden kann. Einzig Broker und VMs bieten auch externe Schnittstellen an, um mit den Clients kommunizieren zu können. Tabelle 7.1 fasst die verschiedenen Schnittstellen der einzelnen Komponenten zusammen und kategorisiert sie nach deren Verwendung. Außerdem ist dargestellt, welche Kommunikationsprotokolle je nach Aufgabe für die einzelnen API-Methoden benutzt werden. Der eigentliche Programmcode zur Unterstützung der mobilen Applikationen wird auf den VMs ausgeführt. Zwischen diesen und den Clients im Internet findet der Großteil der Kommunikation statt. Hier fließen die Nutzdaten. Dieses auf Pipelining basierende Kommunikationsprotokoll ist in Kapitel 4.3 vorgestellt worden und verwendet kein allgemein standardisiertes Protokoll. *SPEEDTEST* ist ebenfalls in Kapitel 4.3 bereits vorgestellt worden und verwendet ein auf HTTP basierendes Kommunikationsprotokoll, welches nicht öffentlich publiziert ist. Details zu Aufgaben und Implementierungen der einzelnen Schnittstellen werden im nachfolgenden Abschnitt vorgestellt.

Die bereitgestellte Programmbibliothek kapselt unter anderem API-Aufrufe zu den Komponenten (beispielsweise der komplexen Broker-API) und reduziert so den Implementierungsaufwand für den Entwickler. Besonders seitens des Clients und der VM ist die Programmbibliothek wichtig, da sie die effiziente Kommunikationstechnologie auf Basis des Task-Konzepts und des Pipelinings implementiert (siehe Kapitel 4.3). Die verfügbaren Programmbibliotheken sind in Abbildung 7.1 ebenfalls dargestellt. Ihre Funktionalität wird später in Kapitel 7.3 genauer erläutert. Zunächst wird jedoch ein Überblick gegeben, welche Softwarepakete in *Java* zur Verfügung stehen und wo diese benutzt werden:

nux AMI ausgeführt wird. Das Amazon Linux AMI wird Amazon EC2-Nutzern kostenlos bereitgestellt.“ (Onlineressource `http://aws.amazon.com/de/amazon-linux-ami/`(abgerufen am 29.10.2013))

`ferb.auth` stellt für die Authentifizierung notwendige Klassen zur Verfügung. Dieses Paket wird auf dem Broker und der Authentifizierung verwendet.

`ferb.log` stellt für die Abrechnung notwendige Klassen zur Verfügung. Dieses Paket wird auf der Abrechnung, den VMs und dem Broker verwendet.

`ferb.master` implementiert die Logik des Broker. Dazu zählt die Lastbalancierung von VM-Instanzen. Außerdem wird auch die Funktionalität zur Ressourcenauswahl zur Laufzeit implementiert. Dieses Paket wird nur auf dem Broker benötigt.

`ferb.common` stellt für den Client Funktionalität für die Ressourcenauswahl zur Laufzeit bereit. Dazu zählen die Kapselung von Broker-API Aufrufen für *Android* und Methoden zur Abschätzung der Netzwerkperformance aufgrund gemessener Parameter. Dieses Paket wird nur auf den Clients benötigt.

`ferb.pipe` implementiert das Transferieren von Tasks unter Verwendung des Pipeliningprotokolls zwischen Client und VM. Dieses Paket wird nur auf den Clients und den VMs verwendet.

Für den Entwickler von *Mobilen Cloud-unterstützten Anwendungen* sind demnach nur die Pakete `ferb.log`, `ferb.common` und `ferb.pipe` wichtig. Die clientseitige Programmbibliothek kann gleichermaßen unter *Android* und JVMs wie dem *OpenJDK* verwendet werden. Die übrigen Pakete dienen der Implementierung der Middlewarekomponenten. Im nachfolgenden Kapitel werden die Netzwerkschnittstellen in ihrer Funktion und Benutzung genauer beschrieben.

7.2 Netzwerk-APIs der Komponenten

Alle Datentransfers zwischen den Komponenten der Middleware finden verschlüsselt statt. Dazu wird die SSL/TLS-Verschlüsselung unter Verwendung von Zertifikaten benutzt (siehe Kapitel 4.2.2). Da die internen API-Methoden nur von Systemkomponenten benutzt werden, wird die Authentifizierung mit Hilfe von SSL-Zertifikaten gelöst, denen die einzelnen Komponenten gegenseitig vertrauen. Die internen Schnittstellen sind bis auf die des Broker als TCP-Socket-Verbindungen ausgeführt. Nur externe Schnittstellen sind für die Entwicklung von *Mobilen Cloud-unterstützten Anwendungen* relevant, somit kann die interne Kommunikation leicht an spezifische Gegebenheiten der Infrastruktur-Schicht angepasst werden.

TCP-Sockets in *Java* wurden für die interne Kommunikation gewählt, da sie zertifikatsbasiertes SSL standardmäßig unterstützen. Zusätzlich wird bei Socket-Verbindungen kein Overhead für die Datencodierung benötigt, da Bytes direkt transferiert werden. Dies ist für die internen Schnittstellen ausreichend, weil ausschließlich Text übertragen wird. Zudem kann auf Komprimierung verzichtet werden, da interne Datentransfers zwischen VM-Instanzen innerhalb des gleichen Rechenzentrums stattfinden und somit eine hohe Übertragungsgeschwindigkeit gewährleistet ist (>100 Mbit). Alle internen Verbindungen arbeiten nach dem Request-Response Muster. Nach dem Aufbau der Verbindung wird nur eine einzelne Anfrage und darauf folgend eine Antwort gesendet. Danach wird die Verbindung sofort getrennt. Interne APIs sind zusätzlich durch eine Firewall des IaaS-Anbieters geschützt und können aus dem Internet nicht angesprochen werden.

Die externe Broker-API ist bis auf *SPEEDTEST* als SOAP-API ausgeführt, da sie in verschiedene Anwendungen integriert werden muss und SOAP dabei vielfältige Möglichkeiten bietet. Die Implementierung der internen API ist dabei im Gegensatz zu allen anderen internen APIs auch als SOAP-API gewählt worden, um die Gleichartigkeit der Broker-API zu erreichen. Sie kann bei Bedarf auch anders realisiert werden. Bei *SPEEDTEST* handelt es sich um eine nicht selbst entwickelte Software, deren HTTP-basierte Web-API bereitgestellt wird und nicht verändert werden kann. Beim Aufrufen der externen SOAP-API des Broker sind stets Benutzername und Passwort anzugeben. Die Authentifizierung des Benutzers bzw. der Applikation auf den VMs erfolgt dagegen wie in Kapitel 5.2.4 beschrieben über einen *Sessionkey*. Die verschlüsselte Kommunikationsverbindung zwischen den VMs und den Clients unter Verwendung des Pipeliningmechanismus wurde bereits in Kapitel 4.3 beschrieben. Die VMs bieten ausschließlich dieses Protokoll über ihre externe API an.

Üblicherweise erhalten neu gestartete VM-Instanzen stets verschiedene IP-Adressen. Bei der Verwendung von SSL-Zertifikaten führt dies zu einem Problem, da Zertifikate an IP-Adressen oder Hostnamen gebunden werden. Als Ergebnis schlägt die Verifizierung des Zertifikats bei abweichender IP-Adresse fehl. Um dieses Problem bei den Komponenten Broker, Authentifizierung und Abrechnung zu lösen, bieten IaaS-Anbieter oft statische IP-Adressen oder Namensauflösungsdienste an, die gemietet werden können. Somit bleiben Name oder IP-Adresse dieser Komponenten gleich und die Verifizierung funktioniert. Da die Anzahl von VMs jedoch schwankt, muss hier ein anderer Mechanismus angewendet werden. Für die VMs wird ein Wildcard-Zertifikat verwendet, welches für alle Instanzen innerhalb des Bereichs des IaaS-Anbieters gültig ist (beispielsweise `*.eu-west-1.compute.amazonaws.com` statt `ec2-54-228-14-184.eu-west-1.compute.amazonaws.com`).

Bis auf die Schnittstelle zur IaaS-Anbieter-API sind nachfolgend alle Schnittstellen der einzelnen Komponenten in Pseudocode angegeben. Die Schnittstelle des verwendeten Anbieters *Amazon EC2* ist als SOAP *Web Service* verfügbar und kann über verschiedene Bibliotheken unter anderem in den Sprachen *Java* und Python angesprochen werden[2]. Alle IaaS-Anbieter Aufrufe werden vom Broker durchgeführt.

7.2.1 Broker

Die API-Methoden des Broker sind bis auf *SPEEDTEST* als SOAP *Web Service* ausgeführt (siehe Tabelle 7.1). Als gemeinsame Ausführungsplattform wurde ein Applikationsserver verwendet, der die Methoden der internen API, der externen API und *SPEEDTEST* als einzelne Applikationen beherbergt. Der Applikationsserver bündelt die Konfiguration und das Logging an zentraler Stelle und wurde deshalb jeweils einzelnen Serverprogrammen vorgezogen.

Die Erzeugung der *Web Services* erfolgte mit Hilfe von *Apache Axis2*[3] und nutzt auch

[2] Amazon stellt zur Verwendung seiner EC2-SOAP-API Programmbibliotheken für verschiedenste Programmiersprachen bereit. Ebenso werden die Entwicklungsumgebungen *Eclipse* und *Visual Studio* unterstützt. Auf dem bereits erwähnten Amazon Linux sind ebenfalls Kommandozeilenwerkzeuge installiert, die zum Ansprechen der EC2-API geeignet sind. (siehe Onlineressource `http://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.html` (abgerufen am 30.10.2013))

[3] Apache Axis (Apache eXtensible Interaction System) ist eine Laufzeitbibliothek zur Realisierung von darauf basierenden *Web Services* und Client-Anwendungen. Apache Axis 2 ist dabei eine komplette Neuimplementierung von Apache Axis. Die aktuelle Version stammt aus dem Jahr 2012. Diese Arbeit stützt sich auf die Verwendung von Axis2/Java. Ebenso ist eine Axis2/C Implementierung vorhanden. Zum Umfang gehören die Werkzeuge JAVA2WSDL und WSDL2JAVA, die das Erzeugen von *Web Services* aus *Java*-Klassen maßgeblich vereinfachen. (siehe [JA11])

die *Apache Axis2* Laufzeitkomponenten (beispielsweise zur Verarbeitung der SOAP XML-Nachrichten). Mit deren Hilfe lässt sich ein *Web Service* als standardisierte *Java*-Web-Komponente erzeugen (*Web Application Archive* (WAR)), die dann auf einem konformen Applikationsserver ausgeführt werden kann. Grundlage ist die *Java Servlet Spezifikation*[4], die die Definition und Ausführung von *Java*-Klassen auf einem Server spezifiziert. Als Server wurde hier der *Apache Tomcat* Applikationsserver[5] benutzt.

Bei allen externen API-Methoden sind stets Benutzername und Passwort anzugeben. Deshalb findet sich in allen externen SOAP-API-Methoden die Parameter `String user` und `String pass` wieder. Die externe Broker-API ist nur über eine verschlüsselte SSL-Verbindung zu verwenden. Die Dienste der Middleware stehen nur registrierten Benutzern zur Verfügung. Diese Einschränkung ist notwendig, um die genutzten VMs auch in Rechnung stellen zu können. Die WSDL-Beschreibung der externen API-Methoden ist in Anhang D zu finden. Nachfolgend wird die Funktionsweise der einzelnen externen und internen API-Methoden des Broker genauer erklärt.

Ressourcenallokation

Die Methode `getInstanceForType(...)` allokiert für einen Client eine Cloud-Ressource und etabliert für den Anfragenden gleichzeitig einen *Sessionkey*. Sie markiert den Beginn eines Cloud-Nutzungszyklus, wie er in Kapitel 5.2.4 beschrieben ist. Sie hat folgenden Funktionsprototyp und liefert eine Struktur vom Typ `CloudSession` zurück:

```
CloudSession getInstanceForType(String name, String type, String user, String pass)
```

Dabei beschreibt `name` den Namen des angeforderten Dienstes. Im Normalfall ist dieser als "PIPE" anzugeben, da nur VM-Images des Pipelining Typs verfügbar sind. Für Erweiterungen der Middleware können hier neue VM-Images hinterlegt werden. Der Parameter `type` spezifiziert die gewünschte Qualität der Cloud-Ressoure, also den VM-Typ. Die zurückgelieferte Struktur enthält die konkrete IP-Adresse (`endpoint`) der allokierten VM und den erstellten *Sessionkey* (`session_key`), der benötigt wird, um vom Client eine Verbindung zu der VM aufzubauen.

```
struct CloudSession {
    String endpoint;
    String session_key;
};
```

[4] Servlets stellen eine Java-spezifische Variante dar, um Web-Inhalte dynamisch nach dem Inhalt einer Anfrage zu erzeugen und auszuliefern. Dazu spezifiziert die Servlet-API, wie eine Java-Klasse auszusehen hat, die dann Anfragen von außen, oft via HTTP, entgegennehmen und beantworten kann. Die Klassen werden von einem sogenannten Servlet-Container ausgeführt, der die eingehenden Anfragen auf die enstprechenden Servlets zuweisen kann. Ähnliche Konzepte sind beispielsweise durch CGI-Skripte oder PHP-Skripte für andere Programmiersprachen realisiert worden. (siehe [HC10])

[5] *Apache Tomcat* ist ein quelloffener, in Java geschriebener Webserver, der die Spezifikation für Java Servlets und Java Server Pages (JSP) implementiert und es damit erlaubt, in Java geschriebene Web-Anwendungen auf Servlet- bzw. JSP-Basis auszuführen. Ursprünglich von *Sun Microsystems* gestartet, wird der Programmcode seit 1999 von einer Gemeinschaft unter der *Apache Software Foundation* weiterentwickelt. Er ist heute als Modul auch in sogenannten Enterprise Applikationsservern integriert, beispielsweise *Apache Geronimo*. (siehe [MM07])

Ressourcenauswahl

Die API zur Ressourcenauswahl wird aus den Methoden `deploy(...)`, `undeploy(...)`, `calculateMCmulti(...)` und `calculateMCsingle(...)` gebildet. Diese vier externen Methoden werden benutzt, um dem Client eine Auswahl sinnvoller VM-Typen zu erstellen, die für die aktuelle Situation und Applikation verfügbar sind. Dabei wird dem Client die Menge Mc zurückgeliefert, deren Berechnung in Kapitel 6.1.3 vorgestellt wurde. Für jede Applikation muss als Datenbasis zunächst ein Profil installiert werden, aus dem dann Mc berechnet werden kann. Ein Profil enthält neben den möglichen VM-Typen auch Methoden zur Schätzung der Laufzeit der Applikation auf diesen VM-Typen. Für das Installieren und Integrieren einer neuen Applikation in das System steht die externe API-Methode `deploy(...)` zur Verfügung. Die Methode hat folgenden Prototyp:

```
bool deploy(String name, String[] lauf, String[] index,
            float[] kosten, String VM_image, float trafficcost, String user, String pass)
```

Die beiden Arrays `index` und `kosten` enthalten die auswählbaren VM-Typen und deren jeweilige Abrechnungskosten in $ pro Stunde in der gleichen Reihenfolge. Beispiel: `index={c1.medium, c1.xlarge}; kosten={0.13, 1.32};` Zu beachten ist hierbei, dass damit die Kosten gemeint sind, die dem Nutzer in Rechnung gestellt werden. Diese sind meist höher als die Kosten, die vom IaaS-Provider als Miete für Cloud-Ressourcen anfallen. Der zu erwartende Aufschlag ist von der Anfragehäufigkeit, der Anfragelänge und der Ressourcenallokationsstrategie abhängig und wurde in Kapitel 5.4 durch Simulation bestimmt. Üblicherweise wird für `name` wieder stets "PIPE" angegeben. Das zu verwendende `VM_image` ist ebenfalls anzugeben. Dieses muss bereits vorher beim IaaS-Anbieter registriert worden sein, aber auch hier handelt es sich üblicherweise stets um das gleiche Image. Das Array `lauf` enthält den Quellcode von Klassen, die das Verhalten verschiedener Applikationen beschreiben. Der Name der Klasse spezifiziert dabei den Name der Anwendung, für die sie die Laufzeitfunktion $lauf(\dots)$ bereitstellt. Die Anwendung wird auch als Subtyp bezeichnet, da alle Anwendungen auf VMs des "PIPE"-Typs ausgeführt werden. Die genaue Spezifikation und Konstruktion des Quellcodes wird später in Kapitel 7.3.4 anhand eines Beispiels erläutert. Die Datenübertragungskosten sind ebenfalls in $ pro GigaByte als `trafficcost` anzugeben. Sie werden später bei der Berechnung von Cloud-Optionen benötigt, um den Anteil der Datenübertragungskosten zu berechnen.

Zur Deinstallation von Applikationen steht die Methode `undeploy(...)` zur Verfügung. Sie hat folgenden Prototyp:

```
bool undeploy(String name, String user, String pass)
```

Eine direkte Modifikation von bereits installierten Applikationen, beispielsweise des Mietpreises konfigurierter VM-Typen, ist nicht vorgesehen. Hierzu muss immer deinstalliert und erneut installiert werden.

Die eigentliche Berechnung passender Cloud-Optionen zur Laufzeit wird durch die Methoden `caculateMCsingle(...)` und `calculateMCmulti(...)` für eingabeterminierte Applikationen beziehungsweise für benutzerterminierte Applikationen realisiert (siehe Kapitel 6.1.1). Sie werden zur Laufzeit mit den entsprechenden Parametern vom Client aufgerufen. Die Methode `calculateMCsingle(...)` hat folgenden Prototyp:

```
List<CloudOption> calculateMCsingle(String user, String pass, String name, String subtype,
                                    int[] size, float t_comm, float t_mobil, int downloadsize)
```

Dabei werden die für die Laufzeitschätzung notwendigen Parameter im Array `size` übergeben und die rein mobile Vergleichslaufzeit `t_mobil` wird in Sekunden angegeben. Die geschätzte Kommunikationszeit `t_comm` wird ebenfalls übergeben. Die dafür notwendigen Parameter müssen, wie in Kapitel 4 beschrieben, vorher gemessen und mit Bibliotheksunterstützung in die Kommunikationszeit umgewandelt werden. Die dafür vorgesehenen Bibliotheksfunktionen werden später bei der Vorstellung der clientseitigen Bibliotheken noch näher erläutert. Abschließend wird die insgesamt zu erwartende Größe der Downloaddaten in Byte als `downloadsize` ebenfalls übergeben. Aus dieser Größe werden die Datenübertragungskosten berechnet, die im Fall des in dieser Arbeit verwendeten IaaS-Anbieters *Amazon EC2* nur für Downloaddaten anfallen. `calculateMCsingle(...)` liefert, wie auch `calculateMCmulti(...)`, eine Liste von möglichen Cloud-Optionen zurück.

Die Methode `calculateMCmulti(...)` hat folgenden Prototyp:

```
List<CloudOption> calculateMCmulti(String user, String pass, String name, String subtype,
                                   int[] size, float f_comm, float f_mobil, int downloadsize)
```

`calculateMCmulti(...)` unterscheidet sich von `calculateMCsingle(...)` dadurch, dass statt einer mobilen Laufzeit und einer Kommunikationszeit hier jeweils eine Rate ausgeführter Tasks und eine Rate möglicher Taskübertragungen pro Sekunde anzugeben ist. Als `downloadsize` ist hier die Downloaddatengröße in Byte pro Task anzugeben. Eine zurückglieferte `CloudOption` repräsentiert eine mögliche sinnvolle Option eines VM-Typs zur Ausführung der angefragten Applikation und die dann zu erwartenden Eigenschaften. Sie hat folgenden Typ:

```
struct CloudOption {
    String type         //Instanztyp (Ressource)
    float  t_remote     //Ausführungszeit oder Rate auf dieser Ressource
    int    cost         //Kosten (pro Sekunde) auf dieser Ressource in cent
};
```

Für eingabeterminierte Applikationen enthält eine Cloud-Option den VM-Type (`type`), die Ausführungszeit eines Tasks des angegebenen Typs (`t_remote`) und die dafür anfallenden Kosten (`cost`). Bei benutzerterminierten Applikationen ist statt der Gesamtausführungszeit eine Rate angegeben. Für die Rate sind die Kosten pro Sekunde angegeben, für die Gesamtausführungszeit sind die Gesamtkosten angegeben.

Sessionmanagement

Das Zurückgeben des *Sessionkey* erfolgt nach dem Trennen der Verbindung durch den Client von der allokierten VM-Ressource durch die VM selbst (siehe Kapitel 5.2.4). Dazu besitzt der Broker eine interne Schnittstelle mit folgendem Funktionsprototyp, die von der VM aufgerufen wird:

```
bool returnKey(String session_key)
```

`returnKey(...)` ist ebenfalls als SOAP *Web Service* realisiert, da die Methode neben anderen als Einzelapplikation auf dem Applikationsserver des Broker installiert ist.

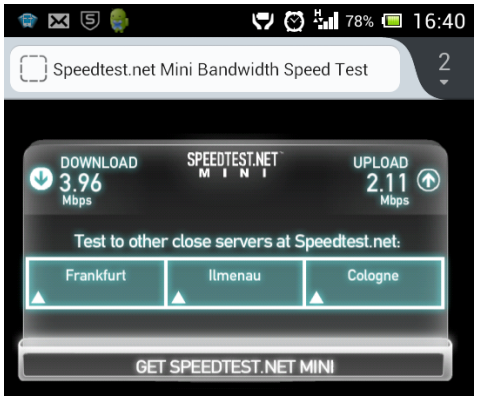


Abbildung 7.2: Einsatz von *SPEEDTEST* Mini zur Bestimmung der Bandbreite zum Broker-Server gemessen auf einem Sony Xperia P Smartphone (siehe Anhang A).

SPEEDTEST zur Bestimmung der Kommunikationsperformance

SPEEDTEST.NET MINI[6], im Weiteren einfach *SPEEDTEST* genannt, wurde bereits in Kapitel 4.4 benutzt, um die Kommunikationsperformance für die Vorhersage der Datenübertragungszeit zu bestimmen. Deshalb ist *SPEEDTEST* Teil der Broker-API, wird jedoch nicht via SOAP angesprochen. *SPEEDTEST* ist als kostenlose Web-Applikation für *Java Servlet Container* verfügbar. Sie wurde im gleichen *Apache Tomcat* Server installiert, der bereits für die Realisierung der restlichen API-Methoden des Broker verwendet wird. Das Servlet liefert zunächst eine Webseite aus, welche ein Flash-basiertes Applet[7] enthält. Über dieses werden vom Broker Dateien heruntergeladen und ebenso hochgeladen. Aus der Übertragungszeit wird die ausgenutzte Übertragungsbandbreite bestimmt. Dieses Werkzeug ist flexibel einsetzbar. Es kann von einem Mobilgerät oder von einem Desktop-PC aufgerufen werden. Einzig ein Flash-fähiger Webbrowser ist Voraussetzung. Ein Screenshot des Messergebnisses zwischen dem Broker und einem Smartphone ist in Abbildung 7.2 zu sehen.

SPEEDTEST wurde aufgrund seiner kostenlosen Verfügbarkeit, der einfachen Installation und Anwendung und ebenso wegen seiner Ausgereiftheit eingesetzt und es wurde auf eine Eigenentwicklung der Bandbreitenbestimmung verzichtet. Zudem wurde in Kapitel 4 festgestellt, dass zur realistischen Bewertung der aktuellen Netzwerkperformance mehrere Messungen durchgeführt werden sollten. Es ist hier schwierig eine automatisierte Bewertung durchzuführen. Deshalb obliegt es dem Nutzer, die tatsächlich verfügbare Performance vor der Bestimmung von Cloud-Optionen mit Hilfe von *SPEEDTEST* möglichst gut abzuschätzen.

[6] Verfügbar unter Onlineressource `http://www.speedtest.net/mini.php` (abgerufen am 3.2.2014)

[7] *Adobe Flash* ist eine Plattform für Anwendungen, die überwiegend im Webbrowser ausgeführt werden. Dazu wird ein Browser-Plugin benötigt, welches für viele aktuelle Webbrowser verfügbar ist. Ursprünglich zur besseren Unterstützung multimedialer Visualisierungen im Web-Umfeld entwickelt, wird *Flash* heute für sehr viele verschiedene Web-Anwendungen genutzt. Darunter befinden sich Spiele, Informationsssoftware, Bildverarbeitung und auch Benchmarks. (siehe Onlineressource `http://www.adobe.com/de/products/flashplayer.html` (abgerufen am 29.10.2013))

7.2.2 Authentifizierung

Der Broker nutzt zur Authentifizierung der Clients mittels Benutzername und Passwort einen dedizierten Dienst, welcher von der Authentifizierung zur Verfügung gestellt wird. Die Authentifizierung stellt die interne API-Methode `authenticate(...)` über die beschriebene verschlüsselte TCP-Socket-Verbindung zur Verfügung. Diese hat folgenden Prototyp:

```
bool authenticate(String user, String pass)
```

Die Methode liefert wahr oder falsch zurück. Auf die Implementierung einer umfassenden Nutzwerverwaltung, welche auch Bonitätsprüfung und monatliche Abrechnung umfassen würde, wurde hier verzichtet, da dieser Aspekt für die bedarfsgerechte Ressourcenauswahl für *Mobile Cloud-unterstützte Anwendungen* nur eine nachrangige Rolle spielt. Es wird davon ausgegangen, dass die konfigurierten Benutzer von einem Administrator direkt auf der Authentifizierung eingetragen werden. Somit ist auch keine externe API zur Authentifizierung implementiert. Zu Testzwecken sind berechtigte Benutzer und zugehörige Passwörter zeilenweise in einer Datei abgelegt.

7.2.3 Abrechnung/Log

Die Abrechnung besitzt ein sehr flexibles, nur intern verfügbares Interface, welches nur den *Sessionkey* zusammen mit einem benutzerspezifischen Logeintrag variabler Länge akzeptiert. Ein initialer Logeintrag wird vom Broker stets zum Beginn einer neuen Session übermittelt. Der größte Teil der Logdaten kommt jedoch von VMs, welche die Überwachungsdaten der Applikationen und das Ende der Sessions übermitteln. Die Logeinträge werden durch die VMs selbst gepuffert und nur gelegentlich zur Abrechnung übertragen. Durch diese wenigen großen Datentransfers kann die Übertragungsbandbreite bestmöglich ausgenutzt werden. Die Methode `writeLogEntry(...)` hat folgenden Prototyp:

```
bool writeLogEntry(String session_key, byte[] log_entry)
```

Zur bestmöglichen Unterstützung von benutzerdefinierten Logformaten werden in der Logdatenbank nur komplette Logdatensätze unter Angabe des *Sessionkey* und des Zeitstempels abgelegt. Diese können später unter Angabe des *Sessionkey* zu Debug- und Analysezwecken abgerufen werden. Eine spezielle Struktur der Logdatensätze wird nicht verlangt.

Für die prototypische Implementierung wurde für jeden stets eindeutigen *Sessionkey* eine eigene Datei angelegt, in der die übertragenen Logdaten unter Angabe des Zeitstempels abgespeichert wurden. Der Prozess der Rechnungsstellung selbst wird jedoch in dieser Arbeit nicht betrachtet, da der Fokus auf der Realisierung der Anwendung liegt.

Für Produktivimplementierungen sollten die Logdaten redundant und sicher abgelegt werden, da sich aus Ihnen auch die Daten für die Abrechnung der Nutzung ableiten. Moderne sogenannte NoSQL-Datenbanken[8] können zur Speicherung verwendet werden, deren Kapazität sehr groß und deren Schreibperformance für INSERT-Operationen sehr hoch ist. Zur

[8] NoSQL (englisch für *Not only SQL*) bezeichnet Datenbanken, die nicht dem relationalen Ansatz folgen. Der heute genutzte Begriff geht zurück auf das Jahr 2009, in dem ein Meeting zum Thema alternative Datenspeicher in San Francisco stattfand, welches diesen Term als Titel trug. Diese Datenspeicher benötigen keine festgelegten Tabellenschemata und versuchen, Joins zu vermeiden, sie skalieren dabei horizontal. Bekannte Implementierungen sind *Google BigTable* und *Amazon Dynamo*. Darüber hinaus gibt es eine Rei-

Auswertung werden zudem ausschließlich Abfragen der Form "`Select * From TABLE where SESSION=abcd0123;`" benötigt, welche keine Joins erfordern, die von gängigen NoSQL-Datenbanken wie MongoDB [MPH10] nicht unterstützt werden.

7.2.4 Compute-Ressource (VM)

Die VMs bieten als externe Schnittstelle das bereits in Kapitel 4.3.2 vorgestellte Pipelining-Protokoll zum Transport einzelner Tasks an. Da es sich hier um eine Eigenentwicklung handelt, wurde im Gegensatz zum Broker auf die Verwendung von Standardkomponenten bei der Implementierung des Servers verzichtet. Das Serverprogramm der VMs ist somit ohne weitere Abhängigkeiten als eigenständiges *Java*-Programm lauffähig.

Die externe API beschränkt sich hier auf einen einzigen TCP-Port, der zum Aufbau einer Pipelining-Verbindung von außen genutzt werden kann. Üblicherweise wird Port `8080` verwendet. Direkt nach dem Verbindungsaufbau wird zunächst der vom Broker-Server erhaltene *Sessionkey* vom Client zur VM übertragen. Dies wurden bei der Vorstellung des Pipelining-Protokolls in Kapitel 4.3.2 als Präambel bezeichnet. Damit authentifiziert sich der Client gegenüber der VM. Als Antwort empfängt der Client anschließend *OK* oder, falls der *Sessionkey* nicht verifiziert werden konnte, eine Fehlernachricht. Schlug die Authentifizierung fehl, wird die Verbindung von der VM anschließend sofort geschlossen. War die Authentifizierung erfolgreich, beginnt danach sofort das Pipelining-Protokoll und die VM wartet auf die Übermittlung des ersten Task. Diese externe Schnittstelle wird als verschlüsselte TCP-Socket-Verbindung genutzt.

Zusätzlich bietet die VM noch eine interne Schnittstelle an, die dem Broker dazu dient, den aktuellen *Sessionkey* für die nächste Clientanfrage an die VM zu übermitteln. Ihre Verwendung innerhalb des Lebenszyklus einer Client-Anfrage wurde in Kapitel 5.2.4 bereits erläutert. Sie hat folgenden Prototyp:

```
void sendeSessionkey(String session_key)
```

7.3 Implementierung der Middleware-Komponenten

Dieses Kapitel stellt nicht nur die Programmbibliotheken vor, die die Middleware zur Verfügung stellt, ihre Verwendung wird auch anhand von Beispielen demonstriert. Um den beispielhaften Programmcode besser von der bereits verwendeten Darstellungsweise der API-Methoden zu unterscheiden, wurde der *Java*-Code als solcher markiert. Die bereits vorgestellten Softwarepakete der *Java*-Programmbibliothek sind nochmals in Abbildung 7.3 als UML-Diagramm (*Unified Modeling Language*) visualisiert und es sind zusätzlich noch die wichtigsten Klassen innerhalb der Pakete dargestellt.

7.3.1 Implementierung der Taskverarbeitung

Die Verwendung des Pipelining-Protokolls zur effizienten Netzwerkkommunikation impliziert die Anordnung von Berechnungen als einzelne Tasks. Zur Unterstützung der Applikationsentwicklung wird eine Programmbibliothek vorgestellt, die das Pipelining implizit durchführt

he von Open Source-Implementierungen wie beispielsweise *CouchDB*, *Apache Cassandra* und *MongoDB*. (siehe [SF12])

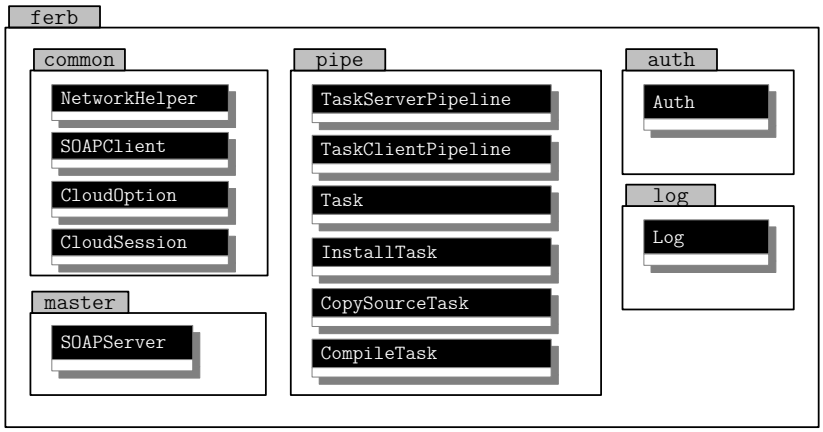


Abbildung 7.3: UML-Diagramm der verfügbaren Softwarepakete und deren wichtigste Klassen.

und die nötigen Funktionen zur Task-Programmierung bereitstellt. Die eigentlichen Datentransfers werden von der Programmbibliothek durchgeführt und ebenso wie das Marshalling und Unmarschalling von Task-Objekten vor der Applikation verborgen. Wie bereits in Kapitel 4 beschrieben, besteht ein Task bei der Übertragung via Netzwerk aus folgenden Teilen:

```
struct Task {
    String task_id;
    String task_function;
    byte[] input_data;
    byte[] output_data;
}
```

Beim Transfer via Netzwerk vom Client zur VM werden `task_id`, `task_function` und `input_data` übertragen. In die Gegenrichtung werden nur `task_id` und `output_data` übertragen. Anhand der `task_id` können auf dem Client ausgehende und eingehende Daten einander zugeordnet werden.

Im Allgemeinen ist davon auszugehen, dass die Ausführungsdauer der Tasks im Vergleich zur Datenübertragungszeit hoch ist, da sonst eine entfernte Ausführung nicht sinnvoll wäre. Somit ist es notwendig, Puffer zu verwenden, die Tasks zwischenspeichern können, damit während der Ausführung weitere Tasks übertragen werden können. Die Größe der Puffer wird begrenzt durch die Überlegung, dass auf einem Client üblicherweise nur begrenzt viel Arbeitsspeicher zur Verfügung steht. Somit sollten die Puffer auf dem Client nicht größer ausfallen als nötig. Außerdem kann eine kabellose Netzwerkverbindung auch unvermittelt getrennt werden. Da jedoch mobile Datenübertragungen oft bezahlt werden müssen, sollten stets nicht mehr Tasks zur VM übertragen werden, also dort auch momentan verarbeitet werden können. Im Falle eines Verbindungsabbruchs müssten sonst eventuell Tasks erneut übertragen werden und deren Übertragungsdatengröße würde erneut abgerechnet.

Task und Task-Client-API

Die *Java*-API, die die Pipelining-Funktionalität und die dazu notwendige Taskverarbeitung auf einem Client nutzbar macht, steht im Softwarepaket `ferb.pipe` zur Verfügung. Die abstrakte Klasse `Task` implementiert die grundlegenden Taskfunktionen. Von `Task` abgeleitete

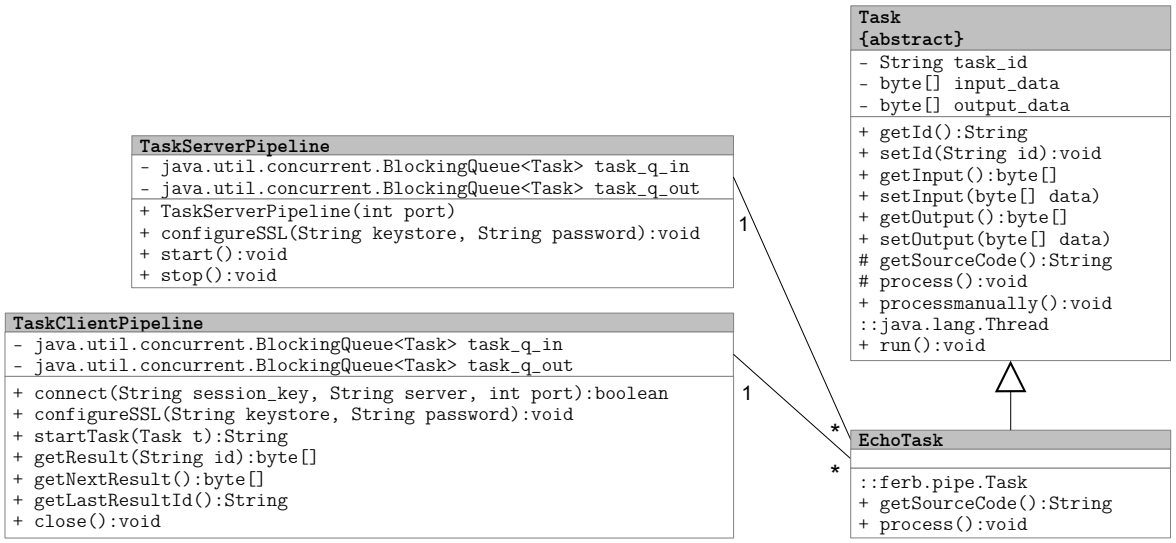


Abbildung 7.4: UML-Diagramm der Klassen aus dem Softwarepaket `ferb.pipe` zur Implementierung und Verarbeitung eines beispielhaften `EchoTask`.

Klassen implementieren die eigentlichen Berechnungstasks. Zudem gibt es für die VMs die Klasse `TaskServerPipeline` als Implementierung der Taskverarbeitung auf Serverseite. Für die Clients gibt es die Klasse `TaskClientPipeline`. Die UML Repräsentation der Klassen ist in Abbildung 7.4 zu sehen.

Die abstrakte Klasse `Task` ist von `java.lang.Thread` abgeleitet und kann somit nebenläufig ausgeführt werden. Dies ermöglicht die parallele Verarbeitung von Tasks. Die Klasse `Task` wird auf Client- und auf Serverseite zur Repräsentation der Berechnungstasks benutzt. Sie implementiert die Methoden zur Übergabe der Eingabedaten `setInput(byte[] data)` und zum Abholen der Ausgaben `getOutput()`. Die eigentliche Taskfunktion `byte[] process(byte[] input)` ist leer und muss von einer abgeleiteten Taskklasse implementiert werden. Eine von `Task` abgeleitete Klasse, die die Methode `byte[] process(byte[] input)` implementiert, kann dann als eigenständiger Thread ausgeführt werden. Über die von `java.lang.Thread` geerbte Methode `start()` wird der Task gestartet.

Der Task kann ebenso synchron vom aktuellen Thread abgearbeitet werden. Dazu wird statt der Methode `start()` die Methode `processmanually()` aufgerufen, die solange blockiert, bis die Methode `process(...)` des Tasks ausgeführt wurde. Diese Methode wird benutzt, wenn der Task beispielsweise wegen einer fehlenden Kommunikationsverbindung nicht entfernt ausgeführt werden kann, sondern lokal ausgeführt werden muss.

Um einen Task auf einer VM auszuführen, muss dessen Taskklasse bekannt sein. Vor der Übertragung eines Tasks muss also zunächst der Quellcode der Taskklasse an die VM gesendet werden. Dazu wird die Methode `getSourceCode()` in der abgeleiteten Taskklasse überschrieben. Sie liefert den Quellcode der Taskklasse zurück. Exemplarisch ist nachfolgend unten eine Klasse `EchoTask` dargestellt, die einen Task implementiert, der eingehende Daten unverändert zurücksendet. Dazu wird `EchoTask` von `ferb.pipe.Task` abgeleitet. Zusätzlich wird die Methode `process(...)` überschrieben. Sie implementiert die Funktionalität des `EchoTask` und gibt die übergebenen Daten direkt zurück. Der eigene Quellcode der Klasse ist in der Konstante `code` abgelegt und wird durch `getSourceCode()` zurückgeliefert. Das nachfolgende

Codebeispiel zur Benutzung der Taskklasse in der erstellten Middleware stellt dabei schon die vollständige Klasse dar, die so durch die Taskverarbeitung der Middleware ausgeführt werden kann, was im nachfolgenden Beispiel dargestellt wird. Für komplexere Funktionalität muss die Methode `process(...)` angepasst werden und der Quellcode erneut in der Konstante `code` abgelegt werden.

```
// Java Code Beispiel
import ferb.pipe.Task;

public class EchoTask extends Task {

   private static final String code = "import ferb.pipe.Task;"
                                    + "public class EchoTask extends Task {"
                                    + " @Override public byte[] process(byte[] input) {"
                                    + " return input; }}}";

   @Override
   public String getSourceCode() {
      return code;
   }

   @Override
   public byte[] process(byte[] input) {
      return input;
   }
}
```

Die Klasse `TaskClientPipeline` wird bereitgestellt, um die entfernte Ausführung von Tasks zu ermöglichen. Sie implementiert die Methoden `connect(String session_key, String server, int port)`, `configureSSL(String truststore, String password)` und `close()`, die der Verwaltung der Verbindung vom Client zur VM dienen. Weitere Methoden dienen der eigentlichen Taskverarbeitung. Nachdem unter Angabe des *Sessionkey* über `connect(...)` eine Verbindung aufgebaut wurde, können die Taskverarbeitungsmethoden bis zum Aufruf von `close()` verwendet werden. Durch Aufruf der Methode `close()` oder durch sonstiges Abbrechen der TCP-Verbindung wird die Abrechnung auf Serverseite beendet und der *Sessionkey* invalidiert. Der clientseitige Quellcode zur Verwendung der `TaskClientPipeline` zur Ausführung eines `EchoTask` ist nachfolgend unten dargestellt:

```
//Java Code Beispiel

ferb.pipe.TaskClientPipeline tcp = new ferb.pipe.TaskClientPipeline();
// SSL konfigurieren
tcp.configureSSL("./resources/SSLClientTrust", "password");
// Zur vm verbinden, weiter nur wenn "OK", sonst Abbruch
if ( tcp.connect(session_key, server_endpoint, 8080).equals("OK"))
   // Task erstellen
   EchoTask ct = new EchoTask();
   // Task mit Nutzdaten füllen
   ct.setInput(custom_data);
   // Task absenden
   tcp.startTask(ct);
   // Auf das Ergebnis der Verarbeitung warten
   try {
      byte[] result = tcp.getNextResult();
   } catch (UnknownTaskFuncException e) {
      // Falls Task auf Serverseite unbekannt wird Exception ausgelöst
   }
   tcp.close();
```

Die interne Verarbeitung der Tasks durch die Klasse `TaskClientPipeline` ist in Abbil-

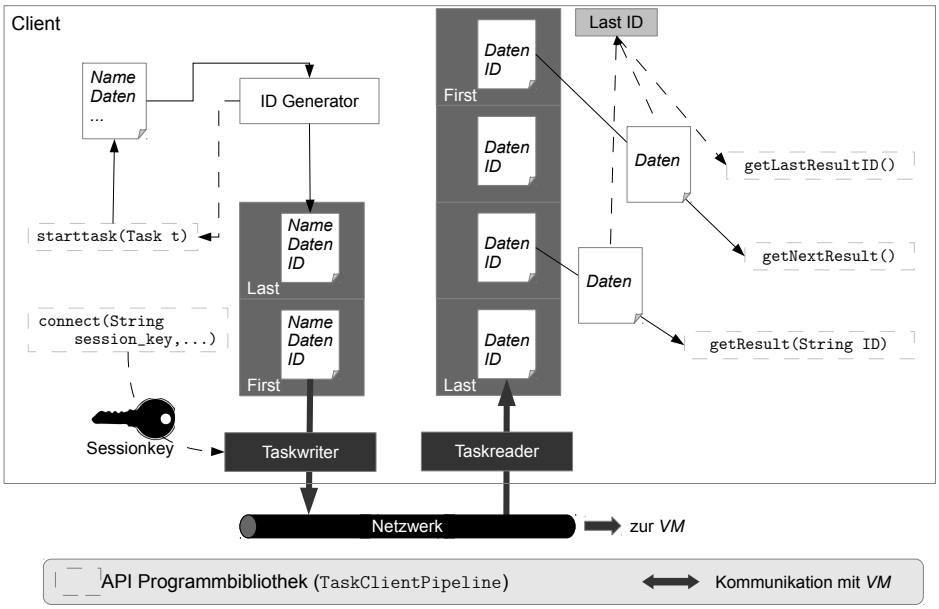


Abbildung 7.5: Visualisierung der Taskverarbeitung in der Klasse `TaskClientPipeline` auf dem Client.

dung 7.5 dargestellt. Die Klasse implementiert die Methode `String startTask(Task t)`, die einen Task in die Warteschlange für ausgehende Tasks einfügt und blockiert, falls diese Warteschlange bereits voll ist. Die Warteschlange für ausgehende Tasks hat zwei Plätze. Die Warteschlange sollte zwar so klein wie möglich sein, jedoch hat sich herausgestellt, dass die Netzwerkschicht bei nur einem Platz manchmal nicht schnell genug mit Daten versorgt werden kann. Nach dem erfolgreichen Einfügen liefert die Methode eine eindeutige Task-ID zurück, die später zum Abfragen des Ergebnisses der Taskausführung verwendet werden kann. Verarbeitete Tasks werden von der VM zurück zum Client übertragen und dort in eine unbeschränkt große Warteschlange eingefügt. Da Ergebnisse eventuell nicht direkt entnommen werden, würde eine beschränkte Größe hier eventuell verhindern, dass alle fertigen Ergebnisse zwischengespeichert werden können. Es ist also Aufgabe der Applikation, die fertigen Ergebnisse auch zeitnah zu entnehmen. Ergebnisse von Tasksausführungen können am Client über die Methode `byte[] getNextResult()` abgefragt werden. Diese Methode liefert die Ausgabe des nächsten Tasks in der Warteschlange zurück und blockiert nur, falls keine Tasks in der Warteschlange vorhanden sind. Wartet man auf das Ergebnis eines bestimmten Tasks, so kann man die Methode `byte[] getResult(String taskID)` aufrufen. Sie kehrt erst zurück, wenn das Ergebnis des Tasks mit der angegebenen ID in der Warteschlange vorliegt. Diese Methode sollte jedoch nur benutzt werden, wenn sichergestellt ist, dass der Task, auf den gewartet wird, nicht vorher schon aus der Warteschlange abgeholt wurde. Sollte dies nicht sichergestellt sein, so kann die Methode dauerhaft blockieren, wenn der Task, auf den gewartet wird, schon entnommen wurde. Um diese Situation zu vermeiden, kann man durch Aufruf der Methode `String getLastResultID()` die ID des zuletzt entnommenen Tasks abfragen. Durch das asynchrone Ausführen von Tasks ist es nicht direkt möglich zu erkennen, ob ein Task, der `startTask(...)` übergeben wurde, auch ordnungsgemäß ausgeführt werden konnte. Im Fehlerfalle werfen erst die Methoden `getNextResult()` und `getResult(String`

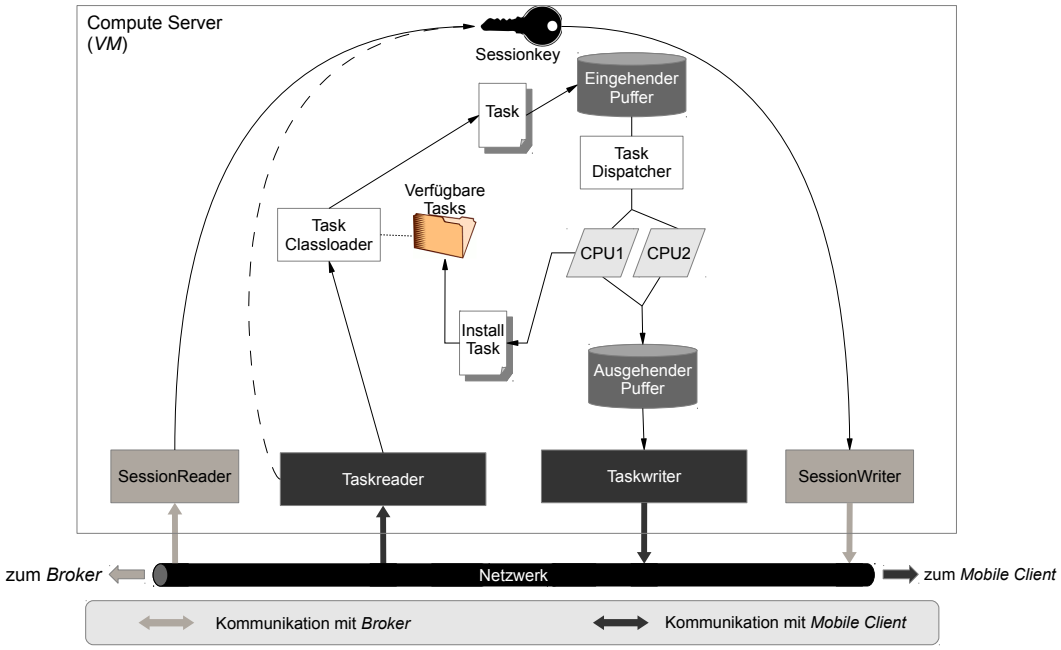


Abbildung 7.6: Visualisierung der Taskverarbeitung auf der VM durch die Klasse `TaskServerPipeline`.

`taskID`) eine Exception vom Typ `UnknownTaskFuncException`. Anhand der ID kann jedoch festgestellt werden, welcher Task den Fehler verursacht hat.

Task-Server-API

Der Task-Server auf der VM wird von der Klasse `TaskServerPipeline` implementiert. Sie akzeptiert eingehende Verbindungen über die externe API von jeweils nur einem Client gleichzeitig. Dazu wird nur ein einziger TCP-Port benötigt, der dann in der Firewall des IaaS-Anbieters für eingehende Verbindungen aus dem Internet geöffnet werden muss. Zusätzlich akzeptiert die Klasse `TaskServerPipeline` auf der VM über die interne API Verbindungen vom Broker. Der Broker teilt der VM während des Allokationsvorgangs eines Clients den zu erwartenden *Sessionkey* mit. Baut der Client dann eine Verbindung zur VM auf, so wird der übertragene *Sessionkey* des Client mit dem zuvor vom Broker erhaltenen *Sessionkey* verglichen. Abbildung 7.6 visualisiert die Interna der Klasse `TaskServerPipeline`.

Die Klasse verwaltet 2 Warteschlangen (`java.util.concurrent.BlockingQueue<Task>`), eine für eingehende und eine für ausgehende Tasks. Deren Größe entspricht der Anzahl verfügbarer Prozessoren auf der VM. Dies entspricht der Forderung, dass stets nicht mehr Tasks transferiert werden, als auch verarbeitet werden können. Sind die Tasklaufzeiten lang genug, dann ist stets ein Platz für das Ergebnis und für einen neuen Task zur Verarbeitung in den Warteschlangen bereit. Tasks werden, da sie von der Klasse `java.lang.Thread` abgeleitet sind, intern von einem Threadpool abgearbeitet, dessen Größe der Anzahl verfügbarer logischer Prozessoren des Systems entspricht. Die Verwendung eines Threadpools ist sinnvoll bei der Verarbeitung vieler Tasks kurzer Laufzeit, da zur Ausführung in einem Threadpool bestehende Threads wiederverwendet werden und die Threaderzeugung nicht für jeden Task als Overhead anfällt. *Java* bietet dazu das Interface `java.util.concurrent.ExecutorService`

an, welches von verschiedenen Threadpool-Varianten implementiert wird. Zunächst wird für eingehende Tasks eine Taskklasse des benötigten Typs aus einem Archiv verfügbarer Taskklassen instanziiert. Es können sowohl von der Middleware bereitgestellte Taskklassen als auch zur Laufzeit installierte Taskklassen geladen werden. Sollte das Laden fehlschlagen, weil keine passende Class-Datei gefunden werden konnte, so wird ein `ErrorTask` in die ausgehende Warteschlange eingefügt, der den Fehler dem Client rückmeldet. War die Instanziierung hingegen erfolgreich, so wird der Task zur Bearbeitung an den Threadpool übergeben. Das Ergebnis der Ausführung wird danach direkt in die Warteschlange für ausgehende Tasks eingefügt. Nach Beendigung einer Verbindung sendet die Klasse `TaskServerPipeline` selbstständig eine Notiz an den Broker, dass die Session beendet ist und die VM für neue Client-Anfragen bereitsteht. Zudem werden die gesammelten Logdaten versendet und das Verzeichnis für benutzerdefinierte Class-Dateien geleert. Somit kann der nächste Benutzer nicht versehentlich Tasks des Vorgängers instanziieren.

Die Klasse `TaskServerPipeline` enthält zusätzlich Methoden, um den Taskserver zu starten und zu konfigurieren. Ein Minimalbeispiel zur Verwendung ist nachfolgend dargestellt.

```
// Java Code Beispiel
ferb.pipe.TaskServerPipeline tsp = new ferb.pipe.TaskServerPipeline(8080);
tsp.configureSSL("./resources/SSLServerStore","password");
tsp.start();
```

Der Konstruktor verlangt als Parameter den zu verwendenden externen TCP-Port. Die Methode `configuteSSL(...)` konfiguriert die Zertifikatdatei für die SSL-Verschlüsselung und das zum Öffnen notwendige Passwort. Gestartet wird der Server durch Aufruf der blockierenden Methode `start()`. Die Methode `stop()` fährt den Server herunter und führt dazu, dass `start()` zurückkehrt.

7.3.2 Code-Installation zur Laufzeit

In Kapitel 5.1 wurde bereits festgestellt, dass es zur Senkung der Kosten notwendig ist, VM-Instanzen für verschiedene Nutzeranfragen wiederzuverwenden. Da jedoch nicht bekannt ist, welche Applikationen die Nutzer tatsächlich ausführen wollen, sind VM-Ressourcen rekonfigurierbar. Jeder Nutzer installiert den benötigten Programmcode zur Laufzeit selbst. Um dies im Rahmen der Taskverarbeitung zu ermöglichen, stehen zur Installation von Programmcode verschiedene elementare Taskfunktionen zur Verfügung. Diese dienen dazu, Quell- oder Programmcode auf die VM zu kopieren und dort gegebenenfalls auch zu kompilieren. Das Laden der einzelnen Tasks erfolgt dann durch einen speziellen *Classloader*. Die vordefinierten Taskklassen sind ebenfalls im Softwarepaket `ferb.pipe` zu finden.

InstallTask

Um Tasks auszuführen, muss die Taskklasse vom *Classloader* auf Serverseite gefunden und geladen werden können. Um neue Tasks zu installieren, kann ein vordefinierter `InstallTask` verwendet werden. Er wird auf Clientseite benutzt, indem man den Quellcode des neuen Tasks als Eingabedaten des `InstallTask` angibt und den Task dann absendet. Hierbei sollte man die synchrone und ordnungsgemäße Bearbeitung des `InstallTask` abwarten, bevor man weitere Tasks zur Verarbeitung übersendet. Das nachfolgende Beispiel verdeutlicht die Verwendung zur Installation des bereits beschriebenen `EchoTask`.

```
// Java Code Beispiel
ferb.pipe.TaskClientPipeline tcp = new ferb.pipe.TaskClientPipeline();
tcp.configureSSL("./resources/SSLClientTrust", "password");
tcp.connect(session_key, server_endpoint, 8080);
// Task installation
ferb.pipe.InstallTask it = new ferb.pipe.InstallTask();
it.setInput(new EchoTask().getSourceCode().getBytes()); // java.lang.String.getBytes()
// Install Task abschicken
String id = tcp.startTask(it);
// Warten, dass der Install Task ordnungsgemäß abgefertigt wurde
System.out.println(new String(tcp.getResult(id))); //Fehlerbehandlung wurde weggelassen
// Nun kann der Task problemlos verwendet werden
EchoTask ct = new EchoTask();
ct.setInput(custom_data);
tcp.startTask(ct);
byte[] result = tcp.getNextResult(); //Fehlerbehandlung wurde weggelassen
tcp.close();
```

Auf Serverseite wird der übergebene Quellcode in einer Datei abgespeichert und anschließend kompiliert. Es ist darauf zu achten, dass die einzige Klasse innerhalb der Quellcodedatei, die als `public` markiert ist, von `ferb.pipe.Task` abgeleitet ist. Es können keinerlei Bibliotheken außer der *Java*-Systembibliothek verwendet werden, wenn der Quellcode als `InstallTask` installiert wird. Aller benötigter Quellcode muss sich innerhalb einer *Java*-Datei befinden, die so kompilierbar ist. Als Ergebnis des `InstallTask` wird dem Client die Debugausgabe des Kompilierlaufs zurückgeliefert. Somit kann der Entwickler im Fehlerfalle erkennen, warum der Quellcode nicht kompiliert werden konnte.

Classloader

Wurde ein Task inklusive seiner Nutzdaten und der zu verwendenden Taskfunktion zum Server übertragen, so muss der Server zunächst die Taskklasse laden, die als Implementierung der übersendeten Taskfunktion konfiguriert wurde, und an das instanziierte Task-Objekt die Eingabedaten anhängen. Ein spezieller *Classloader* versucht, die Klasse zu laden, deren Name der übersendeten Taskfunktion entspricht. Um beispielsweise die Klasse `EchoTask` zu laden, muss als Taskfunktion „EchoTask“ übergeben worden sein. Der *Classloader* versucht dann, die Datei `EchoTask.class` zu finden und zu laden. Ist dies nicht möglich, wird als Ergebnis der Taskverarbeitung ein Fehler an den Client gesendet, der dann dort als Ergebnis eine Exception vom Typ `UnknownTaskFuncException` auslöst. Alle verfügbaren Klassen, die beispielsweise durch einen `InstallTask` auf dem Server installiert wurden, werden in ein dafür vorgesehenes Arbeitsverzeichnis kopiert. Aus diesem Arbeitsverzeichnis versucht auch der *Classloader*, die entsprechenden Klassen zu laden.

CopySourceTask und CompileTask

Umfangreichere Programmmodule können mit Hilfe der durch die Middleware bereitgestellten Taskklassen `CopySourceTask` und `CompileTask` vom Client zur VM übertragen und dort gegebenenfalls kompiliert werden. Manchmal kann der Programmcode eines Tasks aus mehreren Klassen bestehen, die mit einem einfachen `InstallTask` nicht installiert werden können. Bei der Verwendung des `InstallTask` müssen alle benötigten Klassen in einer einzigen CompilationUnit (Quellcodedatei) stehen. Dies kann unpraktisch sein und zerstört eventuell die Pakethierarchie, wenn bereits vorhandener Quellcode entfernt ausgeführt werden soll. Um dieses Problem zu umgehen, können auch umfangreichere Pakete von Quellcode- oder Programmcodedateien im Verbund zur VM übertragen werden.

Ein `CopySourceTask` dient dazu, Dateien zur VM zu übertragen, und akzeptiert als Eingabedaten ein Byte-Array, welches ein durch den LZMA-Algorithmus[9] gepacktes Tar-Archiv von Dateien repräsentiert. Nach der Übertragung eines `CopySourceTask` wird auf Serverseite die zugehörige Taskfunktion ausgeführt, welche das Archiv entpackt und die einzelnen Dateien in ihrer Paketstruktur in das Arbeitsverzeichnis extrahiert.

Wurden Quellcodedateien übertragen, kann zusätzlich ein `CompileTask` vom Client zur VM gesendet werden. Die Taskfunktion eines `CompileTask` kompiliert alle `.java`-Dateien im Arbeitsverzeichnis der VM und verlangt keine Übergabeparameter. Dabei können keine Bibliotheken im `.jar`-Format berücksichtigt werden. Der gesamte Programmcode muss als Quellcode vorliegen. Da dies nicht immer gegeben ist und da die Kompilierung teilweise recht lange dauern kann (bis zu 40 s bei größeren Projekten), ist es auch möglich, einen `CopySourceTask` zu benutzen, um alle benötigten vorkompilierten `.class`-Dateien direkt zu kopieren. Dabei müssen diese ebenfalls in ein LZMA-komprimiertes Tar-Archiv gepackt sein. Hierbei kann es jedoch dazu kommen, dass die *Java*-Version auf Serverseite die übertragenen Dateien aufgrund inkompatibler Bytecode-Versionen nicht verarbeiten kann. In diesem Fall müssen die Class-Dateien für die entsprechende *Java*-Version neu kompiliert werden.

Die Übertragung von Quellcode via `CopySourceTask` und die Möglichkeit der Kompilierung direkt auf der VM via `InstallTask` oder `CompileTask` sind vor allem zur Entwicklung von *Mobilen Cloud-unterstützten Anwendungen* hilfreich, da kleinere Code-Anpassungen schnell umgesetzt werden können. Ist der Quellcode hinreichend stabil, so ist für die produktive Phase die Installation als Bytecode-Archiv eventuell günstiger. Dies trifft insbesondere für umfangreiche Archive zu, da durch die LZMA-Komprimierung die Dateigröße zur Übertragung zur VM erheblich gesenkt wird. Die Besonderheit der schnellen Dekomprimierung von LZMA-Archiven macht sich ebenfalls positiv bemerkbar. Dem gegenüber spielt die geringe Performance beim Komprimieren des Tar-Archivs kaum eine Rolle, da üblicherweise nur einmal verpackt wird.

7.3.3 Implementierung von Applikationen auf Basis des Task Konzepts

Die Verwendung des hier beschriebenen Taskkonzepts setzt zunächst voraus, dass sich eine Applikation so transformieren lässt, dass Teile der Applikation als abgeschlossene Tasks ausführbar sind. Besonders gut lässt sich das Konzept anwenden, wenn die Tasks unabhängig sind und deshalb asynchron, also nebenläufig, ausgeführt werden können. Ein verbreitetes Konzept in der verteilten Programmierung ist jedoch auch die Verwendung entfernter Funktionsaufrufe wie es beispielsweise von *Java* RMI realisiert wird (siehe Kapitel 4.2.4). Mit Hilfe der bereitgestellten Programmbibliothek ist es in Grenzen auch möglich, dieses Konzept des synchronen entfernten Funktionsaufrufs via Adapter bereitzustellen. Nachfolgend werden Beispiele für die beiden Implementierungsvarianten der asynchronen Taskverarbeitung und des entfernten Funktionsaufrufs vorgestellt.

Asynchrone Taskverarbeitung

Die asynchrone Taskverarbeitung wird anhand der Auswertung einer Folge von Kamerabildern auf einem *Android*-Gerät demonstriert. Zur asynchronen Verarbeitung von Tasks ist es

[9] Der Lempel-Ziv-Markow-Algorithmus (LZMA) ist ein freier Datenkompressionsalgorithmus, der von Igor Wiktorowitsch Pawlow seit 1998 entwickelt wird. Er liefert im Vergleich zu anderen Verfahren eine bessere Kompressionsrate und erreicht zudem beim Entpacken eine hohe Geschwindigkeit. Der Algorithmus ist im Quellcode für verschiedene Programmiersprachen, darunter auch *Java*, verfügbar. (siehe [SM10, S.411 ff.])

sinnvoll, wenn das Absetzen der Tasks und das Aufsammeln und Auswerten der Ergebnisse von unterschiedlichen Threads auf dem Client erledigt werden. Das nachfolgende Beispiel verdeutlicht dies. Links ist das Starten der Tasks und rechts die Verarbeitung der Ergebnisse dargestellt. Im Beispiel nicht dargestellt ist die initiale Erzeugung des `TaskClientPipeline`-Objekts (`tcp`), der Aufbau der Verbindung zur VM und die Installation des Programmcodes des verwendeten `PicTask`.

```
// Java Code Beispiel
@Override
public void onPreviewFrame(byte[] data,
                           Camera camera)
{
    ...
    PicTask pt = null;
    synchronized(Application.this){
        if(pending < 3){
            pt = new PicTask();
            pt.setInput(data);
            pending++;
        }
    }
    if (pt != null) tcp.startTask(pt);
    ...
}
```

```
// Java Code Beispiel
@Override
public void run()
{
    ...
   while(running){
      byte[] result = tcp.getNextResult();
      synchronized(Application.this){
         pending--;
      }
      show(result);
   }
    ...
}
```

Üblicherweise wird für einen bestimmten Sensor eine Callback-Funktion registriert, die immer dann vom Laufzeitsystem aufgerufen wird, wenn neue Sensordaten verfügbar sind. Je häufiger die Methode aufgerufen wird, desto höher ist die zeitliche Auflösung des Sensors. Bei Kamerabildern spricht man auch von Bildverarbeitungsrate. Eine solche Callback-Funktion (`onPreviewFrame(...)`) ist im Codebeispiel auf der vorherigen Seite links dargestellt. Rechts hingegen ist ein Codefragment eines separaten Threads zu sehen, der die verarbeiteten Daten von der VM in einer Schleife entgegennimmt und zur Anzeige bringt. `tcp` repräsentiert hier in beiden Fällen das gleiche `TaskClientPipeline`-Objekt.

Beide Methoden werden innerhalb einer Klasse `Application` definiert und benutzen das Objekt als Synchronisationsobjekt. Dabei wird der Zugriff auf eine Variable `pending` synchronisiert, die dafür sorgt, dass nicht mehr als drei Bilder gleichzeitig auf der VM verarbeitet werden. Gegebenenfalls werden Bilder verworfen, wenn noch zu viele Ergebnisse ausstehen. Zu beachten ist auch, dass nicht davon ausgegangen werden kann, dass die Reihenfolge der abgesendeten Tasks im Codebeispiel auf der vorherigen Seite links mit der Reihenfolge der empfangenen Tasks rechts identisch ist, da die Verarbeitungszeiten aufeinander folgender Tasks eventuell ungleich lang sind. Will man die Reihenfolge synchronisieren, müssten die Task-IDs ausgewertet werden, was jedoch mehr Speicherplatz aufgrund der Zwischenspeicherung von Ergebnissen seitens des Clients erfordern könnte.

Synchrone Task Verarbeitung

Die synchrone Task-Verarbeitung entspricht einer Art *Remote Object* Emulation. Dabei wird der synchrone Methodenaufruf auf einem entfernten Objekt über die Klasse `TaskClientPipeline` modelliert. Es wird nachfolgend beschrieben, wie aus einem bestehenden *Java* RMI Interface eine Task-Pipelining-kompatible Variante zur Ausführung erzeugt werden kann. Dazu werden auf Client- und Serverseite Adapterklassen eingesetzt, die die notwendigen Verhaltenskonvertierungen übernehmen. Es wird von folgendem *Java* RMI Interface (`IRemoteAPI`) und dessen Implementierung (`RemoteAPIImplementation`) ausgegangen:

```java
// Java Code Beispiel
public interface IRemoteAPI extends java.rmi.Remote {

    public String getInfo() throws java.rmi.RemoteException;

    public boolean setInfo(String info) throws java.rmi.RemoteException;
}
```

```java
// Java Code Beispiel
public class RemoteAPIImplementation implements IRemoteAPI {

    private String info;

    public String getInfo() throws java.rmi.RemoteException{
        return info;
    }
    ...
}
```

Auf Clientseite wird eine Adapterklasse benötigt, welche das *Java* RMI Interface implementiert, die (synchronen) Funktionsaufrufe entgegennimmt und die übergebenen Parameter via Task-Pipelining zur VM überträgt. Anschließend wird das Ergebnis der entfernten Verarbeitung als Rückgabewert des Funktionsaufrufs zurückgeliefert. Zur Realisierung wird für jede Methode im RMI Interface ein Hilfstask erstellt. Für die Methode `getInfo()` aus dem Interface `IRemoteAPI` wird beispielsweise eine Klasse `GetInfoTask` entwickelt. Die Implementierung der Adapterklasse `ClientRemoteAdapter` auf dem Client, die das Interface `IRemoteAPI` implementiert, ist nachfolgend dargestellt. Sie verwendet intern eine Instanz der Klasse `GetInfoTask`, wenn die Methode `getInfo()` aufgerufen wird.

```java
// Java Code Beispiel
public class ClientRemoteAdapter implements IRemoteAPI {
   @Override
   public String getInfo(){
       // Eine Instanz der Klasse TaskClientPipeline (tcp) wurde bereits erstellt
       GetInfoTask git=new GetInfoTask();
       String id = tcp.startTask(git);
       String result = null;
       try {
         result = new String(tcp.getResult(id)); // blockiert bis Task id verfügbar
       }
       catch (UnknownTaskFuncException e) { }
       return result;
   }
   ...
}
```

Der Aufruf `tcp.getResult(id)` blockiert, bis das Ergebnis der Taskverarbeitung vorliegt. Dies entspricht einem synchronen Methodenaufruf.

Auf Serverseite wird die Taskfunktion der Klasse `GetInfoTask` ausgeführt. Dabei wird die ursprüngliche Methode der RMI-Implementierung `RemoteAPIImplementation` aufgerufen. Auf Serverseite wird Adaptercode benötigt, um die Beziehung zwischen einer Objektinstanz und dem zugehörigen Methodenaufruf herzustellen. Bildet man das RMI-Verhalten nach, so erhält man stets die gleiche Objektinstanz bei Anfragen an dasselbe RMI-Objekt seitens des Client. Die Instanz lebt, solange die Pipelining-Verbindung besteht. Die nachfolgende Implementierung stellt eine Variante dar, die pro Client nur ein Serverobjekt der Klasse `RemoteAPIImplementation` erstellt, welches als *Singleton* realisiert ist. Die Implementierung der Klasse `GetInfoTask` ist nachfolgend beispielhaft dargestellt.

```
// Java Code Beispiel
public class GetInfoTask extends ferb.pipe.Task {
    @Override
    protected byte[] process(byte[] input) {
        // Abbildung des Methodenaufrufs auf ein serverseitiges Objekt als Singleton
        RemoteAPIImplementation rapi = RemoteAPIImplementation.getInstance();
        String info = rapi.getInfo();
        return info.getBytes();
   }
}
```

Bei dieser Art der Modellierung sind jedoch auch Einschränkungen zu beachten: Die Taskverarbeitung lässt nur Byte-Arrays als Datentypen zu. Andersartige Parameter müssen konvertiert werden. Komplexe Strukturen müssen wie bei *Java* RMI serialisiert werden. Mehrere Parameter werden zu einem einzigen Übergabeobjekt zusammengefasst und via Objektserialisierung in ein Byte-Array umgewandelt. Darüber hinaus muss über die Relation zwischen Client-Instanz und Objekt-Instanz auf Serverseite nachgedacht werden. Im Beispiel wurde stets auf die gleiche serverseitige Objektinstanz abgebildet. Auch eine statische Methode oder eine immer wieder neue Instanz könnte man realisieren, je nach Intention der Applikation.

7.3.4 Debugging und Analyse von Cloud-unterstützten Applikationen

Die Erstellung von *Mobilen Cloud-unterstützten Anwendungen* erfordert nicht nur die Implementierung der Applikation selbst, sondern auch das Messen wichtiger Parameter der Applikation wie beispielsweise der Performance in Abhängigkeit des verwendeten VM-Typs. Dies ist notwendig, um eine bedarfsgerechte Auswahl des VM-Typs zur Laufzeit zu ermöglichen. Nachfolgend werden die Debug-Möglichkeiten beschrieben, die die Middleware zur Verfügung stellt, um die notwendigen Informationen zur Performancevorhersage zu erhalten und im System anzuwenden.

Auf den VMs wird für jeden ausgeführten Task automatisch die übertragene Datenmenge (Eingabe und Ausgabe) und die jeweilige Bearbeitungszeit als Logdaten erhoben und zur Komponente Abrechnung gesendet. Im Paket `ferb.log` steht zusätzlich die statische Methode der Klasse `Log` (`Log.LOG(String logstring)`) zur Verfügung, die während der Taskausführung verwendet werden kann, um benutzerdefinierte Loginformationen abzuspeichern. Diese benutzerdefinierten Loginformationen werden ebenfalls an die Komponente Abrechnung gesendet. Der zur Abspeicherung notwendige aktuelle *Sessionkey* wird durch die VM automatisch ergänzt.

Aus den abgespeicherten Logdaten kann später die Laufzeitfunktion *lauf(...)* zur Vorhersage der Laufzeit konstruiert werden. Um die in Kapitel 6.1.1 eingeführte Laufzeitfunktion *lauf(...)* zur Beschreibung des Verhaltens der Applikation bereitstellen zu können, führt man die *Mobile Cloud-unterstützte Anwendung* zunächst aus, ohne vorher ein Applikationsprofil auf dem Broker zu installieren. Dazu testet man die Applikationsperformance unter Verwendung verschiedener Eingaben und VM-Typen. Die VM erfasst im sekündlichen Abstand auch die aktuelle Serverauslastung, die einen Anhaltspunkt für die erreichte Effizienz liefert. Die Logausgabezeile für die automatische Erfassung eines Task (nicht die Zeile mit der CPU-Auslastung) enthält die Uhrzeit, den verwendeten VM-Typ, die Task-ID, den Name des Task, die Größe der eingehenden und ausgehenden Daten und abschließend die Laufzeit in Millisekunden. Ein entsprechender Ausschnitt einer Logdatei hat folgende Gestalt.

```
13:01:50 PM c1.medium CPUload 4.0
13:01:51 PM c1.medium CPUload 13.9
13:01:52 PM c1.medium task2 CopySourceTask in 203521 out 1 416ms
13:01:53 PM c1.medium CPUload 11.9
13:01:54 PM c1.medium CPUload 37.8
13:01:56 PM c1.medium CPUload 2.0
13:01:57 PM c1.medium CPUload 105.3
13:01:59 PM c1.medium CPUload 93.3
13:02:00 PM c1.medium CPUload 103.3
13:02:02 PM c1.medium CPUload 119.3
13:02:03 PM c1.medium CPUload 77.5
13:02:05 PM c1.medium CPUload 173.5
13:02:06 PM c1.medium task3 CompileTask in 1 out 1 13574ms
13:02:07 PM c1.medium CPUload 35.8
13:02:08 PM c1.medium CPUload 152.8
13:02:09 PM c1.medium task4 PrepareBucketRenderingTask in 2010198 out 4 2146ms
13:02:09 PM c1.medium task5 StartRenderingTask in 1 out 4 1ms
13:02:10 PM c1.medium CPUload 186.7
13:02:11 PM c1.medium CPUload 198.6
13:02:13 PM c1.medium CPUload 198.6
13:02:15 PM c1.medium CPUload 192.9
13:02:15 PM c1.medium task6 GetNextBucketTask in 1 out 1363 6385ms
13:02:16 PM c1.medium CPUload 194.6
```

Aus diesen Informationen lassen sich anschließend bereits Auswertungen bezüglich der Performance der Applikation erstellen. Die Loginformationen müssen dazu je nach Applikation entsprechend ausgewertet werden. Dies erfolgt nicht automatisch, da jede Applikation eventuell andere Abspeicherungsformate der relevanten Daten benötigt. Ein applikationsspezifischer Logeintrag für die Software *Sunflow*, die später in Kapitel 8 als Beispielapplikation zur Bewertung der Middleware benutzt wird, hat folgende Gestalt.

```
08:27:26 AM m1.small SCENE  info  : Scene stats:
08:27:26 AM m1.small SCENE  info  :   * Infinite instances:  1
08:27:26 AM m1.small SCENE  info  :   * Instances:           875
08:27:26 AM m1.small SCENE  info  :   * Primitives:          271306
08:27:27 AM m1.small LIGHT  info  : Light Server stats:
08:27:27 AM m1.small LIGHT  info  :   * Light sources found: 15
08:27:27 AM m1.small LIGHT  info  :   * Light samples:       57
08:27:27 AM m1.small SCENE  info  : Rendering ...
```

Bei der prototypischen Middleware-Implementierung wird jeder Logeintrag zu einem *Sessionkey* in einer Datei abgespeichert, welche den Namen des *Sessionkey* trägt. Der *Sessionkey* selbst ist in den Beispielen nicht abgebildet. Er wird jeweils zum Beginn der Session am Dateianfang und zusätzlich am Ende der Session am Dateiende geschrieben.

7.3.5 Installation/Deployment von Applikationsprofilen

Nachdem aus den Logdaten einer *Mobilen Cloud-unterstützten Anwendung* ein Anwendungsprofil konstruiert wurde, wird diese auf dem Broker installiert. Dazu und zur nachfolgenden Nutzung durch Clients stehen die vier externen API-Methoden `deploy(...)`, `undeploy(...)`, `calculateMCmulti(...)` und `calculateMCsingle(...)` zur Verfügung. Es sei hier nochmals darauf hingewiesen, dass die Installation von Applikationsprofilen auf dem

Broker ausschließlich zum Zweck der bedarfsgerechten Auswahl von VM-Ressourcen für *Mobile Cloud-unterstützte Anwendungen* geschieht. Die *Mobile Cloud-unterstützte Anwendung* selbst ist auch ohne ein hinterlegtes Profil lauffähig.

Diese vier externen Methoden werden benutzt, um dem Client zur Laufzeit eine situationsbezogene Auswahl sinnvoller VM-Typen zur Unterstützung bei berechnungsintensiven Aufgaben zu präsentieren. Ein Profil wird über die bereits vorgestellte Methode `deploy(...)` installiert. Neben Parametern wie den Namen der möglichen VM-Typen und deren Kosten ist auch der Quelltext der Laufzeitfunktion *lauf(...)* Teil der Parameterliste dieser Methode. Dabei wird für jeden möglichen Subtyp der Applikation eine eigene Vorhersagemethode benötigt, weshalb ein Array aus Quelltexten übergeben werden muss (`String[] lauf`). Dies ist notwendig, da der Haupttyp (`String name`) stets "PIPE" ist. Der Prototyp der Methode `deploy(...)` ist nachfolgend noch einmal beschrieben.

```
bool deploy(String name, String[] lauf, String[] index,
            float[] kosten, String VM_image, float trafficcost, String user, String pass)
```

Um größtmögliche Flexibilität bei der Konstruktion der Laufzeitfunktion $lauf(\dots)$ zu gewährleisten, wird die Laufzeitfunktion $lauf(\dots)$ in *Java* spezifiziert. Dazu wird für jede Applikation eine Klasse erstellt, die das *Java* Interface `LaufInterface` implementiert, welches nachfolgend dargestellt ist.

```
// Java Code Beispiel
public interface LaufInterface {
        public float lauf(int type, int[] size);
}
```

Die implementierte Laufzeitfunktion `lauf(...)` muss dann für jeden konfigurierten VM-Typ aus `String[] index` und einer gegebenen Eingabegröße `size` eine Laufzeit Δt_{work} oder eine Rate f_{work} berechnen, je nachdem um welche Art von Applikation es sich handelt (siehe Kapitel 6.1.1). Ist die Laufzeitschätzung von mehreren Parametern abhängig, so kann die Eingabegröße `size` mehrere Werte umfassen, die als Array anzugeben sind. Der Klassenname der Implementierung von `LaufInterface` ist dabei identisch zum Subtyp der Applikation. Nachfolgende exemplarische Klasse `BeispielApp` verdeutlicht eine solche Implementierung.

```
// Java Code Beispiel
public class BeispielApp implements LaufInterface {

    @Override
    public float lauf(int type, int[] size) {
        float ret = 0.0f;
        switch (type) {
        case VMTypes.m1_small:
            ret = size[0] / 100;
            break;
        case VMTypes.c1_medium:
            ret = size[0] / 250;
            break;
        case VMTypes.c1_xlarge:
            ret = size[0] / 400;
        default:
            break;
        }
        return ret;
    }
}
```

Bei der Installation eines Applikationsprofils durch die Methode `deploy(...)` enthält jeder Eintrag des Arrays `lauf` den Quellcode einer Klasse wie `BeispielApp`. Jede dieser Klassen wird vom Broker-Server kompiliert und kann entsprechend einer Anfrage später geladen werden. Der Kompilier- und Ladevorgang ist der Code-Installation zur Laufzeit ähnlich, die bereits in Kapitel 7.3.2 vorgestellt wurde.

Um für eine Vorhersage der Laufzeit für die Applikation "PIPE" und den Subtyp "BeispielApp" die möglichen Cloud-Optionen zu berechnen, wird die Klasse `BeispielApp` auf dem Broker instanziiert und die Laufzeitfunktion `lauf(int type, int[] size)` entsprechend innerhalb von `calculateMCsingle(...)` oder `calculateMCmulti(...)` aufgerufen. Im nachfolgenden Kapitel ist beschrieben, wie die dafür notwendigen API-Aufrufe seitens des Clients ausgeführt werden, um Cloud-Optionen zur Laufzeit zu berechnen.

Für den Aufruf der externen SOAP-API-Methoden `deploy()` und `undeploy()` existiert keine Bibliotheksunterstützung, da die Installation und Deinstallation von Anwendungsprofilen nicht vom Client sondern nur durch einen Administrator erfolgt. Aufrufe können durch bekannte Hilfsprogramme wie *SoapUI*[10] durchgeführt werden.

7.3.6 Bestimmung von Cloud-Optionen auf Clientseite

Seitens des Clients kann die Klasse `SOAPClient` aus dem Paket `ferb.common` verwendet werden, um die SOAP API des Broker anzusprechen. Dazu implementiert die Klasse die bereits beschriebenen externen API-Methoden des Broker `calculateMCmulti(...)` und `calculateMCsingle(...)` in *Java*. Die Klasse `SOAPClient` führt dabei auch die notwendigen Konvertierungen zwischen *Java*-Typen und XML-Typen durch.

Um Cloud-Optionen für eine beim Broker installierte Applikation zu berechnen, müssen zunächst die benötigten Parameter bestimmt werden. Deshalb sind zur Laufzeit die Größe der Eingabe (`size`), die zu veranschlagende Kommunikationszeit (`t_comm`) und die lokal auf dem Client zu veranschlagende Ausführungszeit (`t_mobil`) zu bestimmen. Diese werden den Methoden der Klasse `SOAPClient` übergeben und zum Broker übertragen, der daraus mögliche Cloud-Optionen berechnet.

Mit Hilfe des Werkzeugs *SPEEDTEST*, welches in Abschnitt 7.2.1 vorgestellt wurde, wird zunächst die verfügbare Upload- und Downloadbandbreite bestimmt. Diese wird anschließend nach Gleichung (4.6) und (4.8) in einen realistischen Wert umgerechnet. Hierfür kann die Klasse `NetworkHelper` aus dem Paket `ferb.common` verwendet werden. Sie implementiert die statische Methode `calculateTransmissionTime(...)`, deren Prototyp nachfolgend dargestellt ist:

```
// Java Code Beispiel
public static float calculateTransmissionTime(int size, int bw)
```

Die Methode erwartet die Größe der Übertragungsdaten `size` in Byte und die verfügbare Bandbreite `bw` in Byte/Sekunde. In Kapitel 4 wurde das Pipelining-Verfahren erarbeitet, wodurch unter Berufung auf Gleichung (4.6) für die betrachteten 3G und WLAN Netzwerke die RTT vernachlässigt werden kann, da sie die Übertragungszeit bei diesem Verfahren kaum beeinflusst. Bei der Angabe der Bandbreite ist darauf zu achten, dass das Minimum aus

[10] *SoapUI* ist eine Desktop-Applikation zum Inspizieren, Aufrufen, Überwachen SOAP/WSDL-basierter *Web Services*. *SoapUI* ist Open Source Software und in Java geschrieben. (Verfügbar unter Onlineressource `http://www.soapui.org/`(abgerufen am 29.10.2013))

Upload- und Downloadbandbreite anzugeben ist (siehe Gleichung (4.8)), da im Mobilbereich oft asynchrone Übertragungsraten für Upload und Download vorliegen.

Die lokal auf dem Client zu veranschlagende Ausführungszeit ist oft nur bei benutzerterminierten Applikationen überhaupt zu bestimmen. Dies kann beispielsweise durch eine initiale lokale Ausführung der Applikation geschehen, bei der die erreichte Performance gespeichert wird. Für Anwendungen, bei denen dies nicht möglich ist oder die schlicht nur sehr eingeschränkt auf dem Client ausgeführt werden können, kann als Laufzeit für eingabeterminierte Applikationen `Float.MAXVALUE` angegeben werden. Für benutzerterminierte Applikationen ist analog eine Rate von `0.0` anzugeben. Dies führt dazu, dass keine Cloud-Optionen bei der Berechnung durch den Broker ausgefiltert werden (siehe Algorithmus 3 und Algorithmus 4 aus Kapitel 6.1.3).

Sind alle Parameter bestimmt, können über die Methoden `calculateMCsingle(...)` oder `calculateMCmulti(...)` Cloud-Optionen berechnet werden. Im nachfolgenden Kapitel werden die Auswahloptionen anhand verschiedener Applikationen noch genauer vorgestellt. Über die externe API-Methode `getInstanceForType(...)` kann dann eine VM des ausgewählten Typs allokiert werden. Die zurückgelieferte Datenstruktur vom Typ `CloudSession` enthält Informationen über den *Sessionkey* und die IP-Adresse der allokierten VM, die benutzt werden, um über die Klasse `TaskClientPipeline` eine Verbindung zur VM aufzubauen und die Taskverarbeitung zu beginnen. Ein Beispiel der Verwendung der bereitgestellten Bibliotheken zur Berechnung von Cloud-Optionen (benutzerterminierte Applikation) und zur Instanziierung einer VM ist nachfolgend gegeben. Dabei werden für den in Abschnitt 7.3.3 bereits genannten `PicTask`, dessen Taskfunktion ein einzelnes Bild verarbeitet, mögliche Cloud-Optionen berechnet. Als Eingabegröße für die Laufzeitapproximation dient die Bildauflösung `imageres`. Die verfügbare Netzwerkbandbreite wurde bereits ermittelt und ist als Variable `bw` verfügbar. Die Variable `currentFramerate` beschreibt die Taskverarbeitungsrate, die lokal auf dem Gerät ohne Cloud-Unterstützung gemessen wurde.

```
// Java Code Beispiel
String selectedtask = "PicTask";          // Taskname
int imageres[] = {640 * 480};             // Bildauflösung für Laufzeitvorhersage
int imagesize = jpegpic.getData().length;// Bildgröße für die Transferzeitvorhersage
float f_mobil = currentFramerate;         // aktuelle frames per second (fps)

// Erreichbare Kommunikationsrate aus Bildgröße und Bandbreite bestimmen
float f_comm = 1.0f/ferb.common.NetworkHelper.calculateTransmissionTime(imagesize, bw);

// Cloud-Optionen via SOAPClient bestimmen lassen
ferb.common.CloudOption[] cs = ferb.common.SOAPClient.calculateMCmulti(user, pass,
                                                    "PIPE", selectedtask, imageres,
                                                    f_comm, f_mobil, imagesize);
if (cs == null) return; // Falls die entfernte Ausführung nicht ratsam wäre

// Cloud-Option 0 zur Demonstration auswählen
ferb.common.CloudOption chosen = cs[0];
// Cloud-Ressource via SOAPClient von Broker anfordern und gültige Session erhalten
ferb.common.CloudSession session = ferb.common.SOAPClient.getInstanceForType("PIPE",
                                                    chosen.type, user, pass);

// Aufbau der Verbindung zur Cloud-Ressource mit der erhaltenen Session
ferb.pipe.TaskClientPipeline tcp = new ferb.pipe.TaskClientPipeline();
tcp.connect(session.session_key, session.endpoint, 8080);
```

8 Bewertung der Middleware anhand ausgewählter Applikationen

In diesem Kapitel werden ausgewählte Applikationen vorgestellt, die unter Verwendung der vorgestellten Middleware in *Mobile Cloud-unterstützte Anwendungen* transformiert wurden. Ziel ist die Demonstration der Anwendbarkeit der entwickelten Middleware für die in Kapitel 6.1.1 beschriebenen Applikationstypen (eingabeterminiert und benutzerterminiert). Zudem werden anhand der Beispielapplikationen auch verschiedene Parameter bestimmt, die eine Einordnung der Leistungsfähigkeit der Middleware erlauben. Hierzu zählt insbesondere die Güte der vorhergesagten Ausführungszeit.

Zunächst werden synthetische Applikationen untersucht. Dabei wird analysiert, wie die Vorhersage der Ausführungszeit für definierte Szenarien, die nicht via Regression bestimmt wurden, funktioniert. Außerdem werden anhand synthetischer Applikationen weitere Metaparameter gemessen wie die Installationszeit für Taskfunktionen und die Gesamtallokationszeit für VM-Ressourcen bis zur Verarbeitung des ersten Tasks.

Anschließend wird die Güte der Vorhersage der Ausführungszeit für eine angepasste Version einer Raytracing-Software zur Berechnung fotorealistischer Bilder untersucht, die als Modul in einem 3D-Hausdesigner eingesetzt wird. Dabei handelt es sich um eine eingabeterminierte Applikation. Eine benutzerterminierte Applikation für *Android*-Geräte demonstriert danach die Anwendung der entwickelten Middleware bei der Verarbeitung von Kamerabildern.

Für alle getesteten Applikationen wurde eine experimentelle Testinstallation der vorgestellten Middleware auf der *Amazon EC2* Infrastruktur im Rechenzentrum in Irland benutzt (siehe Kapitel 2.3.2).

8.1 Synthetische Applikationen

Einige synthetische Applikationen wurden implementiert, um die Leistungsfähigkeit der Middleware zu untersuchen. Dabei war zunächst von Interesse, wieviel Zeitoverhead bis zur Bearbeitung des ersten wirklichen Tasks zu erwarten ist. Außerdem wurde untersucht, wie gut die vorhergesagte Performance der Cloud-Optionen für definierte Tasks mit der tatsächlich erreichten Performance übereinstimmt. Für die Messungen in diesem Kapitel wurden erneut die Netzwerkkonfigurationen und Testgeräte genutzt, die bereits in Kapitel 4 verwendet wurden. Die Netzwerkkonfigurationen umfassen zwei WLAN-Konfigurationen und eine Mobilfunkkonfiguration (siehe Tabelle 4.1). Zu den Testgeräten zählen zwei *Android*-Geräte und zwei Laptops, deren Eigenschaften in Anhang A zusammengefasst sind.

8.1.1 Bereitstellungszeit von VM-Ressourcen und Code-Offloading

Die Bereitstellungszeit einer Ressource umfasst die Schritte „Allokieren der VM durch den Broker “ und „Aufbau der Verbindung zur VM und Übermitteln des *Sessionkey*“ (siehe Kapitel 5.2.4). Bei beiden Schritten ist Netzwerkkommunikation notwendig. Da jedoch nur wenige Daten übermittelt werden, überwiegt hier der Einfluss der Netzwerklatenz. Die

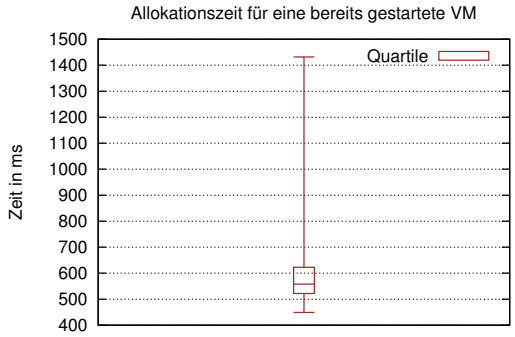


Abbildung 8.1: Gemessene Allokationszeiten für VM-Instanzen unter Verwendung der entwickelten Middleware.

Tabelle 8.1: Laufzeit ($\Delta t_{install}$) elementarer, von der Middleware bereitgestellter, Tasks auf verschiedenen VM-Typen in ms

Taskart ($msg\_size_{install}$)		Install/ Compile (10 kB)	Install/ Compile (34 kB)	CopySource (496 kB)
m1.small	∅	1400	3708	950
	σ	2031	4228	421
c1.medium	∅	1869	1794	423
	σ	1317	2522	175
c1.xlarge	∅	2336	2859	618
	σ	2239	2748	138

σ = Standardabweichung
∅ = Mittelwert

Allokation einer bereits gestarteten VM über die Broker-API-Methode `getInstanceForType(...)` wird über ein SOAP *Web Service* Interface abgewickelt. Dabei ist im Gegensatz zum Aufbau einer verschlüsselten TCP-Socket-Verbindung mit zusätzlichem Overhead für die XML-Codierung zu rechnen. Abbildung 8.1 stellt die Verteilung der Allokationszeiten für eine Reihe solcher Anfragen an den Broker dar. Die Anfragen wurden von einem Dell Streak 7 *Android* Tablet in allen drei Netzwerkkonfigurationen gestellt und umfassten 176 Allokationsvorgänge für die drei VM-Typen `m1.small`, `c1.medium` und `c1.xlarge`. Dargestellt sind die Quartile der Verteilung (Min, Max, Median sowie das 25 %- und 75 %-Quartil). Für die Bereitstellung der VM ist demnach mit einem Wert von mindestens ca. 450 ms bis ca. 1400 ms zu rechnen. Anschließend wird unter Verwendung des vom Broker zurückgelieferten *Sessionkey* eine Verbindung zur VM aufgebaut. Dieser Aufbau einer verschlüsselten Socket-Verbindung zur VM entspricht dem Experiment aus Kapitel 4.2.2, dessen Ergebnisse in Abbildung 4.9 dargestellt sind. Es wurde festgestellt, dass für diesen Verbindungsaufbau auch ca. 400 ms bis 1100 ms, je nach Kommunikationstechnologie und Konfiguration, zu erwarten sind. Somit vergehen bis zum Beginn der ersten Taskübertragung zwischen 950 ms und 2500 ms, wobei die Messungen auch starke Ausreißer enthalten und in mindestens 75 % der Fälle mit einer Dauer bis ca. 1500 ms gerechnet werden kann. Hierbei wurde jedoch nicht beachtet, dass die Allokation einer freien VM durch den Broker auch erheblich mehr Zeit in Anspruch nehmen kann, wenn diese erst noch gestartet werden muss, wie in Kapitel 5.4 durch Simulation untersucht wurde.

Vor der Bearbeitung benutzerdefinierter Tasks müssen die entsprechenden Taskklassen zunächst via `InstallTask` auf der VM installiert werden. Hierfür wird weitere Zeit benötigt. Analog kann auch ein `CopySourceTask` verwendet werden, um bereits vorkompilierte *Java* Class-Dateien zur VM zu übertragen. Bei `InstallTask` und `CopySourceTask` handelt es sich um elementare, also durch die Middleware bereitgestellte, Klassen (siehe Kapitel 7.3.2). Für beide Arten der Codeinstallation wird sowohl Übertragungszeit als auch Ausführungszeit auf der VM benötigt. Für die Übertragungszeit sind die in Kapitel 4.4 erarbeiteten Vorhersagemethoden zu nutzen. Die reinen Ausführungszeiten verschiedener elementarer Tasks auf drei verschiedenen VM-Typen sind in Tabelle 8.1 zusammengefasst. Die Werte stammen von Messungen der in den nachfolgenden Abschnitten vorgestellten Applikationen. Der `Copy-`

`SourceTask` ist für die Installation des Programmcodes der Raytracing-Applikation zuständig. Die beiden `InstallTask` erledigen die Installation des Quellcodes der Bildverarbeitungstasks der *Android*-Applikation. Für die Ausführungszeiten der `InstallTasks` ist kein Muster zu erkennen. Sie schwanken sehr stark. Es kann jedoch davon ausgegangen werden, dass die durchschnittliche Ausführungszeit nicht mehr beträgt als 4 s. Ein Erklärung für die starken Schwankungen ist darin zu suchen, dass beim Kompilieren viele benötigte Bibliotheken von der virtuellen Festplatte der VM-Instanz gelesen werden müssen. Wie in Kapitel 6.3.3 analysiert wurde, sind bei I/O-intensiven Aufgaben für VMs besonders hohe Schwankungen zu erwarten. Etwas anders verhält es sich für den `CopySourceTask`. Dessen mittlere Verarbeitungsdauer und Standardabweichung der Messwerte ist wesentlich geringer. Für die Werte in Tabelle 8.1 wurden zwischen 57 und 333 verschiedene Messwerte pro Taskart und VM-Typ ausgewertet. Für den `CompileTask` sind keine Ergebnisse abgebildet. Dieser entspricht von seinem Verhalten bis auf das anfängliche Speichern der Quellcodedatei aber einem `InstallTask`. Das Speichern benötigt jedoch nur wenige Millisekunden. Somit ist für beide Taskarten bei gleicher Quellcodemenge etwa das gleiche Verhalten und somit auch die gleiche Laufzeit zu erwarten. Eine Abhängigkeit zwischen der Größe der übergebenen Daten und der Tasklaufzeit wurde hier nicht analysiert, da nur maximal zwei verschiedene Eingabegrößen zur Verfügung standen und die Schwankung sehr hoch war. Als Ergebnis der Untersuchungen kann dennoch festgehalten werden, dass es wesentlich schneller geht, vorkompilierte Klassendateien zu installieren und dass diese Art der Codeinstallation zu bevorzugen ist.

Insgesamt setzt sich die Zeit bis zur ersten Übertragung eines Anwendungstasks für die getesteten Applikationen aus der Allokationszeit für die VM von ca. 1,5 s und der Zeit für die Übertragung und Installation des Codes zusammen. Dafür fallen ca. 4 s für die Quellcodeinstallation und für die Programmcodeinstallation sind nur ca. 0,5–1 s zu erwarten (siehe Tabelle 8.1). Gemäß der geringeren Datengröße fallen bei 2 Mbit/s Upload maximal 0,5 s für die Datenübertragung der Quellcodedatei an. Für den umfangreicheren Programmcode werden jedoch ca. 2 s an Übertragungszeit erwartet. Somit ergeben sich bei Quellcodeinstallation insgesamt ca. 6 s Wartezeit bis zur ersten Taskausführung. Verwendet man vorkompilierten Programmcode, ist mit einer Wartezeit von ca. 4–4,5 s zu rechnen.

8.1.2 Laufzeitvorhersage für synthetische benutzerterminierte Applikationen

In diesem Abschnitt wird untersucht, wie gut die Vorhersage der Laufzeit tatsächlich ist, wenn alle Tasks exakt die gleich Taskfunktion ausführen. Dazu wurde als Taskfunktion eine Zählschleife verwendet, die unabhängig von der Eingabe immer die gleichen Arbeitsschritte ausführt und damit einen CPU-Kern der VM-Instanz gänzlich beansprucht. Zudem wird die Größe der Übertragungsdaten exakt festgelegt. Laufzeitschwankungen durch verschiedene Eingaben und dadurch auch unterschiedlich aufwendige Taskfunktionen werden somit ausgeschlossen. Etwaige dennoch messbare Schwankungen können nur durch die Middleware selbst, durch die verwendeten VM-Instanzen oder die verwendete Kommunikationstechnologie verursacht werden. Für die Untersuchungen werden drei VM-Typen aus dem Portfolio von *Amazon EC2* benutzt, die in ihrer Performance um ca. den Faktor 4 verschieden sind: `m1.small`, `c1.medium` und `c1.xlarge`. Dass diese Auswahl für gut abgegrenzte Vorschläge von Cloud-Optionen sinnvoll ist, wurde in Kapitel 6.2.5 erläutert.

Für benutzerterminierte Applikationen kann aufgrund der unzureichend bekannten Eingabe nur eine Verarbeitungsrate für gleichbleibende Taskfunktionen und schwankungsarme Ein-/Ausgabedaten gemessen werden. Die Taskverarbeitungsrate wird durch die maximal

Tabelle 8.2: Abweichung der Vorhersage für synthetische benutzerterminierte Applikationen in % (Taskrate $f_{work} = 1.0$ konstant)

Datengröße	UMTS				HOME				CAMPUS			
in kB →	0	10	50	100	0	10	50	100	0	10	50	100
`m1.small` $\Delta t_{work} \approx 1\,s$	0,81	1,37	0,76	0,35	0,62	1,77	0,09	0,90	1,15	4,42	6,37	8,77
`c1.medium` $\Delta t_{work} \approx 2,5\,s$	0,13	3,14	8,84	1,08	2,43	0,86	0,15	0,24	0,87	0,87	0,87	0,87
`c1.xlarge` $\Delta t_{work} \approx 10\,s$	2,28	6,20	5,55	7,90	3,26	1,29	3,24	8,34	1,82	0,57	2,91	7,00

mögliche Taskverarbeitungsrate auf der VM (f_{work}) und durch die Kommunikationsrate f_{comm} begrenzt, die durch die Größe der Ein-/Ausgabedaten pro Task und die Netzwerkperformance gegeben ist (siehe Kapitel 6.1.3, insbesondere Bedingung (6.7)). Der Vorhersagewert entspricht demnach dem Minimum aus f_{work} und f_{comm}.

Zunächst wird untersucht, welche Abweichung von der Vorhersage für eine Taskrate f_{work} von ca. 1,0 gemessen werden kann. Es wurde dafür Sorge getragen, dass f_{comm} stets größer war als f_{work}, um eine Beeinflussung durch die Netzwerkperformance für diesen Test zu vermeiden. Als Taskfunktion wurde eine Zählschleife verwendet, deren Anzahl von Schleifendurchläufen auf einer `m1.small`-Instanz so geeicht wurde, dass die Länge der Tasklaufzeit Δt_{work} ca. 1 s beträgt. Hierbei wurden einige Schleifendurchläufe abgewartet, bis durch den JIT-Compiler vorgenommene Optimierungen abgeschlossen waren und eine stabile Tasklaufzeit erreicht wurde. Bei einem CPU-Kern ergibt sich somit eine Taskverarbeitungsrate von ca. 1,0. Durch eine auf dem Dell Streak 7 *Android* Tablet installierte Testapplikation, wurden dann fortwährend Tasks zur VM-Instanz übertragen. Um die Taskrate auch für die performanteren Instanzen bei ca. 1,0 zu belassen, wurden die Durchläufe der Zählschläfe auf diesen Instanzen mit dem Faktor des Performanceunterschieds zur `m1.small`-Instanz multipliziert. Als Maß wurde die ECU-Bewertung durch *Amazon EC2* benutzt. Die `m1.small`-Instanz hat 1 ECU, die `c1.medium` hat beispielsweise 5 ECU. Somit wurde die Zählschleife mit dem Faktor 5 multipliziert. Dies führte im Ergebnis tatsächlich zu einer etwa gleichen Taskrate auf allen VM-Typen. Die erreichte Tasklaufzeit wurde für jede Instanz trotzdem noch einmal separat überprüft und gegebenenfalls leicht angepasst. Für die drei Netzwerkkonfigurationen HOME, CAMPUS und UMTS und verschiedene Größen von Ein-/Ausgabedaten der einzelnen Tasks wurde anschließend die Abweichung der gemessenen Taskrate von der Vorhersage bestimmt. Die Ergebnisse sind in Tabelle 8.2 dargestellt. Dabei wurde pro Wert jeweils nur eine Messung über einen Zeitraum von mindestens 30 s durchgeführt. Ein wirklicher Benutzer würde die Applikation vermutlich auch nur einmal ausführen. Dargestellt ist jeweils die Abweichung der durchschnittlich erreichten Taskverarbeitungsrate im gemessenen Zeitraum vom vorhergesagten Wert. Die Netzwerkperformance limitierte die erreichbare Taskrate hier nicht. Auch für 100 kB Ein-/Ausgabedaten war in allen drei Konfigurationen die mögliche Kommunikationsrate höher.

Die Ergebnisse zeigen, dass die Abweichung für alle Messungen sehr gering sind. Zudem lässt sich an den Messungen auch erkennen, dass die für die Taskverarbeitung notwendigen Datenübertragungen ohne merkliche Auswirkung auf die tatsächliche Verarbeitungszeit der Tasks auf der VM im Hintergrund durchgeführt werden. Zudem war die Performance der getesteten VM-Instanzen über den gesamten Messzeitraum von mehreren Stunden stabil.

Tabelle 8.3: Abweichung der Vorhersage für synthetische benutzerterminierte Applikationen in % (Programmcode des Tasks konstant → f_{work} variabel)

		UMTS				HOME				CAMPUS			
Datengröße in kB →		0	10	50	100	0	10	50	100	0	10	50	100
c1.medium $\Delta t_{work} \approx 500\,ms$	f_{work}	3,97	3,97	3,97	3,97	3,97	3,97	3,97	3,97	3,97	3,97	3,97	3,97
	f_{comm}	∞	22,40	4,48	2,56	∞	14,72	2,94	1,47	∞	35,84	7,17	3,58
	f_{mess}	3,88	3,86	3,77	3,70	3,90	4,12	3,47	1,84	3,39	3,42	3,54	3,33
	Δ in %	2,16	2,75	4,93	44,71	1,78	3,78	17,71	25,18	14,55	13,93	10,76	7,02
c1.xlarge $\Delta t_{work} \approx 800\,ms$	f_{work}	10,00	10,00	10,00	10,00	10,00	10,00	10,00	10,00	10,00	10,00	10,00	10,00
	f_{comm}	∞	22,4	4,48	2,56	∞	14,72	2,94	1,47	∞	35,84	7,17	3,58
	f_{mess}	9,55	9,26	4,08	2,36	9,89	9,85	4,25	2,14	8,62	9,71	8,04	3,17
	Δ in %	4,46	7,37	9,01	7,70	1,11	1,46	44,40	45,06	13,82	2,92	12,13	11,57

Abweichung > 20 %

In einem zweiten Szenario wurde im Gegensatz zum bereits beschriebenen Test die Taskfunktion auf allen Instanzen nicht verändert. Es wurde die Variante benutzt, die auf einer `m1.small`-Instanz eine Ausführungszeit von ca. 1 s benötigt. Dies führt dazu, dass die Taskfunktion auf den `c1.medium`- und `c1.xlarge`-Instanzen nicht nur schneller bearbeitet wird, sondern aufgrund der höheren CPU-Kernanzahl auch mehr Tasks parallel verarbeitet werden können. Somit steigt die Taskverarbeitungsrate gegenüber `m1.small` teils erheblich an. Dafür kann in diesem zweiten Szenario die Netzwerkperformance nicht mehr vernachlässigt werden. Die messbare Taskverarbeitungsrate f_{mess} wird begrenzt durch das Minimum aus der maximal möglichen Taskverarbeitungsrate auf der VM (f_{work}) und durch die Kommunikationsrate f_{comm}. Tabelle 8.3 stellt die Ergebnisse der Messungen dar. Auch hier wurde jede Messung einmal für eine Dauer von mindestens 30 s durchgeführt. Zusätzlich zur durchschnittlich erreichten Taskrate f_{mess} sind noch f_{work}, f_{comm} und die Abweichung von der Vorhersage angegeben. Für die `m1.small`-Instanz entsprechen die Messungen den gleichen Werten wie in Tabelle 8.2, weshalb sie nicht noch einmal dargestellt sind.

Es weichen nur 4 Werte der 36 Messungen (ca. 11 %) um mehr als 20 % von der Vorhersage ab. Dabei ist der Grund für die starke Abweichung stets die falsch eingeschätzte Netzwerkperformance. Keiner der Messwerte weicht um mehr als 50 % ab, wodurch das Güterkriterium $K_{Q_{gut}}$ erfüllt ist. Mehr als zwei Drittel aller Werte weicht sogar nur um weniger als 10 % von der Schätzung ab. In der Zusammenfassung kann davon ausgegangen werden, dass die Taskverarbeitungsrate sich sehr gut vorhersagen lässt, wenn die Netzwerkperformance als limitierender Faktor keine Rolle spielt, wenn die Tasks also nur wenige Ein-/Ausgabedaten haben oder die Tasklaufzeiten recht hoch sind. Limitiert jedoch die Netzwerkperformance die Taskverarbeitungsrate, so wird die Vorhersage etwas ungenauer. Im Allgemeinen sollte es jedoch vermieden werden, dass die Taskverarbeitungsrate durch die Netzwerkperformance limitiert wird, da dadurch auch der Kostenoverhead durch eine schlecht ausgelastete VM-Instanz steigt. Somit relativiert sich die teils ungenaue Vorhersage in diesem Bereich.

Eine Restriktion des entwickelten Verfahrens lässt sich im Vergleich der Tabellen 8.2 und 8.3 erkennen. Die Messungen wurden an unterschiedlichen Tagen gemacht. Dabei wurden für beide Messreihen offensichtlich verschieden schnelle VM-Instanzen der Typen `c1.xlarge` und `c1.medium` zurückgeliefert. Es wurden jeweils mehrere Instanzen angefordert. In Tabelle 8.2 zeigt die `c1.xlarge`-Instanz eine vierfach höhere Performance als die `c1.medium`-Instanz, was der um viermal höheren CPU-Keranzahl entspricht. In Tabelle 8.3 beträgt der Unterschied für die Messungen ohne Datentransfer hingegen nur 2,5. Dass es Perioden größerer Schwankungen gibt, wurde in Kapitel 6.3.3 bereits erläutert. Um diesen längerfristigen Schwankungen entgegenzuwirken, müssten die Profile für die Laufzeitvorhersage häufiger aktualisiert werden. Im hier beschriebenen Fall wurden für beide Tage jeweils neue Profile erstellt.

Tabelle 8.4: Abweichung der Vorhersage für synthetische eingabeterminierte Applikationen in %

	Datengröße ↓	CAMPUS (7 Mbit/s)			HOME (3 Mbits/s)			UMTS (4,5 Mbit/s)		
	Δt_{work} →	10 s	60 s	5 min	10 s	60 s	5 min	10 s	60 s	5 min
`m1.small`	0,2 MB	3,06	4,96	2,32	14,71	16,31	0,66	13,65	5,92	3,76
	12 MB	6,17	4,12	6,83	17,44	5,11	10,20	10,21	2,20	0,27
	60 MB	15,39	9,32	10,23	28,01	17,61	6,60	10,02	12,86	5,83
`c1.medium`	0,2 MB	0,51	1,97	13,35	1,16	1,78	2,03	48,46	14,57	11,98
	12 MB	12,61	1,58	3,03	17,10	11,89	2,92	14,37	8,23	9,94
	60 MB	0,29	16,74	0,57	26,02	14,87	11,62	0,24	8,94	1,64
`c1.xlarge`	0,2 MB	4,44	4,82	4,40	1,04	1,06	16,14	15,45	10,26	3,52
	12 MB	15,98	10,05	3,44	26,20	8,96	8,79	18,12	1,71	7,18
	60 MB	19,72	14,17	12,65	24,76	22,87	3,27	11,11	18,42	17,75

■ Abweichung > 20 %

8.1.3 Laufzeitvorhersage für synthetische eingabeterminierte Applikationen

Die Taskfunktion zur Generierung einer synthetischen Last auf den VM-Instanzen wurde für die eingabeterminierte Applikation so verändert, dass die Zählschleife, die schon für die Einzelprozessortasks der benutzerterminierten Applikation benutzt wurde, nun länger und gegebenenfalls von mehreren Threads gleichzeitig abgearbeitet wird. Dabei werden jeweils so viele Threads verwendet, wie CPU-Kerne zur Verfügung stehen. Somit wird stets genau ein sogenannter Multiprozessortask ausgeführt. Es ergibt sich eine gleichbleibende Abfolge von Arbeitsschritten, die jede VM-Instanz vollständig beansprucht und auf jeder Instanz eines VM-Typs in etwa die gleiche Bearbeitungsdauer benötigt.

Zur Bestimmung der Güte der Vorhersage wurden jeweils einzelne Tasks verschiedener Konfigurationen abgesetzt und deren Gesamtbearbeitungsdauer gemessen. Diese besteht bei eingabeterminierten Applikationen aus der Summe der Übertragungsdauer der Daten und der tatsächlichen Bearbeitungszeit auf der VM (siehe Kapitel 6.1.3). Für den Test wurden alle Kombinationen aus Verarbeitungszeit und Datengröße benutzt, die bereits in Kapitel 5.4.6 zur Analyse der Datenübertragungskosten verwendet wurden. Als Verarbeitungszeiten Δt_{work} wurden 10 s, 60 s und 5 min gewählt. Als Datenübertragungsgrößen wurden 200 kB, 12 MB und 60 MB gewählt. Die Daten wurden für jede Messung zunächst zur VM übertragen und nach Berarbeitung der Taskfunktion wieder zurück transferiert. Jede einzelne Messung wurde genau einmal durchgeführt auf den drei konfigurierten VM-Typen `m1.small`, `c1.medium` und `c1.xlarge` und den drei Netzwerkkonfigurationen HOME, UMTS und CAMPUS. Als Testgerät wurde der Dell Latitude Laptop verwendet. Für jede VM-Instanz wurde zunächst die Zählfunktion passend geeicht, sodass sie die gewünschte Tasklaufzeit realisiert. Vor jedem Test wurde dann die verfügbare Bandbreite ermittelt und entsprechend zur Vorhersage benutzt. Tabelle 8.4 fasst die Abweichungen der gemessenen Werte von der Vorhersage für alle 81 Messungen zusammen.

Nur 6 Messungen (ca. 7 %) wichen um mehr als 20 % von der Schätzung ab. Für die HOME-Konfiguration war es aufgrund der asynchronen Übertragungsrate für Upload und Download schwierig, die tatsächlich nutzbare Bandbreite abzuschätzen. Deshalb traten hier vermehrt hohe Abweichungen auf, wenn die Taskausführungszeit gering und die Datengröße hoch war. In der UMTS- und CAMPUS-Konfiguration waren Upload und Download nur leicht ver-

schieden, wodurch sich gleichmäßigere Abweichungen ergaben. In der UMTS-Konfiguration war eine starke Abweichung festzustellen, die auf eine kurzzeitig stark verlangsamte Ausführungsgeschwindigkeit der VM-Instanz zurückzuführen war. Im Allgemeinen ist die Vorhersage jedoch auch für diese Art Applikationen sehr gut. Kein Messwert wich um mehr als 50 % von der Vorhersage ab.

8.1.4 Zusammenfassung synthetische Applikationen

Getestet wurde die Verlässlichkeit der Vorhersage für Applikationen, die im Verhalten selbst nicht schwanken, deren ausgeführte Verarbeitungsschritte also immer gleich sind. Etwaige Schwankungen der Ausführungszeit werden dann durch die Middleware und deren Komponenten selbst verursacht. Die gemessenen Abweichungen von der Vorhersage lagen sowohl für eingabeterminierte als auch für benutzerterminierte Applikationen in der überwiegenden Mehrzahl unter 20 %. Abweichungen sind fast immer auf die falsch eingeschätzte Netzwerkperformance zurückzuführen. Dabei wurde die Übertragungsrate häufig zu pessimistisch eingeschätzt. In einem Fall wurde jedoch auch ein kurzzeitiger Performanceeinbruch auf einer VM-Instanz registriert. Im Ergebnis ist festzuhalten, dass die Erstellung von Profilen zur Laufzeitvorhersage von *Mobilen Cloud-unterstützten Anwendungen* auf der entwickelten Middleware und im getesteten Rechenzentrum von *Amazon EC2* von Zeit zu Zeit überprüft und erneuert werden muss. Es ist anzunehmen, dass dies auch für andere IaaS-Anbieter gilt (siehe Kapitel 6.3.3).

8.2 Raytracing

Die Software *Sweethome 3D*[1] ist ein quelloffener und in *Java* geschriebener 3D-Hausdesigner. Er ermöglicht die grafische Modellierung von Wohnungen und Häusern inklusive der Ausstattung mit Möbeln und Texturen. Abschließend kann die Szene aus verschiedenen Perspektiven (virtuell) durchschritten werden. Zusätzlich können auch Bilder aus frei wählbaren Kamerablickwinkeln via *Raytracing*[2] gerendert werden. Dabei wird zum Rendern der Szene die ebenfalls quelloffene und in *Java* geschriebene Raytracing-Software *Sunflow*[3] verwendet. Die Visualisierung ist sehr zeitaufwendig. *Sunflow* unterstützt jedoch Multicore-Systeme und deshalb lässt sich die Renderinggeschwindigkeit auf solchen Systemen deutlich beschleunigen. Oft findet die Erstellung der Szenen beispielsweise vor Ort in einer Wohnung auf einem mobilen Gerät statt. Laptops und Netbooks haben üblicherweise keine leistungsstarken Prozessoren eingebaut, um stattdessen eine hohe Akkulaufzeit zu ermöglichen. Das Erstellen fotorealistischer Bilder ist somit im mobilen Einsatz nur eingeschränkt möglich. Die Software *Sweethome 3D* wurde deshalb im Rahmen dieser Arbeit in eine *Mobile Cloud-unterstützte Anwendung* transformiert. Dazu wurde die Rendering-Funktionalität ausgelagert. Ein Screenshot der Applikation ist in Abbildung 8.2 zu sehen. Da die Eingabe für das Rendering vollständig bekannt ist, handelt es sich um eine eingabeterminierte Applikation (siehe Kapitel 6.1.1). Dass die Aufspaltung von *Sunflow* in eine verteilte Anwendung durch Auslagerung des Renderings auf

[1] Verfügbar unter Onlineressource `http://www.sweethome3d.com/de/index.jsp` (abgerufen am 16.2.2014)

[2] *„Raytracing (Strahlverfolgung) ist ein wirkungsvolles, aber bezüglich Rechenzeit und Speicherplatz aufwändiges Verfahren zur Berechnung fotorealistischer Bilder. [...] Das Prinzip besteht im Gegensatz zur physikalischen Natur darin, vom Betrachterstandpunkt aus Sichtstrahlen durch jedes Pixel zu schicken und den Weg derselben durch die Szene zu verfolgen.“* [OM04, S. 257ff.]

[3] Verfügbar unter Onlineressource `http://sunflow.sourceforge.net/` (abgerufen am 16.2.2014)

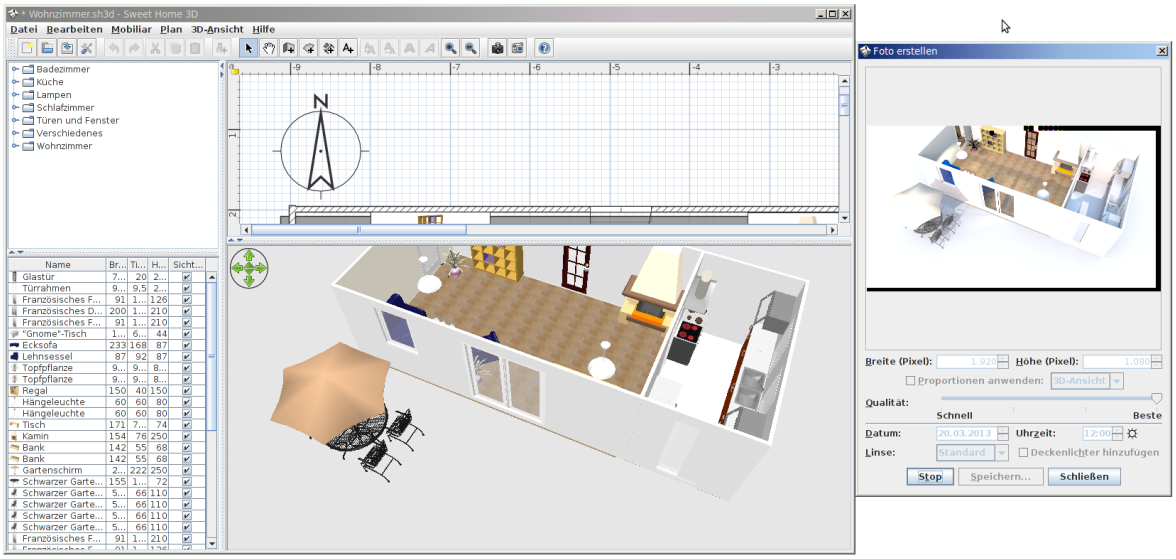


Abbildung 8.2: Screenshot des 3D-Hausdesigners *Sweethome 3D*. Rechts ist zusätzlich das Fenster zur Erstellung des durch Raytracing gerenderten Bildes zu sehen.

performante Serverressourcen zu guten Speedups führen kann, wurde bereits in einer früheren Arbeit [FRH10] gezeigt.

Um die Leistungsfähigkeit des Raytracings zu bewerten und eine Vorhersagemethode der Ausführungszeit für die bedarfsgerechte Nutzung zu realisieren, wurden 32 Beispielszenen von Häusern, Wohnungen, einzelnen Zimmern und Gärten benutzt. Sie sind in Anhang E visualisiert. Um eine breit gefächerte und realistische Stichprobe zu erhalten, wurden die Szenen aus verschiedenen Quellen zusammengestellt. Sieben Szenen stammen aus den offiziellen Beispielszenen zu *Sweethome 3D*. Vier weitere Szenen wurden selbst konstruiert und reflektieren Extremfälle, also sehr einfache oder sehr komplexe Szenen, wie beispielsweise mit vielen Grünpflanzen. Die restlichen 21 Szenen stammen von 20 weiteren Personen, die gebeten wurden, eine Szene zu kreieren. Diese Szenen reflektieren teils realistische Modellierungen und teils fiktionale. Es wurde darauf geachtet, dass die Szenen sinnvolle Bilder erzeugen. Beispielsweise wäre es auch denkbar, die Kamera in Richtung Himmel oder auf eine weiße Wand zu richten. Für diese Szene würde dann jedoch die Ausführungszeit wesentlich kürzer ausfallen und die zu entwickelnde Vorhersage eventuell verfälschen. Für die Bewertung der Güte der Vorhersage wurden die 32 Szenen in 16 Trainingsszenen und 16 Testszenen unterteilt. Anhand der Trainingsszenen wurde das Profil zur Vorhersage generiert. Anhand der Testszenen wurde die Güte der Vorhersage überprüft. Die Trainingsszenen sind in Anhang E jeweils links abgebildet.

8.2.1 Implementierung

Sunflow wird innerhalb des Quellcodes von *Sweethome 3D* als Modul genutzt und alle Aufrufe werden über die Klasse *SunflowAPI* realisiert. Die Transformation von *Sweethome 3D* in eine *Mobile Cloud-unterstützte Anwendung* erforderte zwei Schritte. Zunächst wurden die Aufrufe zur `SunflowAPI` in eine Variante zur entfernten Ausführung überführt. Dazu wurde die Klasse *SunflowAPI* mit einem *Remote Object* Interface versehen und damit für *Java* RMI nutzbar

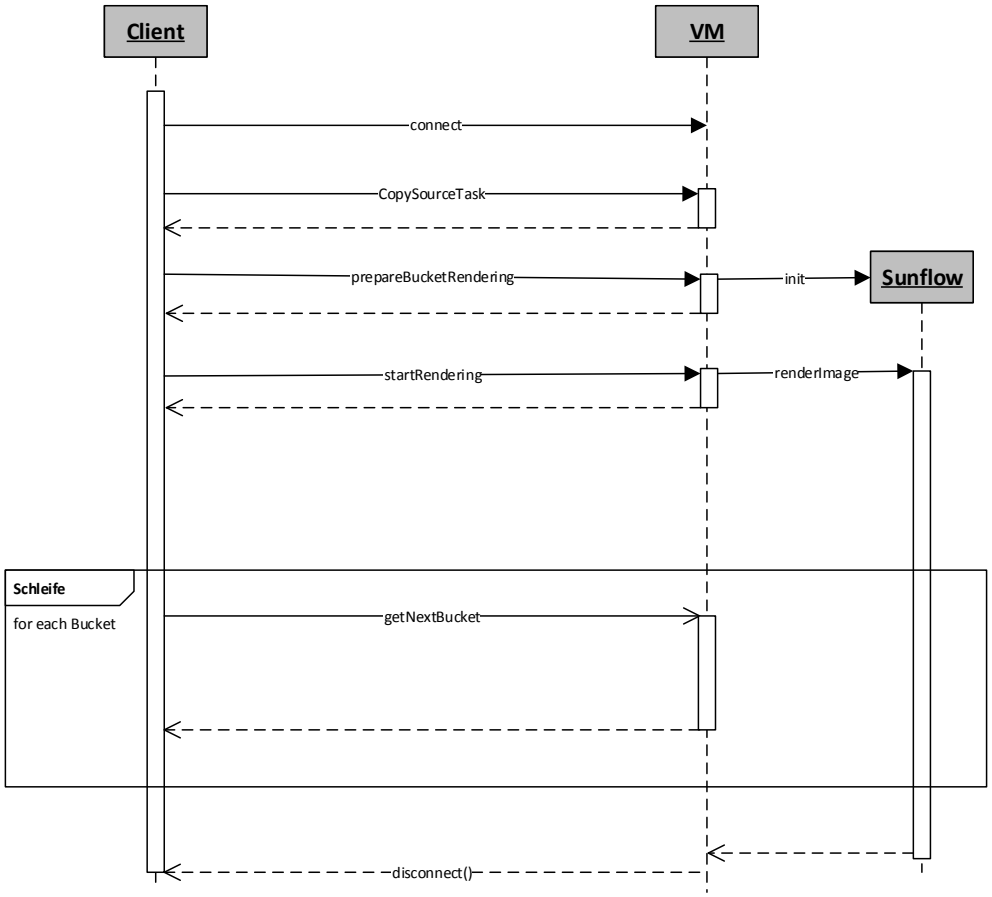


Abbildung 8.3: Sequenzdiagramm des Ablaufs eines Renderings.

gemacht. Das resultierende neue Interface der `SunflowAPI` konnte durch die in Kapitel 7.3.3 beschriebene Methode in eine Task-Pipelining-kompatible Variante transformiert werden. Als Ergebnis ist die Ausführung der *Remote Object* Aufrufe via synchroner Task-Verarbeitung möglich. Somit ist *Sweethome 3D* als eingabeterminierte Applikation mit der entwickelten Middleware nutzbar.

Obwohl *Sunflow* innerhalb des Quellcodes von *Sweethome 3D* bereits über eine abgeschlossene API genutzt wird, konnten die einzelnen Methoden der Klasse `SunflowAPI` aufgrund der Komplexität der Parameter nicht direkt als entfernt aufrufbare Methoden ausgelagert werden. Einige Messungen mit verschiedenen Szenen ergaben, dass teilweise bis zu 100 MB an Parameterdaten über das Interface übergeben werden, was im ausgelagerten Fall via (kabellosem) Netzwerk transferiert werden müsste. Daraufhin wurde das Interface der `SunflowAPI` aufgeteilt in das Interface `ISunflowRemoteAPI`, das weiterhin die entfernt aufrufbaren Methoden enthält, und eine Klasse `SunflowSzene`, die die Methoden der `SunflowAPI` zur Manipulation der eigentlichen Szene implementiert. Im Ergebnis werden so alle die Szene betreffenden Daten weiterhin zunächst lokal abgespeichert. Erst wenn die Erstellung der Szene abgeschlos-

sen ist, wird diese über das neue Interface `ISunflowRemoteAPI` zur entfernten Berechnung übertragen. Dazu wird die Objektinstanz der Klasse `SunflowSzene` zunächst serialisiert und anschließend noch mit dem LZMA-Algorithmus komprimiert. Hierbei werden insgesamt nur wenige MB an Daten transferiert. Je nach Größe der Szene dauert in diesem Fall jedoch die Komprimierung einige Sekunden, was bei der Gesamtverarbeitungszeit zu berücksichtigen ist. Das Remote Interface von *Sunflow* in *Sweethome 3D* ist nachfolgend dargestellt.

```
// Java Code Beispiel
public interface ISunflowRemoteAPI extends java.rmi.Remote {

    public Integer prepareBucketRendering(SunflowSzene scene) throws RemoteException;

    public boolean startRendering() throws RemoteException;

    public ReadyBucket getNextBucket() throws RemoteException;
}
```

Das Interface `ISunflowRemoteAPI` enthält zudem auch eine Optimierung, die der besseren Überlagerung von Berechnungen und Datentransfers dient. Beim *Raytracing* wird jedes Pixel unabhängig von den anderen berechnet. Somit ist es möglich, bereits berechnete Pixel unverzüglich zum Client zurückzusenden. Dadurch findet der teilweise Rücktransport des gerenderten Bildes in einzelnen Kacheln (englisch *Buckets*) schon während der Berechnung statt. Dies spart Zeit, da am Ende nur noch ein geringer Teil des übrigen Bildes übertragen werden muss und nicht mehr das Gesamtbild. Abbildung 8.3 visualisiert den Ablauf eines Renderings. Die Optimierung ist im Interface `ISunflowRemoteAPI` folgendermaßen implementiert. Die Funktion `prepareBucketRendering(...)` implementiert die beschriebene Übertragung der Szene zur VM und liefert eine Zahl zurück. Diese Zahl gibt an, in wie viele Teile (Buckets) das Gesamtbild zur Berechnung zerlegt wurde. Die Funktion `startRendering()` startet den Rendering-Vorgang. Einzelne Buckets können dann asynchron durch Aufruf der Methode `getNextBucket()` abgerufen werden. Die Anzahl der Aufrufe dieser Funktion korrespondiert mit der Anzahl Buckets, in die das Bild zerlegt wurde. Dabei enthält das zurückgelieferte Objekt der Klasse `ReadyBucket` nicht nur das Teilbild selbst im PNG-Format[4], sondern auch dessen Abmaß und die Koordinaten der linken oberen Ecke des Teilbildes im Gesamtbild.

Die Klasse `SunflowRemoteTaskAPI` implementiert das Interface `ISunflowRemoteAPI` und ist zusätzlich von der Klasse `TaskClientPipeline` abgeleitet. Somit wird auch der Lebenszyklus der Cloud-Nutzung durch `SunflowRemoteTaskAPI` verwaltet. Der Lebenszyklus ist ebenfalls in Abbildung 8.3 visualisiert. Dazu gehören das Verbinden zu einer VM und das Installieren des benötigten Programmcodes. Da der Programmcode von *Sunflow* sehr umfangreich ist, wurde ein `CopySourceTask` gewählt, um diesen zu übertragen (siehe Kapitel 7.3.2). Die Kompilierung des Quellcodes auf Serverseite ist hier aufgrund der Menge des Codes nicht praktikabel. Das Kompilieren der 227 Quellcodedateien mit einer Gesamtgröße von ca. 1,4 MB dauert auf einer Cloud-Ressource vom Typ `c1.medium` ca. 40 s. Das Auspacken der Kompilate wird hingegen in weniger als einer Sekunde beendet. Die durchschnittlichen Zeiten für das Entpacken des 496 kB-Archivs sind in Tabelle 8.1 angegeben.

[4] *„Portable Network Graphics (PNG [...], engl. portable Netzwerkgrafik) ist ein Grafikformat für Rastergrafiken mit verlustfreier Kompression. Es wurde als freier Ersatz für das ältere, bis zum Jahr 2006 mit Patentforderungen belastete Format GIF entworfen und ist weniger komplex als TIFF. PNG unterstützt neben unterschiedlichen Farbtiefen auch Transparenz per Alphakanal. PNG ist das meistverwendete verlustfreie Grafikformat im Internet.“* (Onlineressource `http://de.wikipedia.org/wiki/Portable_Network_Graphics`(abgerufen am 5.11.2013))

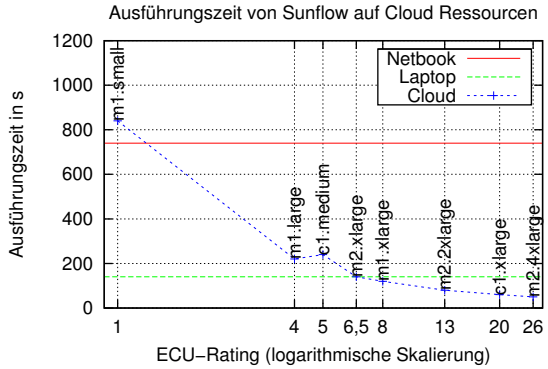


Abbildung 8.4: Renderingzeiten auf verschiedenen Mobilgeräten im Vergleich zur *Mobilen Cloud-unterstützten Anwendung*.

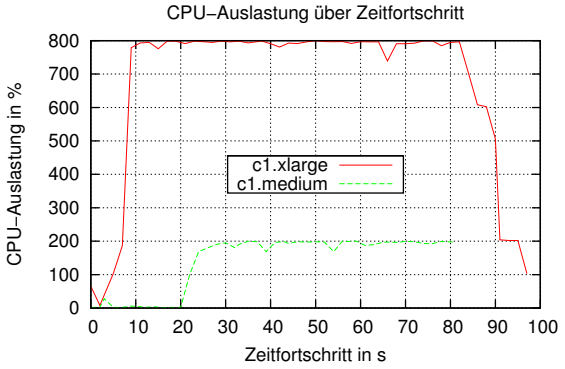


Abbildung 8.5: CPU-Auslastung auf verschiedenen VM-Typen und für verschiedene Szenen.

8.2.2 Rechenzeitbedarf

Um zunächst die Performanceverbesserung durch Auslagerung des Raytracings auf eine VM zu untersuchen, wurde eine Beispielszene aus dem Softwarepaket von *Sunflow* namens *Aliens Shiny* auf verschiedenen Ressourcen in der Auflösung 640x480 gerendert (eine Abbildung der Beispielszene ist ebenfalls in Anhang E dargestellt). Dazu wurden je eine Variante eines 10”Netbooks und eines aktuellen 15”Laptops (siehe Anhang A) als Mobilgeräte mit der Cloud-basierten Variante verglichen. Abbildung 8.4 visualisiert die Ausführungszeiten auf den Ressourcen. Die Cloud-basierte Variante wurde dabei auf dem Netbook in der Netzwerkkonfiguration HOME ausgeführt. Es ist deutlich zu erkennen, dass gegenüber der Ausführungszeit auf dem Netbook eine deutliche Beschleunigung durch die *Mobile Cloud-unterstützte Anwendung* erreicht werden kann. Dadurch lässt sich bei entsprechender Wahl des VM-Typs gegenüber der lokalen Ausführung eine wesentlich geringere Ausführungszeit erreichen. Dies gilt insbesondere für mobile Geräte mit geringer Prozessorperformance wie dem Netbook. Für aktuelle Laptops ist eventuell von Fall zu Fall zu entscheiden, ob eine entfernte Ausführung bevorzugt werden sollte. Da die Berechnung für große Bilder (beispielsweise 1920x1080) mehrere Minuten dauert, kann jedoch durch die entfernte Ausführung auf dem Laptop Akkukapazität gespart werden, was sich bei einem Laptop aufgrund der relativ hohen Leistungsaufnahme unter Last besonders positiv auswirkt. Die entfernte Ausführung ist demnach, eine ausreichende Netzwerkperformance vorausgesetzt, für große Bilder bezüglich Ausführungszeit und Energieeinsparung fast immer sinnvoll.

Abbildung 8.5 zeigt, dass die verfügbaren CPU-Kerne verschiedener VM-Instanzen gut ausgelastet werden können. Dargestellt ist jeweils die CPU-Auslastung für ein komplettes Szenenrendering, also Übertragung und Vorverarbeitung der Szene (`prepareBucketRendering(...)`) sowie das Rendering selbst. Die vierfach höhere CPU-Kernanzahl der `c1.xlarge`-Instanz kann während des Renderings gut genutzt werden. Zu erkennen ist auch, dass die Übertragung zu Beginn unterschiedlich lang dauern kann, je nach Szenenkomplexität. Hier sind verschiedene Szenen gerendert worden. Im Ergebnis ist es also günstig, unter Beachtung der zu erwartenden Kosten, möglichst performante VM-Typen zu wählen. Das beste Verhältnis aus Preis und Leistung bieten hier die `c1.medium`- und `c1.xlarge`-Instanztypen.

8.2.3 Hauptspeicherbedarf

Der tatsächliche Speicherbedarf ist in *Java* schwierig zu ermitteln, da er nicht nur von der Applikation und den Eingabedaten abhängt, sondern auch von den gewählten JVM-Startparametern bezüglich der Speicherobergrenze. Somit wurde hier ein anderer Weg gewählt. Aufgrund der Berechnungsperformance wurde der VM-Typ `c1.medium` als minimal möglicher VM-Typ aus dem Portfolio von *Amazon EC2* gewählt. Auf diesem und allen besser ausgestatteten VM-Typen sind mindestens 1,7 GB Arbeitsspeicher verfügbar. Um für das auf der VM ausgeführte Betriebssystem noch etwas Spielraum zu lassen, wurde der Speicher für die JVM auf 1,4 GB festgesetzt. Die höchste sinnvolle Auflösung wurde anhand des am Markt erhältlichen Monitors mit der höchsten Displayauflösung (4096x2160) gewählt[5].

Alle Beispielszenen konnten in dieser Auflösung und mit der beschränkten Speichermenge gerendert werden. Abschließend wird festgestellt, dass alle VM-Typen aus dem Portfolio von *Amazon EC2* bezüglich des Hauptspeicherbedarfs genutzt werden können, solange sie wenigstens 1,7 GB Arbeitsspeicher zur Verfügung stellen. Für die weiteren Betrachtungen werden demnach die bezüglich Rechengeschwindigkeit günstigen VM-Typen `c1.medium` und `c1.xlarge` benutzt.

8.2.4 Konstruktion der Laufzeitfunktion zur Vorhersage

Die eigentliche Schwierigkeit bei der Bereitstellung von *Sweethome 3D* als *Mobile Cloud-unterstützte Anwendung* ist die Prognose der Verarbeitungszeit einer gegebenen Szene auf verschiedenen Cloud-Ressourcen. Hierzu muss eine aussagekräftige Laufzeitfunktion $lauf(...)$ gefunden werden, die aufgrund von Metadaten über die Szene eine Verarbeitungszeit schätzen kann. Dazu wird zunächst für die einzelnen Methoden des entfernten Interfaces `ISunflowRemoteAPI` erläutert, aus welchen Teilen sich die Gesamtausführungszeit zusammensetzt.

- `prepareBucketRendering(...)` überträgt die relativ große Szene zur VM und liefert einen Integerwert zurück. Zudem benötigt das Dekomprimieren, Deserialisieren und Vorverarbeiten der Szenendaten auf Serverseite Zeit. Für `prepareBucketRendering(...)` muss also eine Laufzeitfunktion $lauf(...)$ erstellt werden, die auf Basis der Beschaffenheit der Szene die Ausführungszeit schätzt.

- `startRendering()` überträgt nur einen Wahrheitswert und benötigt auf Serverseite auch kaum Berechnungszeit, da nur das Rendering gestartet wird. Somit ist `startRendering()` sowohl für die Kommunikations- als auch für die Ausführungszeit nicht interessant.

- `getNextBucket()` wird häufig aufgerufen und überträgt kleine Bildausschnitte zurück zum Client. `getNextBucket()` selbst benötigt keine Berechnungszeit auf Serverseite. Jedoch findet parallel zur Bearbeitung das eigentliche Rendering statt. Für $lauf(...)$ ist demnach von Bedeutung, wie lange das Rendering aufgrund der Beschaffenheit der Szene dauert. Zusätzlich ist für die Pixelanzahl des Ergebnisbildes die Gesamtdatenmenge zu bestimmen, um daraus die Kommunikationszeit abzuschätzen. Dabei kann davon ausgegangen werden, dass die Einzelbilder der Buckets zusammen nicht deutlich größer sind als das Gesamtbild in PNG-Codierung.

[5] Das Modell mit der höchsten Displayauflösung, welches in Deutschland verfügbar ist, ist ein *Eizo DuraVision FDH3601 schwarz, 36.4"*. Die Auflösung beträgt 4096x2160 Bildpunkte. Bestimmt wurde das Modell unter allen auf `http://geizhals.at/` am 8.11.2013 in der EU gelisteten Produkten in der Kategorie Monitor.

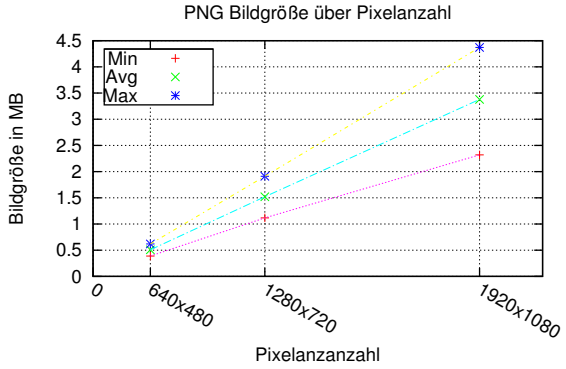


Abbildung 8.6: Approximation der Größe eines PNG-Bildes.

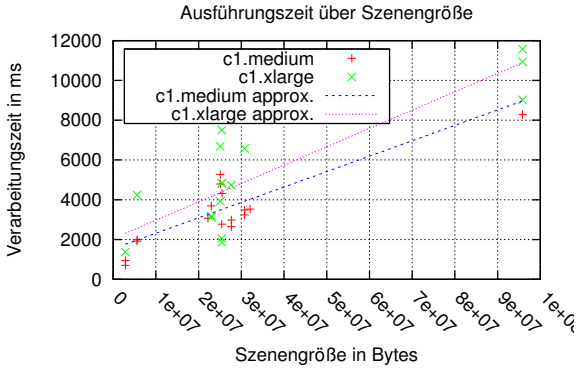


Abbildung 8.7: Ausführungszeit von `prepareBucketRendering(..)` und Approximation.

Zusammenfassend wird die Ausführungszeit maßgeblich durch die sequentielle Ausführung von `prepareBucketRendering(...)` und dem eigentlichen Rendering bestimmt. Die Kommunikationszeit setzt sich zusammen aus der Upload-Zeit für die Szene und der Download-Zeit für das gerenderte Bild. Die genaue Konstruktion der Approximationsmethoden wird nachfolgend durchgeführt. Bekannt sind für die Konstruktion von $lauf(...)$ nur die Szenenparameter, die Szenengröße für den Upload und die Pixelanzahl des zu erstellenden Bildes. Die Größe des resultierenden PNG-Bildes muss anhand der Pixelanzahl geschätzt werden, da sie vor dem Rendering im Gegensatz zur Szenengröße nicht bekannt ist. Die Größe ist zur Schätzung der Downloadzeit relevant.

Schätzung der Größe von PNG-Bildern

Abbildung 8.6 zeigt die Bildgrößen in kB für die verschiedenen Beispielszenen und verschiedene Bildgrößen. Es existiert eine Schwankung, da die Größe auch stark vom Bildinhalt abhängt. Dargestellt sind jeweils die minimalen, maximalen und durchschnittlich gemessenen Werte von jeweils 11 Bildern in den Auflösungen 640x480, 1280x720 und 1920x1080. Es wurden Bilder der sieben offiziellen Beispielszenen zu *Sweethome 3D* und der vier selbst konstruierten Szenen benutzt (siehe Anhang E). Aus der Abbildung lässt sich erkennen, dass die Bildgröße linear mit der Pixelanzahl steigt. Dabei wurde für die mittlere Bildgröße via Regression ein Anstieg von 1,7 bestimmt. Daraus ergibt sich folgende Gleichung zur Berechnung für eine Pixelanzahl $\#Pixel > 0$.

$$Bildgröße \quad = \quad \#Pixel \cdot 1,7 + 9988. \tag{8.1}$$

Diese Berechnung wird genutzt, um anhand der Pixelanzahl die Größe des zu übertragenden fertigen PNG-Bildes zu schätzen. Die maximale Abweichung der Größe der 33 Testbilder von der jeweiligen Schätzung lag bei ca. 31 %.

Schätzung der Ausführungsdauer für `prepareBucketRendering(...)`

Für die Bestimmung der Laufzeitfunktion *lauf(...)* ist es notwendig, einen sinnvollen Zusammenhang zwischen einer Eingabegröße und der Verarbeitungszeit zu finden.

Die Ausführungszeit von `prepareBucketRendering(...)` hängt maßgeblich von der Komplexität der Szene ab. Das heißt, es wird vermutet, dass die Ausführungszeit mit der Szenengröße in Byte (e_{Szene}) korreliert. Abbildung 8.7 zeigt die gemessenen Werte der Ausführungszeit für alle Trainingsszenen auf verschiedenen VM-Typen. Durch die gemessenen Werte wurde eine lineare Funktion $lauf_{pbr}(\dots)$ approximiert, die aus einer gegebenen Szenengröße die zu erwartende Verarbeitungszeit des `PrepareBucketRenderingTask` auf Serverseite in Abhängigkeit der verwendeten Ressource R schätzt. Die approximierten Laufzeitfunktionen für die beiden VM-Typen haben folgende Gestalt:

$$lauf_{pbr}(R_{c1.medium}, e_{Szene}) = e_{Szene} \cdot 0,0000000775101 + 1,540, \quad (8.2)$$

$$lauf_{pbr}(R_{c1.xlarge}, e_{Szene}) = e_{Szene} \cdot 0,0000000922838 + 2,048. \quad (8.3)$$

Dabei wurden für das Güteverhältnis $V_{Q_{gut}}$, welches die Prozentzahl von Messwerten beschreibt, die um maximal 50% von der Schätzung abweichen, Werte zwischen 91–96 % bestimmt. Somit wird das Gütekriterium $K_{Q_{gut}}$ eingehalten (siehe Kapitel 6.2.4).

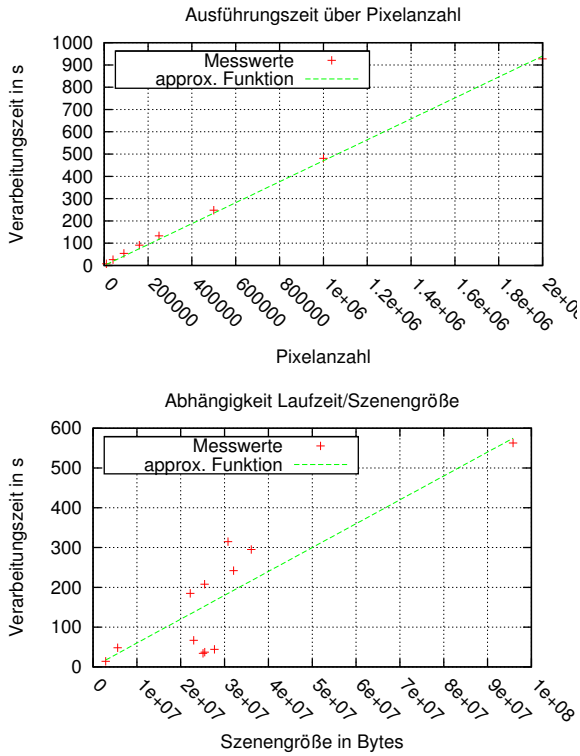
Schätzung der Ausführungsdauer für das Rendering

Die Ausführungszeit des Raytracings wird maßgeblich von zwei Faktoren bestimmt:

- der Beschaffenheit/Komplexität der Szene und
- der Größe (Pixelanzahl) des resultierenden Bildes.

Zur Findung eines Zusammenhangs zwischen der Ausführungszeit und diesen Parametern muss eine multiple Regression durchgeführt werden (siehe Kapitel 6.2.3). Bei der Analyse der eigentlichen Berechnung wurde vermutet, dass das Rendering sehr gut mit der Anzahl der zu berechnenden Pixel skaliert. Abbildung 8.8 (links oben) bestätigt diesen Zusammenhang. Abgebildet ist die Renderingzeit für jeweils die gleiche Szene unter Verwendung verschiedener Bildgrößen.

Schließlich musste für die weitere Beschreibung der Laufzeitfunktion $lauf_{render}(...)$ noch ein weiterer Parameter gefunden werden, der eine Korrelation zwischen den Parametern einer Szene und der Ausführungszeit der Berechnung herstellt. Vor dem eigentlichen Rendering stehen nur begrenzt Informationen über die Szene zur Verfügung. Gemessen werden können nur die Anzahl von Objekten (Instanzen), die Anzahl von Dreiecken (Primitive) und die Szenengröße im Arbeitsspeicher. Die Szenengröße berücksichtigt dabei auch Texturen. Abbildung 8.8 zeigt zwei der gemessenen Abhängigkeiten in Bezug zu verschiedenen Eingabegrößen. Dafür wurden die Ausführungszeiten von 12 verschiedenen Szenen in der Auflösung 640x480 bestimmt. Auch hier wurden dafür die sieben offiziellen Beispielszenen zu *Sweethome 3D*, die vier selbst erstellten Testszenen und zusätzlich noch die Szene „extern21“ benutzt (siehe Anhang E). Daraus lässt sich erkennen, dass eine direkte Korrelation kaum zu finden ist. Einzig eine Abhängigkeit zwischen der Szenengröße und der Ausführungszeit lässt sich vermuten. In Abbildung 8.8 ist zusätzlich die Güte der Approximation in Form des Verhältnisses $V_{Q_{gut}}$ dargestellt. Die Szenengröße liefert den besten Wert. Zur Anzahl Instanzen konnte kein Zusammenhang ermittelt werden ($V_{Q_{gut}} = 0$), weshalb dieser auch nicht abgebildet ist. Da die Szenengröße (e_{Szene}) bereits zur Vorhersage der Ausführungszeit von `prepareBucketRendering(...)` benutzt wird, wurde abschließend festgelegt, dass diese ebenfalls für $lauf_{render}(\dots)$ benutzt wird. Um die Abhängigkeit der Ausführungszeit von der



Qualität der approximierten Funktion für das Rendering

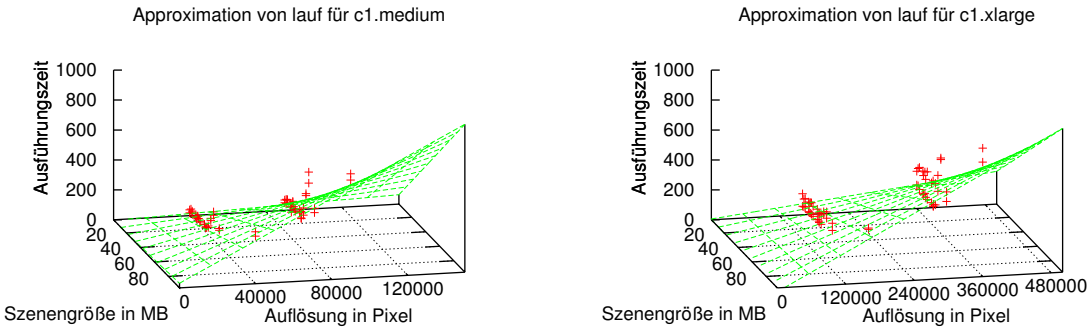
unabhängige Variable	$V_{Q_{gut}}$
Szenengröße	ca. 58 %
Anzahl Primitive	ca. 42 %

Abbildung 8.8: Visualisierung möglicher Korrelationen der Renderingzeit zu verschiedenen Eingabegrößen. (In der Abbildung oben links ist die Szene konstant, sonst wurden je 12 verschiedene Szenen in der Auflösung 640x480 erstellt.)

Bildgröße ($\#Pixel$) und von der Szenengröße (e_{Szene}) zu reflektieren, wird für $lauf_{render}(\dots)$ der Eingabeparameter e_{Render} aus Szenengröße und Pixelanzahl konstruiert:

$$e_{Render} \quad = \quad (\#Pixel, e_{Szene}). \tag{8.4}$$

Nach der Festlegung der Eingabegröße für die Laufzeitfunktion $lauf_{render}(...)$ wurden Messungen für die konfigurierten VM-Typen durchgeführt. Dazu wurden die 16 Trainingsszenen in zwei Auflösungen jeweils zweimal berechnet. Für `c1.medium` wurden die Auflösungen 200x200 und 300x300 verwendet. Für die `c1.xlarge`-Instanzen wurden die Auflösungen 400x400 und 600x600 benutzt. Aus diesen je 64 Messwerten wurde dann durch lineare Approximation für jeden VM-Typ eine Funktion erstellt, die für eine Eingabe die entsprechende Ausführungszeit schätzt. Obwohl zwischen den Koeffizienten e_{Szene} und $\#Pixel$ prinzipiell keine kausale Abhängigkeit besteht, lieferte die multiple Regression unter Verwendung des zusätzlichen Koeffizienten $e_{Szene} \cdot \#Pixel$ deutlich bessere Ergebnisse. Somit wurde die multiple Regression, wie in Kapitel 6.2.3 beschrieben, mit diesem Koeffizienten durchgeführt. Die Messpunkte und die approximierten Flächen sind in Abbildung 8.9 zu sehen. Die approxi-



$V_{Q_{gut}}$ ca. 68 %

$V_{Q_{gut}}$ ca. 78 %

Abbildung 8.9: Approximation der Laufzeitfunktion $lauf_{render}(...)$ für Sunflow auf verschiedenen VM-Typen.

mierten Schätzfunktionen haben folgende Gestalt:

$$\begin{aligned} lauf_{render}(R_{\text{c1.medium}}, e_{Render}) &= 0,0000003753 \cdot e_{Szene} + 0,0003284 \cdot \#Pixel \\ &\quad +0,00000000006309 \cdot e_{Szene} \cdot \#Pixel \\ &\quad -4,151, \end{aligned} \tag{8.5}$$

$$\begin{aligned} lauf_{render}(R_{\text{c1.xlarge}}, e_{Render}) &= -0,0000003343 \cdot e_{Szene} + 0,0003957 \cdot \#Pixel \\ &\quad +0,00000000001644 \cdot e_{Szene} \cdot \#Pixel \\ &\quad +6,947. \end{aligned} \tag{8.6}$$

Für die verwendeten Trainingsszenen lässt sich erkennen, dass die Approximation allein für das Rendering unter Verwendung der Szenengröße und der Bildauflösung das Gütekriterium $K_{Q_{gut}}$ nicht erfüllen kann. Jedoch wird es wenigstens für die `c1.xlarge`-Instanz mit ca. 78 % fast erreicht. Da es sich bei dem Raytracing jedoch um eine eingabeterminierte Applikation handelt, deren Gesamtausführungszeit sich aus mehreren sequentiellen Einzeloperationen zusammensetzt, ist es nicht notwendig, dass alle approximierten Funktionen zur Tasklaufzeit das Gütekriterium $K_{Q_{gut}}$ erfüllen. Es reicht aus, wenn die Gesamtsumme $K_{Q_{gut}}$ einhält. Da also $lauf_{pbr}$ bereits über 90 % für $V_{Q_{gut}}$ erreicht hat, können die niedrigeren Werte für $lauf_{render}$ toleriert werden.

Berechnung der Gesamtausführungszeit

Die Gesamtausführungszeit für das Raytracing wird abschließend als $lauf_{raytracing}(\dots)$ angegeben und ergibt sich aus der Summe von $lauf_{pbr}(\dots)$ und $lauf_{render}(\dots)$. Für die Gesamtlaufzeitfunktion $lauf_{raytracing}(\dots)$ wird ebenfalls der Parameter e_{Render} benutzt (siehe Gleichung (8.4)), da dieser alle für die Ausführung der Teilfunktionen notwendigen Parameter enthält. Die Laufzeitfunktion $lauf_{raytracing}(\dots)$ ist wie folgt definiert:

$$lauf_{raytracing}(e_{Render}) = lauf_{pbr}(e_{Szene}) + lauf_{render}(e_{Render}). \tag{8.7}$$

Diese Schätzfunktion wird anschließend auf dem Broker als Applikationsprofil für *Sweethome 3D* installiert. Die Auswertung der Güte der Laufzeitvorhersage für die Testszenen wird im nachfolgenden Abschnitt betrachtet.

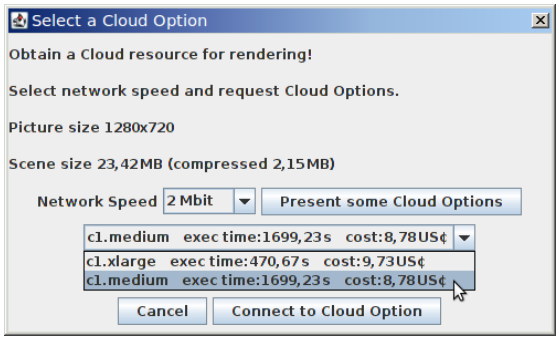


Abbildung 8.10: Auswahldialog zur Bestimmung von Cloud-Optionen und zur Allokation von VM-Instanzen.

Tabelle 8.5: Güte der Schätzung für verschiedene VM-Typen und Netzwerke

		`c1.medium`	`c1.xlarge`
UMTS	Abw.* Mittelwert	ca. 34 %	ca. 29 %
	$V_{Q_{gut}}$	0,83	0,84
HOME	Abw. Mittelwert	ca. 29 %	ca. 27 %
	$V_{Q_{gut}}$	0,86	0,94

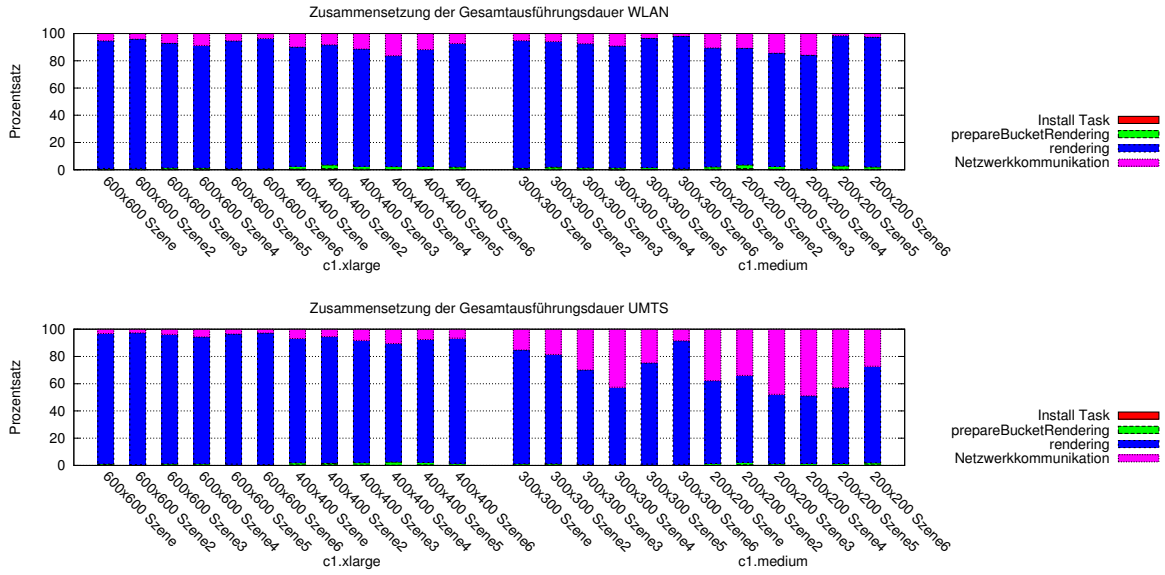
* Abw. = mittlere Abweichung der 32 Messungen von der Vorhersage

8.2.5 Bewertung der Auswahlstrategie

Zur Bewertung der Auswahlstrategie wurden unter Verwendung der erstellten Laufzeitvorhersagemethoden mit den verbleibenden 16 Testszenen Messungen mit unterschiedlichen Kommunikationsverbindungen durchgeführt. Als mobiles Testgerät kam erneut das bereits in Abschnitt 8.2.2 benutzte 10" Netbook zum Einsatz. Es wurden je zwei verschiedene Auflösungen auf den konfigurierten VM-Typen `c1.medium` und `c1.xlarge` unter Verwendung einer der Netzwerkkonfigurationen HOME und UMTS gerendert. Auf den `c1.medium`-Typen wurden die Auflösungen 200x200 und 300x300 verwendet, was zu Gesamtausführungszeiten zwischen ca. 0,5–20 min führte. Auf dem `c1.xlarge`-Instanzen wurden die Auflösungen 400x400 und 600x600 gerendert, was in Ausführungszeiten zwischen ca. 1–15 min resultierte. Die je 64 Kombinationen aus Netzwerkkonfiguration, VM-Typ und Testszene ergaben unter Verwendung der je zwei verschiedenen Bildauflösungen genau 128 Messwerte.

Zunächst wurde die momentan verfügbare Netzwerkperformance mit Hilfe des *SPEEDTEST* Widgets ermittelt, welches auf dem Broker installiert ist (siehe Kapitel 7.2.1). Auf deren Basis wurde dann unter Verwendung des Auswahldialogs in Abbildung 8.10 die Ausführungszeit geschätzt, eine Ressource allokiert und anschließend das Rendering gestartet. In der Abbildung ist auch zu sehen, dass für eingabeterminierte Applikationen die Gesamtausführungskosten durch die Cloud-Optionen zurückgeliefert werden. Hier bekommt der Nutzer die Möglichkeit, zwischen einer schnelleren Bearbeitung oder geringeren Kosten zu wählen. Tabelle 8.5 fasst die Güte der je 32 Einzelmessungen für die vier Kombinationen zusammen. Angegeben ist das Verhältnis $V_{Q_{gut}}$ aller Messungen, die nicht mehr als 50 % von der Schätzung abweichen, zur Gesamtzahl Messungen. Zusätzlich ist auch die mittlere Abweichung vom Vorhersagewert angegeben. Die erreichten Werte zeigen, dass das Gütekriterium $K_{Q_{gut}}$ für die Vorhersage mit $lauf_{raytracing}(\dots)$ insgesamt eingehalten wird, obwohl für diese eingabeterminierte Applikation $K_{Q_{gut}}$ nicht für alle einzelnen Berechnungsschritte erfüllt wird. Eine sinnvolle Auswahl der Ressource ist dadurch für diese Applikation möglich.

Die tatsächliche Zusammensetzung der Gesamtausführungszeit verschiedener Messungen ist in Abbildung 8.11 dargestellt. Jeder Balken im Diagramm repräsentiert eine Messung. Die Zusammensetzung ist dabei fast immer ähnlich. Der Anteil für die Vorbereitung der Szene (`prepareBucketRendering(...)`) seitens der VM und die Installation des Programmcodes



Szene ≙ SweetHome3DExample, Szene2 ≙ SweetHome3DExample2, usw. (siehe Anhang E)

Abbildung 8.11: Gemessene prozentuale Zusammensetzung der Gesamausführungszeit für das Rendering verschiedener Szenen in unterschiedlichen Konfigurationen.

ist in der Gesamtausführungszeit sehr gering. Der Anteil der Netzwerkkommunikation variiert von Szene zu Szene. Auch die verwendete Kommunikationstechnologie hat einen teils großen Einfluss. Die Messungen für UMTS sollten den Messungen für WLAN ähnlich sein, jedoch stand bei den Messungen unter Verwendung einer `c1.medium`-Instanz in der UMTS-Konfiguration weniger Bandbreite zur Verfügung als bei den Messungen unter Verwendung von WLAN. Hier wird erneut deutlich, dass die verfügbare Netzwerkperformance für die aktuelle Situation stets neu bestimmt werden sollte.

Da die Gesamtausführungszeit sehr gut mit der Bildauflösung skaliert, kann der Vorhersagefehler begrenzt werden, indem zunächst in einer kleineren Auflösung gerendert und die benötigte Zeit mit der Vorhersage verglichen wird. Dadurch erhält man einen guten Anhaltspunkt über die Güte der Schätzung und kann so die Ausführungszeit für das Rendering eines größeren Bildes im Vergleich zur Vorhersage besser bewerten.

8.3 Mobile Streaming-Bildverarbeitung

Als benutzerterminierte Applikation wurde eine *Android*-App erstellt, die fortlaufend Kamerabilder verarbeitet. Jede Verarbeitung eines Bildes wird dabei durch einen Task repräsentiert. Das Ausgangsbild dient als Eingabe und das bearbeitete Bild als Ausgabe. Da die Dauer der Programmausführung nicht bekannt ist und durch den Benutzer gesteuert wird, handelt es sich um eine benutzerterminierte Applikation (siehe Kapitel 6.1.1), für die eine Bildverarbeitungsrate pro Sekunde (englisch *frames per second* (fps)) angegeben werden kann. Für die eigentliche Verarbeitung der Bilder wurden zwei verschiedene Algorithmen implementiert.

Abbildung 8.12: Abbildung der Streaming-Bildverarbeitungsapplikation für Android auf dem DELL Streak 7. (*Links:* Gesichtserkennungsalgorithmus *Rechts:* Kantenerkennungsalgorithmus)

8.3.1 Algorithmen

Bei der erstellten *Android*-App kann zwischen einer lokalen Verarbeitung und einer entfernten Cloud-unterstützten Verarbeitung der Kamerabilder umgeschaltet werden. Somit ist der Algorithmus eventuell auch ohne verfügbare Netzwerkverbindnung noch lokal verwendbar. Bevor die Algorithmen näher beschrieben werden, sei noch darauf hingewiesen, dass *Java SE* für PCs und Laptops und die *Android*-Implementierung verschiedene Systembibliotheken verwenden. Dies führt dazu, dass unterschiedliche Klassen zur Abspeicherung von Bilddaten auf dem Mobilgerät und der VM benutzt werden müssen. Unter *Android* wird `android.graphics.Bitmap` und unter *Java SE* `java.awt.image.BufferedImage` verwendet. Als Ergebnis ist es nicht direkt möglich, denselben Code auf *Android*- und VM-Seite zur Bearbeitung der Algorithmen zu verwenden.

Es wurden für Testzwecke zwei verschiedene Bildverarbeitungsalgorithmen implementiert.

Sobel Kantenerkennung

Mit Hilfe des *Sobel*-Operators werden scharfe Kantenübergänge in einem Bild sichtbar gemacht (siehe [NA08, S. 123ff.]). Ein Beispiel unter Verwendung der vorgestellten Applikation ist in Abbildung 8.12 (rechts) zu sehen. Die Bildtransformation wurde auf Basis der Software *ImageProx*[6] implementiert. Der Programmcode für die lokale und die entfernte Ausführung unterscheiden sich in diesem Beispiel nur für die bereits beschriebenen verschiedenen Speicherformate für Bilder unter *Android* und der *Java SE*.

Gesichtserkennung

Als weiterer Bildverarbeitungsalgorithmus wurde ein Gesichtserkennungsalgorithmus auf Basis von Hautfarbenerkennung implementiert [HAMJ02]. Ein Beispiel unter Verwendung der vorgestellten Applikation ist in Abbildung 8.12 (links) zu sehen. Hierzu wurde auf dem Quellcode des Programms *Face Detection in Color Images*[7] aufgebaut. Der Gesichtserkennungsalgorithmus benötigt viele Verarbeitungsschritte und große Datenmengen an Zwischenergebnis-

[6] Verfügbar unter Onlineressource `http://www.3programmers.com/mwells/research.html` (abgerufen am 19.2.2014)

[7] Verfügbar unter Onlineressource `http://sourceforge.net/projects/facedetectionin/` (abgerufen am 19.2.2014)

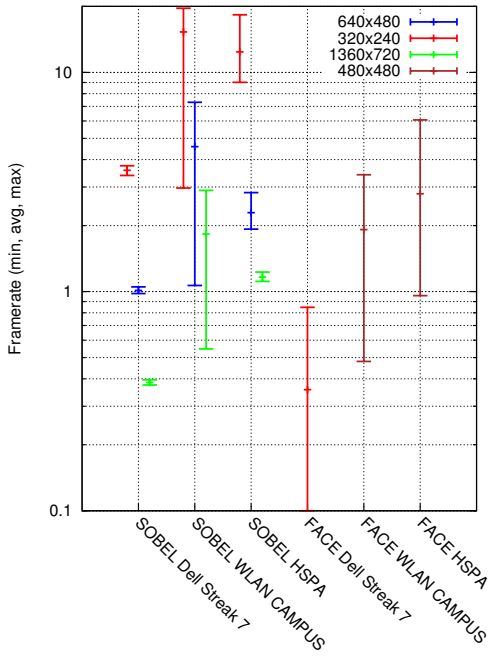


Abbildung 8.13: Verarbeitungsraten für Bildverarbeitungstasks der Streaming-Bildverarbeitung.

Tabelle 8.6: Charakteristiken von Bildverarbeitungstasks für Streaming-Bildverarbeitung

	Upload	Download	Δt_{mobil} (lokal)	Δt_{work} (`c1.medium`)
320x240 (Sobel)	ca. 7 kB	ca. 12 kB	280 ms	25 ms
640x480 (Sobel)	ca. 22 kB	ca. 29 kB	990 ms	80 ms
1360x720 (Sobel)	ca. 54 kB	ca. 85 kB	2500 ms	250 ms
320x240 (Face)	–	–	2860 ms	–
480x480 (Face)	ca. 20 kB	ca. 21 kB	–	1000 ms

sen müssen gespeichert werden. Somit ist die Gesichtserkennung auf dem verfügbaren Tablet und Smartphone in ihrer ursprünglichen Form selbst für die geringste Kamerbildauflösung von 176x144 nicht ausführbar.

Um dennoch eine auf den Mobilgeräten ausführbare Variante zu erhalten, wurde der Algorithmus um einige Analyseschritte beschnitten. So werden in der vereinfachten Variante alle Hautpartien schon als Gesicht erkannt, ohne tatsächlich eine Überprüfung bezüglich der Erkennung der Gesichtsgeometrie durchzuführen. Diese Variante ist zwar ungenau, liefert aber auch das eigentliche Gesicht zurück und lässt sich auf dem Dell Streak 7 Tablet beispielsweise noch in einer Auflösung bis 320x240 ausführen.

8.3.2 Rechenzeitbedarf

Zunächst wurde die Geschwindigkeit der Ausführung der Bildverarbeitungstasks auf dem Dell Streak 7 Mobilgerät bestimmt und mit der auf eine VM ausgelagerten Variante verglichen, um die erreichbaren Verbesserungen zu zeigen. Abbildung 8.13 visualisiert die Ergebnisse. Dargestellt sind jeweils die minimale, maximale und durchschnittliche Bildverarbeitungsrate für einen Messzeitraum von ca. 100 s. Die nebenstehende Tabelle enthält zudem die Datengrößen für die übermittelten Bilder und die gemessenen durchschnittlichen Verarbeitungszeiten auf dem Gerät und auf der VM. Für die Performancemessungen zur Kantenerkennung wurde eine `c1.medium`-Instanz genutzt, für die Gesichtserkennung hingegen eine `c1.xlarge`-Instanz.

Abbildung 8.13 zeigt, dass die Kantenerkennung als *Mobile Cloud-unterstützte Anwendung*

deutlich beschleunigt werden kann. Interessant ist hier zudem, dass die Variante unter Verwendung der WLAN-Konfiguration CAMPUS wesentlich größere Schwankungen, vor allem nach unten, aufweist. Wie bereits in Kapitel 4.2 erläutert wurde, hängt dies damit zusammen, dass sich hier alle WLAN-Geräte im gleichen Netzwerk das Übertragungsmedium dynamisch teilen. Dies kann zu Störungen führen, die die Übertragungsrate kurzzeitig stark vermindern. Bei UMTS/HSPA wird jedem Teilnehmer hingegen eine Bandbreite fest zugeteilt, die er dann auch verwenden kann, ohne dabei andere Teilnehmer zu stören.

Für die Bewertung des Gesichtserkennungstasks wurde nur eine Auflösung von 480x480 gewählt. Bei dieser Auflösung benötigt ein Task ca. 1 s auf einer `c1.xlarge` oder `c1.medium` VM, siehe Tabelle 8.6. Diese Dauer ist als Verzögerung im Bildfluss erkennbar. Bei höherer Auflösung würde sich diese Verzögerung noch erhöhen und die Applikation in der Anwendbarkeit einschränken. Bei niedrigeren Auflösungen kann es sein, dass die Gesichtserkennung nicht mehr zufriedenstellend funktioniert. In Abbildung 8.13 ist zudem zu erkennen, dass die *Mobile Cloud-unterstützte Anwendung* im Mittel zwischen 2–3 fps erreicht. Für diese Variante ist eine Anwendbarkeit des Algorithmus somit gewährleistet. Die Unterschiede zwischen der HSPA-Verbindung und der WLAN-Verbindung sind hier gering und eher auf die unterschiedliche Beschaffenheit der Bilder bezüglich der Anzahl von Menschen/Gesichtern zurückzuführen als auf Performanceunterschiede zwischen den mobilen Kommunikationsverbindungen.

8.3.3 Konstruktion der Laufzeitfunktion zur Vorhersage

Das Ziel der vorgestellten Middleware ist es, einen passenden VM-Typ für die entfernte Ausführung zur Laufzeit zu bestimmen. Um die dafür notwendigen Applikationsprofile zu erstellen, muss zunächst für jede der beiden Testapplikationen zur Kantenerkennung (`SobelTask`) und zur Gesichtserkennung (`FaceTask`) eine Laufzeitfunktion $rate_{multi}(\dots)$ erstellt werden. Im Gegensatz zu der in Kapitel 8.2 vorgestellten Beispielapplikation handelt es es sich hierbei um eine benutzerterminierte Applikation, deren vollständige Eingabe also zum Beginn der Ausführung nicht feststeht. Deshalb wird hier die Verarbeitungszeit eines einzelnen Tasks geschätzt und daraus eine Rate berechnet.

Zunächst wurden die Applikation und beide Bildverarbeitungstasks auf verschiedenen VM-Typen und unter Verwendung verschiedener Auflösungen mehrfach ausgeführt. Danach erfolgte die Auswertung der gesammelten Logdaten. Insgesamt wurden die Laufzeitfunktionen $rate_{multi}$ für beide Tasks aus 9497 Messwerten generiert. Die Auswahl konfigurierter VMs bestand aus den Typen `m1.small`, `c1.medium` und `c1.xlarge`, da sich diese, wie in Kapitel 6.2.5 gefordert, in der Ausführungsgeschwindigkeit jeweils um circa den Faktor 4 unterscheiden. Da hier für viele gleichartige Tasks mit derselben Eingabeauflösung verschiedene Messungen vorlagen, wurde jeweils der Mittelwert gebildet. Abbildung 8.14 zeigt die Häufigkeitsverteilung (Histogramm) der gleichartigen Tasks an. Für die Kantenerkennung (links) reflektiert der Mittelwert sichtbar auch den Großteil der Werte. Bei der Gesichtserkennung (rechte Abbildung) ist der Mittelwert durch eine etwas nach rechts verschobene Verteilung aller Messungen vom Median deutlich verschieden (Mittelw.: 2253 ms Median: 1525 ms). Deshalb wurde der Mittelwert hier nur aus den Werten aus dem Interquartilsabstand ($\pm$ 25 % der Werte um den Median) gebildet. Dass diese Anpassung sinnvoll sein kann, wurde in Kapitel 6.2.4 zur Regressionsanalyse erläutert. Der angepasste Mittelwert (1645 ms) ist eher typisch für den Großteil der Werte (siehe Abbildung 8.14 (rechts)).

Abbildung 8.15 zeigt das Ergebnis der Regressionsanalyse für beide Bildverarbeitungstasks für die durchschnittliche Ausführungszeit. Es ist gut zu erkennen, dass ein Zusammenhang

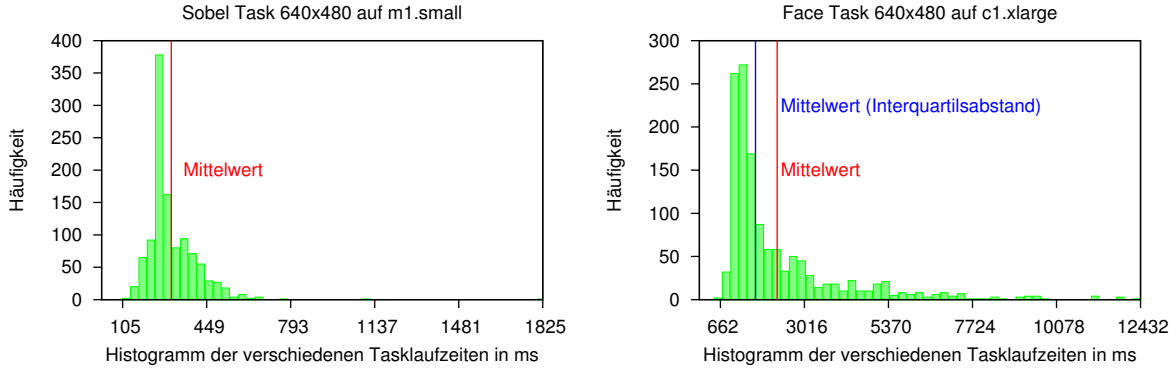


Abbildung 8.14: Häufigkeitsverteilung verschiedener Tasklaufzeiten gleicher Art (Taskart und Bildauflösung) aus den Logdaten mehrerer Testläufe der Cloud-unterstützten Streaming-Bildverarbeitung für Mobilgeräte.

zwischen der Pixelanzahl des Bildes und der Ausführungszeit besteht. Beide Taskausführungszeiten lassen sich gut durch eine lineare Funktion approximieren. Jeder Punkt im Diagramm stellt den Mittelwert aus minimal 34 bis zu 3067 Einzelmessungen dar. Es wurden die Auflösungen 320x240, 480x480 und 640x480 getestet. Außerdem ist den Digrammen auch zu entnehmen, dass die Approximationsgeraden für `c1.medium` und `c1.xlarge` sehr ähnlich sind. Dies ist darauf zurückzuführen, dass diese VM-Typen die gleiche Einzelprozessorperformance aufweisen und alle Bildverarbeitungstasks Einzelprozessortasks sind.

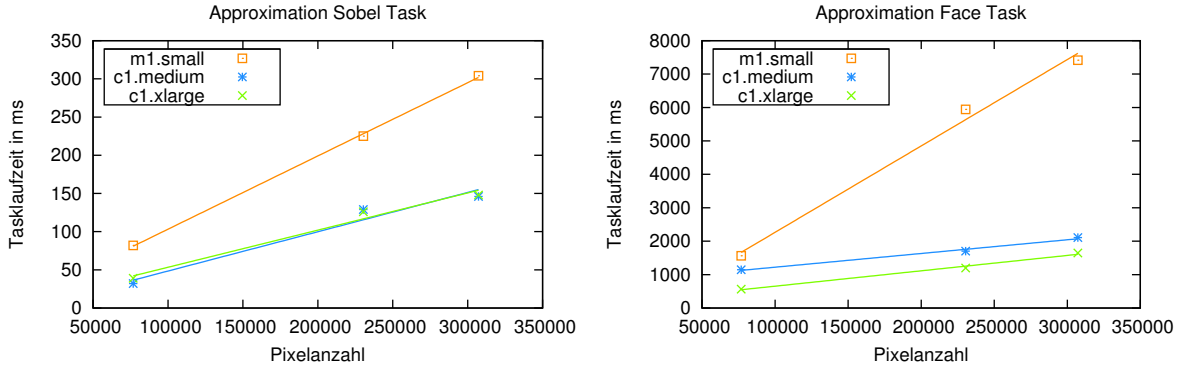
Es wurden nachfolgende Approximationsgeraden ermittelt:

$$
\begin{aligned}
lauf_{face}(R_{m1.small}, \#Pixel) &= 0,02585 \cdot \#Pixel - 320,6,\\
lauf_{face}(R_{c1.medium}, \#Pixel) &= 0,004103 \cdot \#Pixel + 811,3,\\
lauf_{face}(R_{c1.xlarge}, \#Pixel) &= 0,004600 \cdot \#Pixel + 193,6,\\
lauf_{sobel}(R_{m1.small}, \#Pixel)) &= 0,0009589 \cdot \#Pixel + 7,286,\\
lauf_{sobel}(R_{c1.medium}, \#Pixel)) &= 0,0005143 \cdot \#Pixel - 3,\\
lauf_{sobel}(R_{c1.xlarge}, \#Pixel)) &= 0,0004864 \cdot \#Pixel + 4,714.
\end{aligned}
$$

Abschließend wurde für alle 9497 Messwerte die Qualität der Vorhersage in Form des Wertes Q_{gut} bestimmt. In der Tabelle in Abbildung 8.15 ist das Verhältnis $V_{Q_{gut}}$ aller positiv vorhergesagten Werte zur Gesamtzahl der Messungen für eine Kombination aus VM-Typ und Algorithmustyp dargestellt. Dabei verfehlt nur die Kantenerkennung auf dem VM-Typ `m1.small` und die Gesichtserkennung auf dem VM-Typ `c1.xlarge` das Gütekriterium $K_{Q_{gut}}$ von 80 %. Im Vergleich zu den anderen Werten ist es dennoch möglich, alle drei VM-Typen für die Kantenerkennung und für die Gesichtserkennung zu konfigurieren, da eine gut abgegrenzte Vorhersage immer noch möglich ist.

8.3.4 Bewertung der Auswahlstrategie

Nach dem Konfigurieren der Applikationen auf dem Broker-Server, wurden insgesamt 36 Tests durchgeführt, um die vorhergesagte Performance und die tatsächlich erreichte Performance



$V_{Q_{gut}}$ aller Einzeltasks

	m1.small	c1.medium	c1.xlarge
FaceTask	0,82	0,8	0,76
SobelTask	0,76	0,92	0,95

Abbildung 8.15: Regressionsanalyse der durchschnittlichen Ausführungszeit $\Delta t_{\text{task cloud}}$ für den Kantenerkennungs- und den Gesichtserkennungstask auf verschiedenen *Amazon EC2*-Instanztypen.

miteinander zu vergleichen. Der Test bestand aus 24 Messungen unter Verwendung von 3G Mobilfunk in der UMTS/HSPA-Konfiguration und weiteren 12 Messungen, die unter Verwendung von WLAN in der HOME-Konfiguration und der CAMPUS-Konfiguration durchgeführt wurden. Die genaue Beschreibung der einzelnen Konfigurationen ist in Tabelle 4.1 zu finden. Als Testgerät kam das Sony Xperia P Smartphone zum Einsatz (siehe Anhang A). Die Ermittlung der momentanen Netzwerkperformance wurde erneut mit dem *SPEEDTEST* Widget durchgeführt, welches auf dem Broker installiert ist (siehe Kapitel 7.2.1).

Jeder einzelne Test wurde folgendermaßen durchgeführt:

1. Bestimmen der momentanen Netzwerkperformance mittels *SPEEDTEST*,
2. Auswählen des Bildverarbeitungstasks und der Displayauflösung,
3. Senden der Anfrage an Broker und Berechnung von Cloud-Optionen,
4. Auswählen der passenden Cloud-Option und Warten auf deren Bereitstellung,
5. Aufnehmen einer Verbindung zur VM und Installation des Programmcodes,
6. Ausführen der Applikation für mindestens eine Minute für aussagekräftige Bildraten,
7. Stoppen der Applikation und Trennen der Verbindung zur VM.

Der Broker berechnet verschiedene Cloud-Optionen auf Basis der gemessenen Netzwerkperformance, des ausgewählten Bildverarbeitungstasks und der ausgewählten Auflösung. Die zurückgelieferten Cloud-Optionen reflektieren dann die erreichbare Bildverarbeitungsrate auf verschiedenen VM-Typen. Ein Beispiel für solche Cloud-Optionen ist in Abbildung 8.16 zu

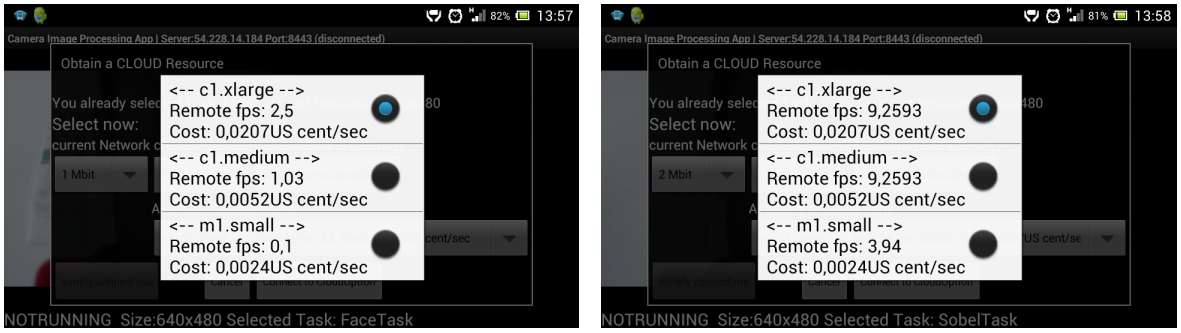


Abbildung 8.16: Auswahldialog für *Cloud-Optionen* für verschiedene Bildverarbeitungstasks unter Verwendung einer Auflösung von 640x480. (*Links:* Gesichtserkennung, c1.xlarge sollte gewählt werden *Rechts:* Sobel Kantenerkennung, c1.medium sollte gewählt werden)

sehen. Hier wurde jeweils der Auswahldialog für verfügbare Cloud-Optionen für beide Bildverarbeitungstasks unter Verwendung einer Auflösung von 640x480 dargestellt, wie er in der Applikation für Streaming-Bildverarbeitung implementiert ist. Ein Benutzer kann dann anhand der gewünschten Bildwiederholfrequenz oder eventuellen Kostenbeschränkungen auswählen.

Es sollte die Cloud-Option mit der höchsten prognostizierten Bildwiederholfrequenz gewählt werden. Falls es mehrere Cloud-Optionen mit der gleichen höchsten Wiederholfrequenz gibt, sollte aus diesen die Cloud-Option mit den niedrigsten Kosten gewählt werden. Dies ist beispielsweise in Abbildung 8.16 (rechts) der Fall. Hier ist die Netzwerkbandbreite der limitierende Faktor, der eine höhere Bildverarbeitungsrate auf der schnelleren Maschine verhindert. Zusätzlich kann es vorkommen, dass die höchste prognostizierte Bildverarbeitungsrate weit über der technisch möglichen Rate liegt. Je nach Lichtverhältnis und Bildauflösung liefern die verwendeten Geräte maximale Bildraten zwischen 7,5–25 fps. Bei höheren prognostizierten Raten sollte, falls möglich, eine andere Cloud-Option gewählt werden, da die prognostizierte Bildverarbeitungsrate nicht erreicht werden kann und eventuell trotzdem ein höherer Preis dafür gezahlt werden muss.

Abschließend wurde die vorhergesagte und die tatsächlich gemessene Bildverarbeitungsrate der 36 Experimente miteinander verglichen. Abbildung 8.17 zeigt die absolut vorhergesagten und gemessenen Werte (Minimum, Maximum und Mittelwert) und die berechnete Abweichung Q vom Schätzwert am oberen Rand (rechte Y-Achse). Von den 36 Experimenten erreichten nur 7 nicht die Qualität Q_{gut} von maximal 50 % Abweichung. Damit liegt $V_{Q_{gut}}$ bei ca. 81 % und somit ist das Gütekriterium $K_{Q_{gut}}$ erfüllt. Die maximale Abweichung wurde mit ca. 77 % gemessen.

Die Experimente zeigen jedoch auch, dass in einzelnen Fällen eine sehr schlechte Güte erzielt wird. Dies trifft vor allem für die UMTS/HSPA-Konfiguration zu. Dies kommt daher, dass die via *SPEEDTEST* vorhergesagte Netzwerkperformance nicht erreicht werden konnte. Ein Grund für dieses Verhalten konnte bereits in Kapitel 4 erörtert werden. Die dynamische Bandbreitenregulierung von HSPA-Verbindungen wirkt sich manchmal negativ auf Übertragungen aus, die nicht die volle Bandbreite in beide Richtungen im Duplexbetrieb senden. Dies Art der Datenübertragung wird jedoch im Fall der Streaming-Bildverarbeitungsapplikation benötigt. Kamerabilder werden zum Server übertragen und wieder heruntergeladen. Da jedoch der Upload beschränkt ist, kann im Download auch nicht die volle Bandbreite ausgeschöpft wer-

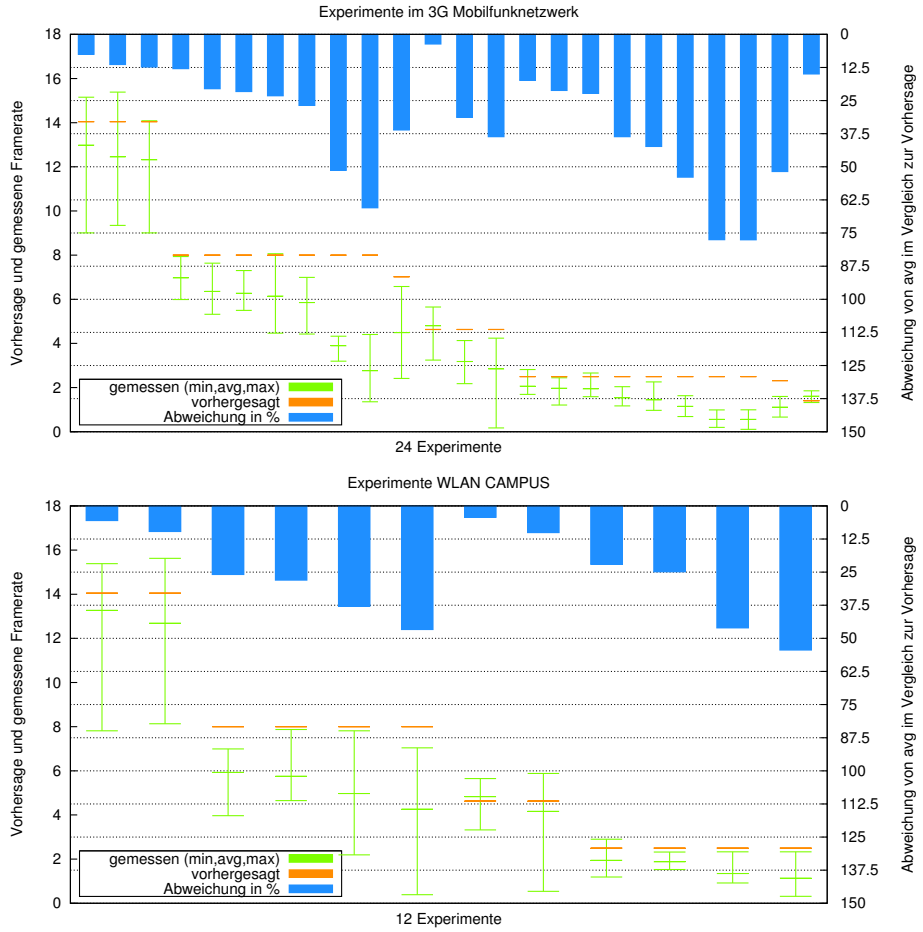


Abbildung 8.17: Visualisierung der gemessenen und der vorhergesagten Bildverarbeitungsrate und der Qualität der vorhergesagten Performance der 36 Experimente.

den. Dadurch wird die Bandbreite durch den Mobilfunkprovider manchmal so weit gekürzt, dass die Gesamtperformance stark absinkt. Dieses Verhalten tritt jedoch nur selten auf.

8.4 Fazit

In diesem Kapitel konnte gezeigt werden, dass die Middleware benutzt werden kann, um reale *Mobile Cloud-unterstützte Anwendungen* zu implementieren. Die Bereitstellungszeit für VM-Ressourcen und der Zeitoverhead für das Code-Offloading wurden anhand der prototypischen Implementierung der Middleware bestimmt. Bis zum Start der Übertragung der ersten tatsächlichen Tasks vergehen im Durchschnitt ca. 5–6 s. Die Middleware wurde durch synthetische Tasks auch auf deren generelle Eignung zur Vorhersage der Laufzeit untersucht. Dabei wurde festgestellt, dass durch den virtualisierten Softwarestack mit Schwankungen gerechnet werden muss, die jedoch nur selten größer sind als 20 %.

Für zwei reale Applikationen wurde außerdem die Konstruktion von Profilen erläutert. Die Güte der Vorhersage in verschiedenen Szenarien für diese realen Applikationen stand an-

schließend im Mittelpunkt der Untersuchung. Dabei konnte sowohl für die eingabeterminierte Applikation als auch für die benutzerterminierte Applikation das definierte Gütekriterium fast immer eingehalten werden. Eine sinnvolle Auswahl einer Cloud-Option zur Laufzeit ist für die betrachteten Applikationen also möglich. Dies versetzt den Nutzer in die Lage, die Performance der Ausführung der *Mobilen Cloud-unterstützten Anwendung* in Abhängigkeit der aktuellen Eingabe, Netzwerkgeschwindigkeit und Kostenbeschränkung passend zu wählen.

Als Restriktion der Middleware wurde jedoch auch festgestellt, dass die Vorhersage der verfügbaren Netzwerkperformance für asynchrone Upload/Download-Konfigurationen (HOME- und UMTS-Konfiguration) oft schlechtere Ergebnisse liefert als für die CAMPUS-Konfiguration. Dies macht erneut die Notwendigkeit deutlich, dass der Nutzer eine Möglichkeit haben sollte, die Vorhersage der Netzwerkperformance zu korrigieren. Diese wurde bei den implementierten Applikationen dadurch berücksichtigt, dass die Bestimmung der Netzwerkperformance gesondert erfolgt und der Nutzer die Werte für die Bestimmung von Cloud-Optionen noch anpassen kann.

Zudem sollten die Applikationsprofile zur Vorhersage der Laufzeit in regelmäßigen Abständen erneuert werden, da auch die Performance der VM-Typen in größeren Abständen deutlich schwanken kann. Allgemein beeinträchtigen diese Restriktionen die Anwendbarkeit des Verfahrens nicht grundlegend, sondern stellen vielmehr Verbesserungsmöglichkeiten dar, die im Umfang dieser Arbeit nicht mehr bearbeitet werden konnten.

9 Zusammenfassung

In diesem Kapitel wird die Arbeit zusammengefasst. Anschließend werden die wichtigsten Ergebnisse der Betrachtungen zu einer Handlungsempfehlung für die Erstellung von *Mobilen Cloud-unterstützten Anwendungen* gebündelt. Abschließend werden Anknüpfungspunkte für weiterführende Arbeiten vorgestellt. Außerdem wird eine kurze Übersicht geliefert über Technologien, die während der Erstellung der Arbeit verfügbar wurden sowie deren zu erwartende Auswirkungen auf die Ergebnisse dieser Arbeit.

9.1 Zusammenfassung der Ergebnisse

Die vorliegende Arbeit befasste sich mit der Fragestellung, wie rechenintensive mobile Applikationen durch *Cloud Computing* derart unterstützt werden können, dass die Dienstqualität für die Ausführungsressourcen (VM-Instanzen) zur Laufzeit sinnvoll anhand der aktuellen Situation gewählt werden kann. Für diese Art Applikationen wurde der Term *Mobile Cloud-unterstützte Anwendung* eingeführt. Als Zielgeräte wurden Mobilgeräte wie Smartphones, Tablets und kleine Laptops bestimmt, deren Ressourcen zugunsten einer längeren Batterielaufzeit oft eingeschränkt sind.

Es wurde festgestellt, dass die sinnvolle Auswahl einer VM-Instanz maßgeblich von drei Einflussfaktoren abhängt: der Netzwerkperformance, der Ausführungsgeschwindigkeit der VM-Instanz und der Beschaffenheit der Applikation selbst. Um eine Auswahl zu ermöglichen, müssen alle drei Faktoren möglichst gut vorhersagbar sein und effizient genutzt werden. Diese Kernanforderung wurde zu Beginn der Arbeit definiert. Zunächst wurde ein Modell erarbeitet, welches alle relevanten Parameter einschließt und die Ausführungszeit von *Mobilen Cloud-unterstützten Anwendungen* erfassen kann. Danach wurde für alle drei Einflussfaktoren untersucht, wie diese die Performance und die Vorhersagbarkeit beeinflussen. Insbesondere war die Schwankungsbreite und -häufigkeit von Interesse.

Im Bereich der Netzwerkperformance wurden WLAN und Mobilfunknetze der 3. Generation (3G) in drei verschiedenen Konfigurationen untersucht. Zudem wurde der Einfluss von Verschlüsselung und Codierung der Nutzdaten erfasst. Die Analyse der Eigenschaften mobiler Kommunikationsnetze mündete in den Entwurf einer effizienten Kommunikationsbibliothek auf Basis einer persistenten TCP-Verbindung mit Pipelining, die sowohl die Gesamtperformance als auch die Vorhersagbarkeit der Datenübertragungsrate verbessern konnte.

Außerdem wurde eine Middleware für *Java* entworfen und implementiert, die die Ausführung von *Mobilen Cloud-unterstützten Anwendungen* und die Auswahl der Dienstqualität zur Laufzeit unterstützt. Die Auswahl zur Laufzeit und eine möglichst gute Auslastung von VM-Instanzen durch Wiederverwendung machen die Installation des Programmcodes zur Laufzeit notwendig. Dies ist unter dem Begriff Code-Offloading bekannt und ebenfalls durch die Middleware implementiert. Durch diese Technik kann der Kostenoverhead beim stundenbasierten Mieten von VM-Instanzen stark reduziert werden, was im Ergebnis für eine feingranularere Abrechnung für den Endkunden genutzt werden kann. Die Middleware ermöglicht eine sekundengenaue Abrechnung. Der verwendete Softwarestack wurde ebenfalls auf Performan-

ceschwankungen untersucht. Hierbei wurde insbesondere der Einfluss durch Virtualisierung analysiert. Die VM-Instanzen selbst sind virtuelle Maschinen, die vom IaaS-Anbieter bereitgestellt werden. Zudem wird auch das *Java*-Programm selbst durch eine JVM ausgeführt. Es wurde festgestellt, dass Performanceschwankungen auftreten können, die die Vorhersage beeinflussen, jedoch nicht unmöglich machen.

Schließlich wurde der Einfluss der eigentlichen Applikation auf die Vorhersagbarkeit der Laufzeit betrachtet. Hierzu wurden Applikationen zunächst kategorisiert in eine Menge, die sich zwar auf der Middleware ausführen lässt, deren Verhalten jedoch kaum vorhergesagt werden kann. Für die verbleibende zweite Menge potenziell vorhersagbarer Applikationen wurden zudem Kriterien bezüglich der Performanceschwankung definiert, unter denen eine Vorhersage möglich ist. Außerdem muss zur Vorhersage ein Zusammenhang zwischen einer Eingabegröße und der Laufzeit hergestellt werden. Dazu wurde die Methode der Regressionsanalyse vorgestellt und anhand von Beispielen erklärt.

Die Evaluation der prototypischen Implementierung der Middleware und insbesondere der Güte der vorhergesagten Performance zur Ressourcenauswahl wurde anhand realer Applikationen durchgeführt. Dazu wurde die Raytracing-Funktionanlität zur Erstellung fotorealistischer Bilder eines 3D-Hausdesigners und eine Applikation zur fortlaufenden Kamerabildverarbeitung auf *Android*-Geräten genutzt. Zusätzlich wurden auch synthetische Applikationen verwendet, um die Grenzen der entwickelten Technik zu untersuchen. Für fast alle Szenarien konnte eine zufriedenstellende Güte der Vorhersage erreicht werden. Fehlprognostizierte Werte sind fast immer auf die falsche Einschätzung der verfügbaren Netzwerkperformance zurückzuführen.

9.2 Erstellungsleitfaden für Mobile Cloud-unterstützte Anwendungen

Die Analyse der Performance mobiler Kommunikationsverbindungen ergab, dass aufgrund der relativ hohen Latenzen der Einsatz einer persistenten TCP-Verbindung große Vorteile bringt, vor allem wenn auch die Verschlüsselung obligatorisch ist. Zudem wurde festgestellt, dass die tatsächlich verfügbare Übertragungsgeschwindigkeit auch dann sehr pessimistisch zu bewerten ist. Vor allem die Duplexfähigkeiten der untersuchten Technologien sind oft beschränkt durch asynchrone Upload/Download-Konfiguration oder durch viele Teilnehmer im WLAN. Tatsächlich nutzbar waren oft nur Übertragungsraten im einstelligen Mbit-Bereich. Dies sollte beim Entwurf von *Mobilen Cloud-unterstützten Anwendungen* berücksichtigt werden. Eventuell ist es sinnvoll, die Daten während der Übertragung zusätzlich zu komprimieren. Auch in Bezug auf die Schwankungsbreite der nutzbaren Übertragungsrate wurde festgestellt, dass diese hoch sein kann und manchmal sehr unvermittelt starke Schwankungen auftreten. Deshalb sollte die verfügbare Übertragungsbandbreite nicht automatisch bestimmt werden. Ein Benchmarkwerkzeug zur Bestimmung der momentan verfügbaren Netzwerkperformance liefert zwar eine gute Orientierung, oft kann der Nutzer selbst durch seine eigene Erfahrung die verfügbare Bandbreite jedoch besser abschätzen. Beispielsweise kann in einem vollen Hörsaal die momentan verfügbare Bandbreite hoch sein. Aufgrund der vielen Teilnehmer ist dies aber über einen längeren Zeitraum kaum zu erwarten. Da man jedoch vor der Allokation der VM die verfügbare Bandbreite schätzen muss, sollte in diesem Fall eher pessimistisch kalkuliert werden. Schnelle Bewegung und eine deutliche Entfernung vom ursprünglichen Ort, für den die VM-Allokation durchgeführt wurde, führen sehr wahrscheinlich auch dazu, dass

die vorhergesagte Performance der *Mobilen Cloud-unterstützten Anwendung* nicht erreicht wird. Für die in dieser Arbeit betrachteten Kommunikationsnetzwerke und das entwickelte Kommunikationsprotokoll war die Bandbreitenvorhersage für Nutzdaten ab ca. 50 kB recht genau.

Bei Entwicklung der Middleware lag das Hauptaugenmerk auf einer möglichst guten Unterstützung der Vorhersagbarkeit und der durchweg verschlüsselten Kommunikation. Dadurch konnten nicht alle Arten von Applikationen unterstützt werden. Alle Applikationen, die auf der entwickelten Middleware ausgeführt werden, müssen taskbasiert arbeiten. Tasks sind als kleinste Einheit für eine effiziente Ausnutzung der Kommunikation notwendig. Zusätzlich sind Tasks auch sehr gut für Code-Offloading geeignet, da nur eine entsprechende Taskfunktion installiert werden muss. Eine weitere Einschränkung musste getroffen werden, wenn die Laufzeit der taskbasierten Anwendung auch vorhersagbar sein soll, sodass eine Ressourcenauswahl zur Laufzeit möglich wird. Hierbei können nur eingabeterminierte Applikationen unterstützt werden, die aus sequentiell zu verarbeitenden Tasks bestehen und deren vollständige Eingabe bekannt ist. Zudem können auch benutzerterminierte Applikationen unterstützt werden, die aus unabhängigen Einzelprozessortasks bestehen, bei denen die Eingabe zwar nicht vollständig bekannt ist, aber deren Tasklaufzeiten und Eingabegröße immer nahezu konstant bleiben. Für diese Art Applikationen lässt sich eine Verarbeitungsrate von Tasks pro Sekunde angeben.

Für die Vorhersage müssen Profile erstellt werden, die aufgrund der Beschaffenheit der Eingabedaten eine Laufzeit schätzen. Es muss also möglich sein, für die Tasks der Applikation einen solchen Zusammenhang herzustellen. In dieser Arbeit wurde gezeigt, dass dies über eine (multiple) Regression für bis zu zwei unabhängige Größen gelingen kann. Die notwendigen Monitoringdaten für die Regression werden dabei von der Middleware selbst gesammelt. Die erreichte Güte der Schätzung muss die Konfiguration verschiedener Typen von Cloud-Ressourcen zur Ausführung zulassen. Streuen die tatsächlichen Laufzeiten beispielsweise stark vom Vorhersagewert, so müssen VM-Typen mit sehr großen Performanceunterschieden konfiguriert werden, damit eine sinnvolle Unterscheidung gegeben ist. Da jedoch Ressourcen nicht in beliebigen Konfigurationen vorliegen, wurde für die Ressourcen des Cloud-Anbieters *Amazon EC2* die Einschränkung getroffen, dass die Performance der Ressourcen nicht mehr als Faktor 4 verschieden sein kann. Dadurch ergibt sich eine Güte der Schätzung von Tasklaufzeiten derart, dass mindestens 80 % der gemessenen Tasklaufzeiten um nicht mehr als $\pm 50\,\%$ vom Vorhersagewert abweichen. Andernfalls können eventuell nur 1-2 VM-Typen konfiguriert werden, was die sinnvolle Auswahl in Frage stellt. Zu beachten ist auch, dass aufgrund des teilweise virtualisierten Softwarestacks auch schon Schwankungen der Laufzeit für berechnungsintensive Tasks im unteren zweistelligen Bereich zu erwarten sind. I/O-intensive Tasks sollten gänzlich vermieden werden. Auch die JVM kann aufgrund der *Garbage Collection* zu schwankenden Laufzeiten beitragen, jedoch ist dies nur für sehr kurze Tasklaufzeiten von weit unter 1 s deutlich. Als besonders tauglich für die Vorhersage haben sich Tasks gezeigt, deren Laufzeit mindestens 1 s beträgt.

Obwohl die entwickelte Middleware die Abrechnung der Nutzung auf Basis von Sekunden ermöglicht, sollte die Ausführungszeit einer *Mobilen Cloud-unterstützten Anwendung* nicht nur wenige Sekunden betragen. Hier macht sich negativ bemerkbar, dass allein für die Dauer der Allokation einer VM-Instanz und für die Installation des benötigten Programmcodes ca. 5–10 s zu erwarten sind. Damit eine Beschleunigung der *Mobilen Cloud-unterstützten Anwendung* zu erreichen ist, muss deren Laufzeit somit entsprechend länger sein. Die getesteten Beispielapplikationen hatten in der Regel Ausführungszeiten im einstelligen Minutenbereich.

9.3 Ausblick und weiterführende Arbeiten

Seit der Formulierung der Anforderungen an die Arbeit in den Jahre 2011/2012 sind diverse Verbesserungen bezüglich der verwendeten Technologien etabliert worden. Zudem wurden in dieser Arbeit für die Untersuchungen der Eigenschaften von *Mobilen Cloud-unterstützten Anwendungen* diverse Einschränkungen getroffen. So wurde beispielsweise nur ein Cloud-Anbieter und nur ein Mobilfunknetz untersucht. Dies lag unter anderem auch in der Tatsache begründet, dass die Benutzung dieser Dienste mit erheblichen Kosten verbunden ist. Dennoch soll hier abschließend kurz darauf eingegangen werden, welche Neuerungen und Eigenschaften anderer Anbieter heute im Jahr 2014 eventuell einen Einfluss auf die Ergebnisse der Arbeit haben könnten und wie die in dieser Arbeit entwickelten Techniken erweitert werden könnten.

Zunächst ist festzuhalten, dass im Laufe der Erstellung der Arbeit stetig an der Verbesserung der mobilen Kommunikationstechnologien gearbeitet wurde. Im Mobilfunkbereich steht im Jahr 2014 in Deutschland beispielsweise ein nahezu flächendeckendes LTE-Netz (4. Generation (4G)) zur Verfügung. Dadurch verbessert sich allgemein die Performance mobiler Kommunikationsverbindungen. Im Vergleich zum Mobilfunknetz der 3. Generation sind ca. die doppelte Download- und dreifache Upload-Geschwindigkeit möglich [PH13]. Außerdem wurden auch im Bereich WLAN neue Standards verabschiedet (beispielsweise 802.11ac), welche die Performance stark verbessern können. Entsprechende Geräte sind jedoch noch sehr rar und werden noch nicht flächendeckend eingesetzt[Ahl14]. Auch die Maximalgeschwindigkeit der Breitbandanschlüsse für Privatkunden hat sich im Vergleich zu 2012 leicht erhöht[1]. Viele Anschlüsse sind jedoch weiterhin asynchron ausgeführt, wodurch die relativ schlechte Uploadgeschwindigkeit die Performance von *Mobilen Cloud-unterstützten Anwendungen* stark negativ beeinflussen kann. Eine Ausnahme bilden hier auch immer häufiger anzutreffende Breitbandanschlüsse via LTE. Diese weisen oft deutlich höhere Uploadperformance auf als beispielsweise Anschlüsse via Telefon/DSL oder Fernsehkabel. Im Bereich der WLAN-Großinstallationen (beispielsweise CAMPUS-Netzwerk) werden bestehende Systeme nur langsam ausgetauscht, was dazu führt, dass flächendeckend hier auch weiterhin nur WLAN 802.11g mit maximal 54 Mbit/s verfügbar ist. Insgesamt gesehen ist jedoch die Unterstützung von *Mobilen Cloud-unterstützten Anwendungen* verbessert worden, insbesondere in den Mobilfunknetzen. Hier können zukünftige Arbeiten zeigen, ob diese Verbesserungen auch eine bessere Nutzung in der Bewegung ermöglichen.

Im Bereich der Cloud-Anbieter wurden während der Erstellung der Arbeit teils erhebliche Neuerungen eingeführt. So bieten immer mehr Anbieter im Jahr 2014 auch sehr performante VM-Instanztypen aus dem Bereich des High-Performance-Computing an[2], wie beispielsweise das im Mai 2012 am Markt erschienene Unternehmen *ProfitBricks*. Diese Instanztypen mit vielen CPU-kernen erlauben nicht nur eine größere Auswahl für *Mobile Cloud-unterstützte Anwendungen*, sondern bieten auch noch bessere Performance, die beispielsweise für die vorgestellte Raytracing-Applikation weitere Verbesserungen bringen könnte. Zudem haben einige

[1] Im Januar 2013 wurden 59 % der stationären Breitbandanschlüsse in der Europäischen Union mit mehr als 10 Mbit/s beliefert. Ein Jahr zuvor wurden nur 48,6 % mit dieser Geschwindigkeit beliefert. (siehe Onlineressource `http://de.statista.com/statistik/daten/studie/173685/umfrage/kabelgebundene-breitbandanschluesse-in-der-eu-nach-geschwindigkeit/`(abgerufen am 8.2.2014))

[2] *„Today, Cloud Computing brings HPC like performance and scalability to a much wider audience, allowing HPC teams to expedite their HPC workloads using highly flexible and scalable compute resources on demand, as needed, and save money by paying only for the resources that are actually consumed.“* (Onlineressource `http://www.profitbricks.com/high-performance-computing-hpc` (abgerufen am 8.2.2014))

Anbieter auch damit begonnen, die Abrechnung der Nutzungsdauer feingranularer zu gestalten. Der Dienst *Google Compute Engine* rechnet seine Instanzen nunmehr im Minutentakt ab, nachdem eine minimale Anlaufphase von 10 min abgeschlossen ist[3]. Dadurch kann der Kostenoverhead für kürzere Nutzungsdauern von *Mobilen Cloud-unterstützten Anwendungen* weiter gesenkt werden. Hier könnte in zukünftigen Arbeiten auf Basis des erarbeiteten Simulationsmodells analysiert werden, wie sich diese Verbesserungen auf den Kostenoverhead auswirken.

Die entwickelte Middleware ist in der Lage, Cloud-Optionen zur Laufzeit zu präsentieren, die vorher als Profil installiert wurden. Es hat sich während der Arbeit als teilweise mühsam herausgestellt, ein Profil auf Basis der Logdaten zu generieren. Als sinnvolle Erweiterung erscheint demnach die automatische Generierung von Applikationsprofilen auf Basis der schon von der Middleware gesammelten Logdaten. Lediglich die Korrelationsparameter sollten noch von Hand gewählt werden. Auf Basis der erarbeiteten Methoden (beispielsweise Regressionsanalyse) und der notwendigen Einschränkungen der Schwankungsbreite der Approximation könnten so nicht nur automatisch Profile erzeugt werden, sie könnten auch fortwährend angepasst werden. Zudem könnte in weiterführenden Arbeiten analysiert werden, ob beispielsweise maschinelles Lernen eingesetzt werden kann, um die Laufzeit von Tasks vorherzusagen.

[3] *„Google bills in minute-level increments (with a 10-minute minimum charge), so you don't pay for unused computing time."* (Onlineressource `https://cloud.google.com/products/compute-engine/`(abgerufen am 8.2.2014))

Literaturverzeichnis

[20009] European Telecommunications Standards Institute - ETSI: Real-time Transport Protocol (RTP) usage model (3GPP TR 26.937 version 8.0.0 Release 8). Version: 2009. `http://www.etsi.org/deliver/etsi_tr/126900_126999/126937/08.00.00_60/tr_126937v080000p.pdf`. 2009. – Technischer Bericht. – Abgerufen am 19.5.2014

[20112] IEEE Computer Society: Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Version: März 2012. `http://standards.ieee.org/getieee802/download/802.11-2012.pdf`. 2012. – Technischer Bericht. – Abgerufen am 19.5.2014

[ACGS04] Ammons, Glenn ; Choi, Jong-Deok ; Gupta, Manish ; Swamy, Nikhil: Finding and Removing Performance Bottlenecks in Large Systems. In: *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*, Springer, 2004, S. 172–196

[AFG+09] Armbrust, Michael ; Fox, Armando ; Griffith, Rean ; Joseph, Anthony D. ; Katz, Randy H. ; Konwinski, Andrew ; Lee, Gunho ; Patterson, David A. ; Rabkin, Ariel ; Stoica, Ion ; Zaharia, Matei: Above the Clouds: A Berkeley View of Cloud Computing / EECS Department, University of California, Berkeley. 2009 (UCB/EECS-2009-28). – Technischer Bericht

[Ahl14] Ahlers, Ernst: Rasante Datenjongleure - 13 Gigabit-WLAN-Router im Vergleich. In: *c't magazin für computer technik* 1/2014 (2014), S. 78–90. – ISSN 07248679

[AZ06] Assaad, Mohamad ; Zeghlache, Djamal: *TCP Performance over UMTS-HSDPA Systems*. Taylor & Francis, 2006. – ISBN 9780849368387

[Ban06] Banks, Tim: *Web Services Resource Framework (WSRF)–Primer v1. 2*. http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf, Mai 2006. – Abgerufen am 19.5.2014

[BDF+03] Barham, Paul ; Dragovic, Boris ; Fraser, Keir ; Hand, Steven ; Harris, Tim ; Ho, Alex ; Neugebauer, Rolf ; Pratt, Ian ; Warfield, Andrew: Xen and the Art of Virtualization. In: *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)*, ACM, 2003, S. 164–177

[BDGS08] Braun, Torsten ; Diaz, Michel ; Gabeiras, Jose E. ; Staub, Thomas: *End-to-End Quality of Service Over Heterogeneous Networks*. Springer, 2008. – ISBN 9783540791201

[BEK+00] Box, Don ; Ehnebuske, David ; Kakivaya, Gopal ; Layman, Andrew ; Mendelsohn, Noah ; Nielsen, Henrik F. ; Thatte, Satish ; Winer, Dave: *Simple*

Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/soap/, 2000. – Abgerufen am 19.5.2014

[BFGY08] Braberman, Victor ; Fernandez, Federico ; Garbervetsky, Diego ; Yovine, Sergio: Parametric Prediction of Heap Memory Requirements. In: *Proceedings of the 7th International Symposium on Memory Management (ISMM)*, ACM, 2008, S. 141–150

[BGH+06] Blackburn, Stephen M. ; Garner, Robin ; Hoffmann, Chris ; Khang, Asjad M. ; McKinley, Kathryn S. ; Bentzur, Rotem ; Diwan, Amer ; Feinberg, Daniel ; Frampton, Daniel ; Guyer, Samuel Z. ; Hirzel, Martin ; Hosking, Antony ; Jump, Maria ; Lee, Han ; Moss, J. Eliot B. ; Phansalkar, Aashish ; Stefanovic, Darko ; VanDrunen, Thomas ; Dincklage, Daniel von ; Wiedermann, Ben: The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In: *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, ACM, 2006, S. 169–190

[BK11] Basole, Rahul ; Karla, Jürgen: Entwicklung von Mobile-Platform-Ecosystem-Strukturen und -Strategien. In: *WIRTSCHAFTSINFORMATIK* 53 (2011), Nr. 5, S. 301–311. – ISSN 09376429

[BMB+05] Bless, Roland ; Mink, Stefan ; Blass, Erik-Oliver ; Conrad, Michael ; Hof, Hans-Joachim ; Kutzner, Kendy ; Schöller, Marcus: *Sichere Netzwerkkommunikation - Grundlagen, Protokolle und Architekturen.* Springer, 2005. – ISBN 9783540278962

[Bra13] Brandt, Siegmund: *Datenanalyse für Naturwissenschaftler und Ingenieure: Mit statistischen Methoden und Java-Programmen.* Springer Spektrum, 2013. – ISBN 9783642376634

[BS96] Buss, Arnold H. ; Stork, Kirk A.: Discrete Event Simulation on the World Wide Web using Java. In: *Proceedings of the 28th Conference on Winter Simulation (WSC)*, IEEE, 1996, S. 780–785

[BSMM12] Bronstein, Ilja N. ; Semendjajew, Konstantin A. ; Musiol, Gerhard ; Mühlig, Heiner: *Taschenbuch der Mathematik.* 2012. – ISBN 3808556706

[CBC+10] Cuervo, Eduardo ; Balasubramanian, Aruna ; Cho, Dae-ki ; Wolman, Alec ; Saroiu, Stefan ; Chandra, Ranveer ; Bahl, Paramvir: MAUI: Making Smartphones Last Longer with Code Offload. In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, ACM, 2010, S. 49–62

[CFM04] Capone, Antonio ; Fratta, Luigi ; Martignon, Fabio: Bandwidth Estimation Schemes for TCP over Wireless Networks. In: *IEEE Transactions on Mobile Computing* 3 (2004), Nr. 2, S. 129–143. – ISSN 15361233

[CIM+11] Chun, Byung-Gon ; Ihm, Sunghwan ; Maniatis, Petros ; Naik, Mayur ; Patti, Ashwin: CloneCloud: Elastic Execution between Mobile Device and Cloud. In:

Proceedings of the 6th Conference on Computer Systems (EuroSys), ACM, 2011, S. 301 – 314

[CLK+11] Cidon, Asaf ; London, Tomer M. ; Katti, Sachin ; Kozyrakis, Christos ; Rosenblum, Mendel: MARS: Adaptive Remote Execution for Multi-threaded Mobile Devices. In: *Proceedings of the 3rd Workshop on Networking, Systems, and Applications on Mobile Handhelds (MobiHeld)*, ACM, 2011, S. 1–6

[CLRS01] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction To Algorithms*. MIT Press, 2001. – ISBN 9780262032933

[CM99] Chen, Xiangping ; Mohapatra, Prasant: Providing Differentiated Service from an Internet Server. In: *Proceedings of the 8th International Conference on Computer Communications and Networks (ICCCN)*, IEEE, 1999, S. 214–217

[CM10] Chun, Byung-Gon ; Maniatis, Petros: Dynamically Partitioning Applications between Weak Devices and Clouds. In: *Proceedings of the 1st Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, ACM, 2010, S. 7:1–7:5

[Com06] Comer, Douglas E.: *Internetworking with TCP/IP.: Principles, protocols, and architecture. Vol. 1*. 5th Edition. Prentice-Hall International, 2006. – ISBN 9780131876712

[CRB11] Calheiros, Rodrigo N. ; Ranjan, Rajiv ; Buyya, Rajkumar: Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments. In: *Proceedings of the International Conference on Parallel Processing (ICPP)*, IEEE, 2011, S. 295–304

[Cro06] Crockford, Douglas: *The application/json Media Type for JavaScript Object Notation (JSON)*. http://tools.ietf.org/html/rfc4627, Juli 2006. – Abgerufen am 19.5.2014

[DLNW11] Dinh, Hoang T. ; Lee, Chonho ; Niyato, Dusit ; Wang, Ping: A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches. In: *Wireless Communications and Mobile Computing* (2011). – ISSN 15308677

[Eck12] Eckert, Claudia: *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. 7, überarbeitete und erweiterte Auflage. Oldenbourg Wissenschaftsverlag, 2012. – ISBN 9783486706871

[Emm03] Emmerich, Wolfgang: *Konstruktion von verteilten Objekten*. dpunkt-Verlag, 2003 (dpunkt-Lehrbuch). – ISBN 9783898641401

[Erl06] Erl, Thomas: *Service-Oriented Architecture: Concepts, Technology, And Design*. Pearson Education, 2006. – ISBN 9788131714904

[FE10] Furht, Borko (Hrsg.) ; Escalante, Armando (Hrsg.): *Handbook of Cloud Computing*. Springer, 2010. – ISBN 9781441965233

[Feh12] Feher, Andrija: *Entwurf und Analyse eines skalierbaren Web-Dienstes zur verteilten Compilierung auf der Amazon EC2 Cloud-Infrastruktur*, Universität Bayreuth, Bachelorarbeit, 2012

[FGM+99] Fielding, Roy ; Gettys, J ; Mogul, J ; Frystyk, H ; Masinter, L ; Leach, P ; Berners-Lee, T: *RFC 2616-Hypertext Transfer Protocol*. http://www.ietf.org/rfc/rfc2616.txt, Juni 1999. – Abgerufen am 20.6.2013

[FHR09] Ferber, Marvin ; Hunold, Sascha ; Rauber, Thomas: Load Balancing Concurrent BPEL Processes by Dynamic Selection of Web Service Endpoints. In: *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW)*, IEEE, 2009, S. 290–297

[Fie00] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Doktorarbeit, 2000

[FK04] Foster, Ian T. ; Kesselman, Carl.: *The Grid 2: Blueprint for a New Computing Infrastructure*. Elsevier Limited, Oxford, 2004 (The Elsevier Series in Grid Computing). – ISBN 9781558609334

[FKL09] Fahrmeir, Ludwig ; Kneib, Thomas ; Lang, Stefan: *Regression*. Springer, 2009. – ISBN 9783642018374

[For12] Forbes.com, Trefis Team (verschiedene Autoren): *Can Intel Challenge ARM's Mobile Dominance?* http://www.forbes.com/sites/greatspeculations/2012/11/09/can-intel-challenge-arms-mobile-dominance/, Nov 2012. – Abgerufen am 15.10.2013

[Fos05] Foster, Ian: Globus Toolkit Version 4: Software for Service-Oriented Systems. In: *Proceedings of the International Conference on Network and Parallel Computing*, Springer, 2005 (NPC), S. 2–13

[FR12] Ferber, Marvin ; Rauber, Thomas: Mobile Cloud Computing in 3G Cellular Networks using Pipelined Tasks. In: *Proceedings of the European Conference on Service-Oriented and Cloud Computing (ESOCC)*, Springer, 2012, S. 192–199

[FRH10] Ferber, Marvin ; Rauber, Thomas ; Hunold, Sascha: Combining Object-Oriented Design and SOA with Remote Objects over Web Services. In: *Proceedings of the 8th European Conference on Web Services (ECOWS)*, IEEE, 2010, S. 83–90

[Fri12] Fries, Aidan: *The use of Java in large scientific applications in HPC environments*, Universitat de Barcelona, Doktorarbeit, November 2012

[FRTH12] Ferber, Marvin ; Rauber, Thomas ; Torres, Mario Henrique C. ; Holvoet, Tom: Resource Allocation for Cloud-Assisted Mobile Applications. In: *Proceedings of the 5th International Conference on Cloud Computing (CLOUD)*, IEEE, 2012, S. 400–407

[FZ94] Forman, George H. ; Zahorjan, John: The Challenges of Mobile Computing. In: *Computer* 27 (1994), April, Nr. 4, S. 38 –47. – ISSN 00189162

[FZRL08] Foster, Ian ; Zhao, Yong ; Raicu, Ioan ; Lu, Shiyong: Cloud Computing and Grid Computing 360-Degree Compared. In: *Proceedings of the Grid Computing Environments Workshop (GCE)*, 2008, S. 1–10

[GB09] Griffiths, Dawn ; Beyer, Jörg: *Statistik von Kopf bis Fuß*. O'Reilly, 2009 (Ein Buch zum Mitmachen und Verstehen). – ISBN 9783897218918

[GGW10] Gong, Zhenhuan ; Gu, Xiaohui ; Wilkes, J.: PRESS: PRedictive Elastic ReSource Scaling for Cloud Systems. In: *Proceedings of the International Conference on Network and Service Management (CNSM)*, IEEE, 2010, S. 9–16

[GJ79] Garey, Michael R. ; Johnson, David S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. – ISBN 0716710455

[GJS+13] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad ; Buckley, Alex: *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley, 2013 (Java Series). – 672 S. – ISBN 9780133260441

[Grü08] Grüne, Lars: *Numerische Mathematik I (Dritte Auflage)*. Version: Wintersemester 2007/2008. `http://num.math.uni-bayreuth.de/de/team/Gruene_Lars/lecture_notes/num1/num1_3.pdf`. Vorlesungsskript. – Abgerufen am 19.5.2014

[GTF06] Geoffray, Nicolas ; Thomas, Gaël ; Folliot, Bertil: Transparent and Dynamic Code Offloading for Java Applications. In: *Proceedings of the Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*, 2006, S. 1790–1806

[HAMJ02] Hsu, Rein-Lien ; Abdel-Mottaleb, Mohamed ; Jain, Anil K.: Face Detection in Color Images. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24 (2002), Nr. 5, S. 696–706

[HBDTD07] Hens, Raf ; Boone, Bas ; De Turck, Filip ; Dhoedt, Bart: Runtime Deployment Adaptation for Resource Constrained Devices. In: *Proceedings of the International Conference on Pervasive Services (ICPS)*, IEEE, 2007, S. 335–340

[HC10] Hunter, Jason ; Crawford, William: *Java Servlet Programming*. O'Reilly Media, 2010. – ISBN 9781449390679

[HL08] Hafner, Katie ; Lyon, Metthew: *Arpa Kadabra oder die Anfänge des Internet*. Dpunkt.Verlag GmbH, 2008. – ISBN 9783898645515

[Hun09] Hunold, Sascha: *Evaluation der Leistungsfähigkeit von gemischt-parallelen Programmen in homogenen und heterogenen Umgebungen unter Berücksichtigung effizienter Schedulingstrategien*, Universität Bayreuth, Doktorarbeit, 2009

[Int12] International Data Corporation (IDC): *Smartphone Market Hits All-Time Quarterly High Due To Seasonal Strength and Wider Variety of Offerings, According to IDC*. http://www.businesswire.com/news/home/20120206005252/en/Smartphone-Market-Hits-All-Time-Quarterly-High-Due, Feb 2012. – Abgerufen am 15.10.2013

[IOY+11] Iosup, Alexandru ; Ostermann, Simon ; Yigitbasi, Nezih ; Prodan, Radu ; Fahringer, Thomas ; Epema, Dick: Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. In: *IEEE Transactions on Parallel and Distributed Systems* 22 (2011), Juni, Nr. 6, S. 931 –945. – ISSN 10459219

[IYE11] Iosup, Alexandru ; Yigitbasi, Nezih ; Epema, Dick: On the Performance Variability of Production Cloud Services. In: *Proceedings of the 11th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE, 2011, S. 104 –113

[JA11] Jayasinghe, Deepal ; Azeez, Afkham: *Apache Axis2 Web Services*. Packt Open Source Pub., 2011. – ISBN 9781849511575

[KBS05] Krafzig, Dirk ; Banke, Karl ; Slama, Dirk: *Enterprise Soa: Service-Oriented Architecture Best Practices*. Prentice Hall, 2005 (The Coad Series). – ISBN 9780131465756

[KEEH10] Kaspar, Dominik ; Evensen, Kristian ; Engelstad, Paal ; Hansen, Audun F.: Using HTTP Pipelining to Improve Progressive Download over Multiple Heterogeneous Interfaces. In: *Proceedings of the International Conference on Communications (ICC 2010)*, IEEE, 2010, S. 1 –5

[KL10] Kumar, Karthik ; Lu, Yung-Hsiang: Cloud Computing for Mobile Users: Can Offloading Computation Save Energy? In: *Computer* 43 (2010), Nr. 4, S. 51–56. – ISSN 00189162

[Koh05] Kohn, Wolfgang: *Statistik: Datenanalyse und Wahrscheinlichkeitsrechnung*. Springer, 2005 (Statistik und ihre Anwendungen). – ISBN 9783540216773

[KPKB10] Kemp, Roelof ; Palmer, Nicholas ; Kielmann, Thilo ; Bal, Henri: Cuckoo: a Computation Offloading Framework for Smartphones. In: *Proceedings of the 2nd International Conference on Mobile Computing, Applications, and Services (MobiCASE)*, Springer, 2010, S. 59–79

[KR08] Kurose, James F. ; Ross, Keith W.: *Computernetzwerke: Der Top-Down-Ansatz*. Pearson Studium, 2008 (IT - Informatik). – ISBN 9783827373304

[Leh03] Lehner, Franz: *Mobile und drahtlose Informationssysteme: Technologien, Anwendungen, Märkte*. Springer, 2003. – ISBN 9783642556265

[Leu04] Leung, Joseph Y-T.: *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., 2004. – ISBN 1584883979

[LML+11] Lenk, Alexander ; Menzel, Michael ; Lipsky, Johannes ; Tai, Stefan ; Offermann, Philipp: What Are You Paying For? Performance Benchmarking for Infrastructure-as-a-Service Offerings. In: *Proceedings of the International Conference on Cloud Computing (CLOUD 2011)*, IEEE, 2011, S. 484 –491

[LYBB13] Lindholm, Tim ; Yellin, Frank ; Bracha, Gilad ; Buckley, Alex: *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, 2013 (Java Series). – 608 S. – ISBN 9780133260441

[LYKZ10] Li, Ang ; Yang, Xiaowei ; Kandula, Srikanth ; Zhang, Ming: CloudCmp: Comparing Public Cloud Providers. In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ACM, 2010, S. 1 – 14

[Mat13] Matsudaira, Kate: Making the Mobile Web Faster. In: *Communications of the ACM* 56 (2013), März, Nr. 3, S. 56–61. – ISSN 00010782

[Mau04] Maufer, Thomas: *A Field Guide to Wireless LANs for Administrators and Power Users*. Prentice Hall Professional Technical Reference, 2004. – ISBN 9780131014060

[MD11] Mazzucco, Michele ; Dumas, Marlon: Achieving Performance and Availability Guarantees with Spot Instances. In: *Proceedings of the 13th International Conference on High Performance Computing and Communications (HPCC)*, IEEE, 2011, S. 296–303

[MG11] Mell, Peter ; Grance, Timothy: The NIST Definition of Cloud Computing / National Institute of Standards and Technology. Version: 2011. `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`. 2011. – Technischer Bericht. – Abgerufen am 19.5.2014

[MM07] Moodie, Matthew ; Mittal, Kunal: *Pro Apache Tomcat 6*. Apress, 2007 (Expert's voice in Java technology). – ISBN 9781430203780

[MPH10] Membrey, Peter ; Plugge, Eelco ; Hawkins, Tim: *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, 2010 (Books for professionals by professionals). – ISBN 9781430230519

[MPPS+12] Mazzola Paluska, Justin ; Pham, Hubert ; Schiele, Gregor ; Becker, Christian ; Ward, Steve: Vision: a Lightweight Computing Model for Fine-Grained Cloud Computing. In: *Proceedings of the 3rd Workshop on Mobile Cloud Computing and Services*, ACM, 2012, S. 3–8

[MW12] Meeker, Mary ; Wu, Liang: *2012 Internet Trends*. http://www.kpcb.com/insights/2012-internet-trends, May 2012. – Abgerufen am 6.2.2013

[NA08] Nixon, Mark S. ; Aguado, Alberto S.: *Feature Extraction & Image Processing*. Academic, 2008 (Academic Press). – ISBN 9780123725387

[Nag09] Nagel, Raik: *Erweiterung des Threadmodelles fur den Einsatz in verteilten und heterogenen Systemumgebungen*, Universität Bayreuth, Doktorarbeit, 2009

[Nie08] Niemeier, Wolfgang: *Ausgleichungsrechnung: Statistische Auswertemethoden*. Walter de Gruyter, 2008. – ISBN 9783110190557

[Obj12] Object Management Group (OMG): *Common Object Request Broker Architecture (CORBA)*. http://www.omg.org/spec/CORBA/, November 2012. – Abgerufen am 19.5.2014

[OK10] Oprescu, Anna ; Kielmann, Thilo: Bag-of-Tasks Scheduling under Budget Constraints. In: *Second International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2010, S. 351–359

[OM04] Orlamünder, D. ; Mascolus, W.: *Computergrafik und OpenGL: eine systematische Einführung ; mit 26 Übungen*. Fachbuchverl. Leipzig im Carl-Hanser-Verlag, 2004. – ISBN 9783446228375

[ORA11] ORACLE: *Remote Method Invocation (RMI)*. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, 2011. – Abgerufen am 19.5.2014

[Pap03] Papazoglou, Mike P.: Service-Oriented Computing: Concepts, Characteristics and Directions. In: *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE)*, IEEE, 2003, S. 3 – 12

[PH13] Pauler, Wolfgang ; Heinfling, Benjamin: *Das beste Netz aller Zeiten*. http://www.chip.de/artikel/Der-haerteste-Handy-Netztest-Deutschlands-Telekom-Vodafone-O2-und-E-Plus-im-Test_63944005.html, September 2013. – Abgerufen am 5.1.2014

[PKKB09] Palmer, Nicholas ; Kemp, Roelof ; Kielmann, Thilo ; Bal, Henri: Ibis for Mobility: Solving Challenges of Mobile Computing Using Grid Techniques. In: *Proceedings of the 10th Workshop on Mobile Computing Systems and Applications (HotMobile)*, ACM, 2009, S. 17:1 – 17:6

[PSS+12] Pawluk, Przemyslaw ; Simmons, Bradley ; Smit, Michael A. ; Litoiu, Marin ; Mankovski, Serge: Introducing STRATOS: A Cloud Broker Service. In: *Proceedings of the 5th International Conference on Cloud Computing (CLOUD)*, IEEE, 2012, S. 891–898

[Rap04] Rappa, Michael A.: The utility business model and the future of computing services. In: *IBM Systems Journal* 43 (2004), Nr. 1, S. 32 – 42

[RDG11] Roy, Nilabja ; Dubey, Abhishek ; Gokhale, Aniruddha: Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In: *Proceedings of the International Conference on Cloud Computing (CLOUD)*, IEEE, 2011, S. 500–507

[RHFN+12] Rodríguez-Haro, Fernando ; Freitag, Felix ; Navarro, Leandro ; Hernánchez-sánchez, Efraín ; Farías-Mendoza, Nicandro ; Guerrero-Ibáñez, Juan A. ; González-Potes, Apolinar: A summary of virtualization techniques. In: *Procedia Technology* 3 (2012), Nr. 0, S. 267 – 272. – ISSN 22120173

[RMK+96] Rekhter, Y. ; Moskowitz, B. ; Karrenberg, D. ; Groot, G. J. ; Lear, E.: *RFC 1918 – Address Allocation for Private Internets.* http://www.rfc-editor.org/info/rfc1918, Februar 1996. – Abgerufen am 19.5.2014

[RR00] Rauber, Thomas ; Rünger, Gudula: *Parallele Und Verteilte Programmierung.* Springer, 2000 (Springer-Lehrbuch). – ISBN 9783540660095

[Sat96] Satyanarayanan, Mahadev: Fundamental Challenges in Mobile Computing. In: *Proceedings of the 15th annual Symposium on Principles of Distributed Computing (PODC)*, ACM, 1996, S. 1 – 7

[Sau13] Sauter, Martin: *Grundkurs Mobile Kommunikationssysteme: UMTS, HSDPA und LTE, GSM, GPRS, Wireless LAN und Bluetooth.* 5. überarbeitete und erweiterte Auflage. Vieweg Verlag, Friedr, & Sohn Verlagsgesellschaft mbH, 2013. – ISBN 978–3–658–01461–2

[SBCD09] Satyanarayanan, Mahadev ; Bahl, Paramvir ; Caceres, Ramon ; Davies, Nigel: The Case for VM-based Cloudlets in Mobile Computing. In: *Pervasive Computing, IEEE* 8 (2009), Nr. 4, S. 14 – 23

[SF05] Su, Ya-Yunn ; Flinn, Jason: Slingshot: Deploying Stateful Services in Wireless Hotspots. In: *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys)*, ACM, 2005, S. 79–92

[SF12] Sadalage, Pramodkumar J. ; Fowler, Martin: *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Pearson Education, 2012. – ISBN 9780133036121

[SH07] Sachs, Lothar ; Hedderich, Jürgen: *Angewandte Statistik: Methodensammlung Mit R.* Springer, 2007. – ISBN 9783540321613

[SKS+11] Saripalli, Prasad ; Kiran, G.V.R. ; Shankar, Ravi R. ; Narware, Harish ; Bindal, Nitin: Load Prediction and Hot Spot Detection Models for Autonomic Cloud Computing. In: *Proceedings of the 4th International Conference on Utility and Cloud Computing (UCC)*, IEEE, 2011, S. 397–402

[SM10] Salomon, David ; Motta, Giovanni: *Handbook of Data Compression.* Springer, 2010. – ISBN 9781848829039

[Sos10] Sosinsky, Barrie: *Cloud Computing Bible.* Wiley, 2010 (Bible Series). – ISBN 9781118023990

[SSW04] Sloss, Andrew ; Symes, Dominic ; Wright, Chris: *ARM System Developer's Guide: Designing and Optimizing System Software.* Elsevier Science, 2004 (The Morgan Kaufmann Series in Computer Architecture and Design). – ISBN 9780080490496

[Sta07] Stallings, William: *Network Security Essentials - Applications and Standards.* Third Edition. Pearson Education, 2007. – ISBN 0132380331

[TJA12] Thomas, Bryce ; Jurdak, Raja ; Atkinson, Ian: SPDYing Up the Web. In: *Communications of the ACM* 55 (2012), S. 64–73. – ISSN 00010782

[TLKF09] Tapia, Pablo ; Liu, Jun ; Karimli, Yasmin ; Feuerstein, Martin: *HSPA Performance and Evolution: A practical perspective*. Wiley, 2009. – ISBN 9780470742051

[Tou02] Toutenburg, Helge: *Lineare Modelle:*. Physica-Verlag, 2002. – ISBN 9783642573484

[TTH05] Tabatabaee, Vahid ; Tiwari, Ananta ; Hollingsworth, Jeffrey K.: Parallel Parameter Tuning for Applications with Performance Variability. In: *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'05)*, 2005, S. 57–68

[Tys99] Tyszer, Jerzy: *Object-Oriented Computer Simulation of Discrete-Event Systems*. Springer, 1999. – ISBN 9780792385066

[Ull12] Ullenboom, Christian: *Java ist auch eine Insel: Das umfassende Handbuch*. Galileo Press, 2012 (Galileo Computing). – ISBN 9783836218023

[VRMCL08] Vaquero, Luis M. ; Rodero-Merino, Luis ; Caceres, Juan ; Lindner, Maik: A Break in the Clouds: Towards a Cloud Definition. In: *SIGCOMM Computer Communication Review* 39 (2008), Nr. 1, S. 50 – 55. – ISSN 01464833

[WBTY11] Wieder, Philipp (Hrsg.) ; Butler, Joe M. (Hrsg.) ; Theilmann, Wolfgang (Hrsg.) ; Yahyapour, Ramin (Hrsg.): *Service Level Agreements for Cloud Computing*. Springer, 2011. – ISBN 9781461416135

[WNLL12] Wang, Wei ; Niu, Di ; Li, Baochun ; Liang, Ben: Dynamic Cloud Resource Reservation via Cloud Brokerage. In: *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2012, S. 400–409

[WSKV08] Wang, Xin ; Schulzrinne, Henning ; Kandlur, Dilip ; Verma, Dinesh: Measurement and analysis of LDAP performance. In: *IEEE/ACM Transactions on Networking* 16 (2008), Februar, Nr. 1, S. 232–243. – ISSN 10636692

[ZBD+03] Zeng, Liangzhao ; Benatallah, Boualem ; Dumas, Marlon ; Kalagnanam, Jayant ; Sheng, Quan Z.: Quality Driven Web Services Composition. In: *Proceedings of the 12th International Conference on World Wide Web (WWW)*, ACM, 2003, S. 411–421

Anhang

Anhang A

Übersicht über die verwendeten Rechner und Mobilgeräte

Für die Durchführung verschiedener Experimente in dieser Arbeit wurden die nachfolgend tabellarisch beschriebenen vier Rechner und Mobilgeräte benutzt.

Tabelle A.1: Verwendete Rechner und Mobilgeräte

Hersteller	DELL	SONY
Name	Streak 7	Xperia P
Gattung	Tablet (7")	Smartphone (4")
Architektur	ARMv7 (Nvidia Tegra 2)	ARMv7 (ST-Ericsson Novathor U8500)
Betriebssystem	Android 3.2	Android 4.1
Displayauflösung	800x480	960x540
CPU-Kerne	2 (Cortex-A9[1]) @ 1 GHz	2 (Cortex-A9[1]) @ 1 GHz
RAM	512 MB	1 GB
WLAN	802.11 b/g/n	802.11 b/g/n
UMTS/HSPA	HSPA	HSPA+

Hersteller	DELL	Asus
Name	Latitude E6530	Eee PC 1000H
Gattung	Laptop (15")	Netbook (10")
Architektur	x86-64 (Intel Core i7-3720QM)	x86 (Intel Atom N270)
Betriebssystem	Linux Mint 13 (Kernel 3.2)	Microsoft Windows XP
Displayauflösung	1920x1080	1024x600
CPU-Kerne	4 @ 2,6 GHz (Turbo 3,6 GHz) + HT[2]	1 @ 1,6 GHz + HT[2]
RAM	8 GB	1 GB
WLAN	802.11 b/g/n	820.11 b/g/n
UMTS/HSPA	HSPA	HSPA+ via USB Dongle

1) Cortex-A9 ist ein 32-bit Prozessordesign (ARMv7 Befehlssatz) mit max. 4 CPU-Kernen.
2) HT $\widehat{=}$ Hyperthreading

Anhang B

Eigenschaften von Diensten ausgewählter IaaS Anbieter

Nachfolgend sind die Angebote einiger IaaS-Anbieter zusammengestellt, die On-Demand Instanzen nach dem Selbstbedienungsprinzip anbieten, diese mindestens auf Stundenbasis abrechnen, in Europa verfügbar sind und Linux-Instanzen anbieten. Preisangaben für Windows sind teilweise zur Information mit angegeben. Die Angebote von *Rackspace Cloud Servers*, *Amazon Elastic Compute Cloud (EC2)*, *GoGrid*, *Windows Azure* und *Google Compute Engine* verwenden das gleiche *pay-as-you-go* Abrechnungsmodell und bieten vorkonfigurierte Instanztypen an. Sie sind gut miteinander vergleichbar.

Flexiscale und *Elastic Hosts* verwenden teilweise *Prepaid* Abrechnungsmodelle und bieten keine festen sondern anpassbare vorkonfigurierte Instanztypen an. Zudem sind diese die einzigen Anbieter, die auch eingehenden Datentransfer Abrechnen.

Rackspace Cloud Servers

Rechenzentren in sechs Regionen: Nord-Virginia (USA), Dallas (USA), Chicago (USA), London (UK), Sydney (Australien) , Hong Kong (China).

Tabelle B.1: Liste verfügbarer Instanztypen von Rackspace Cloud Servers in London (UK)

Typ	**vCPU**	**RAM (GB)**	**Instanz-Speicher (GB)**	**Netzwerkleistung zum Internet in Mbit/s**	**Kosten Linux in £/h (Windows)**
1 GB Performance	1	1	20 SSD	80	0,03 (n.v.)
2 GB Performance	2	2	40+20 SSD	160	0,06 (0,075)
4 GB Performance	4	4	40+40 SSD	320	0,12 (0,15)
8 GB Performance	8	8	40+80 SSD	640	0,24 (0,3)
15 GB Performance	4	15	40+150 SSD	500	0,5 (0,59)
30 GB Performance	8	30	40+300 SSD	1000	1 (1,18)
60 GB Performance	16	60	40+600 SSD	2000	2 (2,36)
90 GB Performance	24	90	40+900 SSD	3000	3 (3,54)
120 GB Performance	32	120	40+1200 SSD	4000	4 (4,72)

vCPU = virtuelle CPU (Anzahl von logischen CPU-Kernen, die einer Instanz zur Verfügung stehen)

Tabelle B.2: Datenübertragungskosten von Rackspace Cloud Servers in London (UK)

Ausgangsbandbreitennutzung	0–10 TB	10–50 TB	50–200 TB	200–500 TB	500– 1024 TB	Mehr als 1024 TB
Kosten in £/GB	0,08	0,07	0,06	0,05	0,04	0,04

Stand 6.3.2014 `http://www.rackspace.com/de/cloud/servers`

Amazon Elastic Compute Cloud (EC2)

Rechenzentren in acht Regionen verfügbar: Europa (Irland), USA Ost (Nord-Virginia), USA West (Oregon), USA West (Nordkalifornien), Asien-Pazifik (Singapur), Asien-Pazifik (Tokio), Asien-Pazifik (Sydney), Südamerika (São Paolo).

Tabelle B.3: Liste verfügbarer Amazon EC2 Instanztypen in EU Irland (Auszug)

Instanz-Familie	**Instanz-Typ**	**Prozessor-architektur**	**vCPU**	**ECUs**	**RAM (GB)**	**Instanz-Speicher (GB)**	**Netzwerkleistung**	**Kosten Linux in US $/h (Windows)**
Allgemeiner Zweck	m1.small	32-Bit od. 64-Bit	1	1	1,7	1 x 160	Niedrig	0,065 (0,091)
Allgemeiner Zweck	m1.medium	32-Bit od. 64-Bit	1	2	3,75	1 x 410	Mittel	0,13 (0,182)
Allgemeiner Zweck	m1.large	64-Bit	2	4	7,5	2 x 420	Mittel	0,26 (0,364)
Allgemeiner Zweck	m1.xlarge	64-Bit	4	8	15	4 x 420	Hoch	0,52 (0,728)
Allgemeiner Zweck	m3.xlarge	64-Bit	4	13	15	nur EBS	Mittel	0,495 (0,702)
Allgemeiner Zweck	m3.2xlarge	64-Bit	8	26	30	nur EBS	Hoch	0,99 (1,404)
Recheneffizienz-optimiert	c1.medium	32-Bit od. 64-Bit	2	5	1,7	1 x 350	Mittel	0,165 (0,225)
Recheneffizienz-optimiert	c1.xlarge	64-Bit	8	20	7	4 x 420	Hoch	0,66 (0,9)
Recheneffizienz-optimiert	cc2.8xlarge	64-Bit	32	88	60,5	4 x 840	10 Gigabit	2,7 (2,97)
RAM-optimiert	m2.xlarge	64-Bit	2	6,5	17,1	1 x 420	Mittel	0,46 (0,51)
RAM-optimiert	m2.2xlarge	64-Bit	4	13	34,2	1 x 850	Mittel	0,92 (1,02)
RAM-optimiert	m2.4xlarge	64-Bit	8	26	68,4	2 x 840	Hoch	1,84 (2,04)
RAM-optimiert	cr1.8xlarge	64-Bit	32	88	244	2 x 120 SSD	10 Gigabit	3,75 (3,831)
Speicher-optimiert	hi1.4xlarge	64-Bit	16	35	60,5	2 x 1024 SSD	10 Gigabit	3,10 (3,58)
Speicher-optimiert	hs1.8xlarge	64-Bit	16	35	117	24 x 2048	10 Gigabit	4,90 (4,931)
Micro-Instances	t1.micro	32-Bit od. 64-Bit	1	variabel	0,615	nur EBS	Sehr niedrig	0,02 (0,035)
GPU-Instances	cg1.4xlarge	64-Bit	16	33,5	22,5	2 x 840	10 Gigabit	2,36 (2,6)

EBS = Elastic Block Store (eine persistente Netzwerkfestplatte einstellbarer Größe und Geschwindigkeit zur Verwendung mit EC2-Instanzen)

Tabelle B.4: Datenübertragungskosten von Amazon EC2 in EU Irland

Ausgangsbandbreitennutzung	0–1 GB	1 GB–10 TB	10–50 TB	50–150 TB	150– 500 TB	Mehr als 500 TB
Kosten in $/GB	0	0,12	0,09	0,07	0,05	individuell

Stand 6.3.2014 `http://aws.amazon.com/de/ec2/`.

GoGrid

Rechenzentren in drei Regionen: San Francisco, Kalifornien (USA), Ashburn, Virginia (USA), Amsterdam, Niederlande. Besonderheit: Windows und Linux kosten gleich

Tabelle B.5: Lister verfügbarer Instanztypen von GoGrid

Instanzfamilie	**Instanz-Typ**	**RAM (GB)**	**vCPU**	**Datenträgergrößen (GB)**	**Netzwerk in Gbit/s**	**Kosten Linux/ Windows in US $/h**
Standard Cloud Servers	X-Small	0,5 GB	shared CPU	25	1	0,03
	Small	1 GB	1	50	1	0,06
	Medium	2 GB	2	100	1	0,12
	Large	4 GB	4	200	1	0,24
	X-Large	8 GB	8	400	1	0,48
	XX-Large	16 GB	16	800	1	0,96
	XXX-Large	24 GB	24	1200	1	1,44
SSD Cloud Servers	Small	2 GB	2	80 SSD	10	0,15
	Medium	4 GB	4	160 SSD	10	0,30
	Large	8 GB	8	320 SSD	10	0,60
	X-Large	16 GB	16	640 SSD	10	1,20
	2X-Large	32 GB	28	1280 SSD	10	2,04
	4X-Large	64 GB	40	2199 SSD	10	2,99
Raw Disk Cloud Servers	Large	8 GB	4	1 x 4000	10	0,60
	X-Large	16 GB	8	3 x 4000	10	1,50
	2X-Large	32 GB	16	6 x 4000	10	3,00
	4X-Large	64 GB	24	12 x 4000	10	5,25
	8X-Large	128 GB	32	24 x 4000	10	8,93
	16X-Large	240 GB	40	45 x 4000	10	17,85

Tabelle B.6: Datenübertragungskosten von GoGrid

Ausgangsbandbreitennutzung	0–1 GB	1 GB–1 TB	1–10 TB	10–50 TB	50– 200 TB	Mehr als 200 TB
Kosten in $/GB	0	0,12	0,11	0,10	0,09	0,08

Stand 6.3.2014 `http://gogrid.com/pricing` `http://gogrid.com/network-pricing`

Windows Azure

Rechenzentren in zehn Regionen: USA Ost, USA West, USA Nord Mitte, USA Süd Mitte, Europa Nord, Europa West, Asien Pazifik Südost, Asien Pazifik Ost, Japan Ost, Japan West.

Tabelle B.7: Liste verfügbarer Instanztypen von Windows Azure in Europa

Instanz-Typ	**vCPU**	**RAM (GB)**	**Datenträgergrößen (GB)**	**Max. Datenträger (jeweils 1 TB)**	**Max. IOPS (500 pro Datenträger)**	**Kosten Linux in €/h (Windows)**
Sehr klein (A0)	shared CPU	0,768	127 + 20	1	1x500	0,015 (0,015)
Klein (A1)	1	1,75	127 + 70	2	2x500	0,045 (0,068)
Mittel (A2)	2	3,5	127 + 135	4	4x500	0,09 (0,135)
Groß (A3)	4	7	127 + 285	8	8x500	0,179 (0,269)
Sehr groß (A4)	8	14	127 + 605	16	16x500	0,358 (0,537)
A5	2	14	127 + 135	4	4x500	0,239 (0,298)
A6	4	28	127 + 285	8	8x500	0,477 (0,596)
A7	8	56	127 + 605	16	16x500	0,961 (1,192)

Tabelle B.8: Datenübertragungskosten von Windows Azure in Europa (Nord und West)

Ausgangsbandbreitennutzung	0–5 GB	5 GB–10 TB	10–50 TB	50–150 TB	150– 500 TB	Mehr als 500 TB
Kosten in €/GB zu ZONE 1 (ZONE 2)	0	0,09 (0,15)	0,07 (0,12)	0,06 (0,10)	0,04 (0,09)	individuell

ZONE 1 = Westen USA, Osten USA, USA (Mitte/Norden), USA (Mitte/Süden), Westeuropa, Nordeuropa
ZONE 2 = Asien Pazifik (Osten), Asien Pazifik (Südosten)

Stand 6.3.2014 `http://www.windowsazure.com/de-de/pricing/details/virtual-machines/`

Google Compute Engine

Rechenzentren weltweit verteilt. Google unterscheidet nur in die Regionen USA und Europa. Es sind ausschließlich Linux-Instanzen verfügbar.

Tabelle B.9: Liste verfügbarer Instanztypen von Google Compute Engine in Europa

Instanz-Typ	**vCPU**	**RAM (GB)**	**GCEUs**	**Kosten Linux in US$/h**
n1-standard-1	1	3,75	2,75	0,114
n1-standard-2	2	7,5	5,5	0,228
n1-standard-4	4	15	11	0,456
n1-standard-8	8	30	22	0,912
n1-standard-16	16	60	44	1,825
n1-highmem-2	2	13	5,5	0,275
n1-highmem-4	4	26	11	0,549
n1-highmem-8	8	52	22	1,098
n1-highmem-16	16	104	44	2,196
n1-highcpu-2	2	1,80	5,5	0,146
n1-highcpu-4	4	3,60	11	0,292
n1-highcpu-8	8	7,20	22	0,584
n1-highcpu-16	16	14,40	44	1,167
f1-micro	1	0,60	shared CPU, keine Garantie	0,021
g1-small	1	1,70	1,4	0.059

GCEU = Google Compute Engine Unit (ein Google-eigenes Performancemaß zum Vergleich der Verarbeitungsgeschwindigkeiten verschiedener Instanztypen, analog zu Amazons Maß ECU)

Tabelle B.10: Datenübertragungskosten von Google Compute Engine

Ausgangsbandbreitennutzung	0–1 TB	1 TB–10 TB	Mehr als 10 TB
Kosten in $/GB zu Americas/EMEA (APAC)	0,12 (0,21)	0,11 (0,18)	0,08 (0,15)

Americas = Wirtschaftsraum Nord-, Zentral- und Südamerika
EMEA = Wirtschaftsraum Europa, Mittlerer Osten und Afrika
APAC = Asiatisch-Pazifischer Raum inkl. Japan

VM-Instanzen von Google Compute Engine haben keinen lokalen Festplattenspeicher. Persistenter Speicher wird über *Compute Engine Disks* bereitgestellt. Dabei handelt es sich um via Netzwerk angeschlossenen Block-Speicher ähnlich zu einem *Storage Area Network (SAN)*.

Stand 6.3.2014 `https://developers.google.com/compute/`

Elastic Hosts

Rechenzentren in neun Regionen: Europa (London Maidenhead, London Portsmouth & Amsterdam), Nordamerika (San Jose, Los Angeles, San Antonio & Toronto), Asien (Hong Kong), Australien (Sydney)

Elastic Hosts bietet keinen vorkonfigurierten Instanztypen an. Instanzen können je nach Bedarf konfiguriert werden. Der Preis berechnet sich dadurch jeweils individuell. Nachfolgende Tabelle fasst die Wahlmöglichkeiten zusammen.

Tabelle B.11: Lister verfügbarer Konfigurationsmöglichkeiten von Elastic Hosts

Eigenschaft	**Wahlmöglichkeit**	**Kosten Linux in €/h**
CPU	1000–20000 MHz	0,03–0,6
RAM	512–32768 MB	0,02–1,28
Festplatte	0–1862 GB	0–0,36
SSD	0–1862 GB	0–1,81

Windows-Lizenzen müssen monatlich hinzugekauft werden. Datentraffic kann monatlich als Paket je nach gewünschter Größe gekauft werden. Darüber hinaus werden die Datenübertragungskosten zu einem höheren Tarif abgerechnet. Prepaid Datentraffic kostet 0,12 €pro GB und darüber 0,24 €pro GB. Es wird sowohl ausgehender wie auch eingehender Datentraffic berechnet.

Stand 6.3.2014 `http://www.elastichosts.com/pricing-information/`

Flexiscale

Flexiscale ist in einem Rechenzentrum im Südosten von England beheimatet. Flexiscale verwendet ein Abstraktes Prepaid-Modell zur Abrechnung und unterscheidet sich dadurch stark von anderen IaaS-Anbietern. Cloud Ressourcen, Datentraffic und andere Dienste werden in *Units* abgerechnet. Diese *Units* müssen vorher von realem Geld gekauft werden. 1000 *Units* kosten beispielsweise 13 € zuzüglich Steuern. Flexiscale bietet verschiedene Kombinationen aus RAM und vCPUS als Instanztypen an. Sie sind in nachfolgender Tabelle zusammengefasst.

Tabelle B.12: Matrix verfügbarer Instanztypen von Flexiscale

	vCPU Cores							
RAM	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0,5Gb	2							
1 Gb	3							
2 Gb	5	6						
4 Gb		10	11	12				
6 Gb			15	16	17	18		
8 Gb				20	21	22	23	24

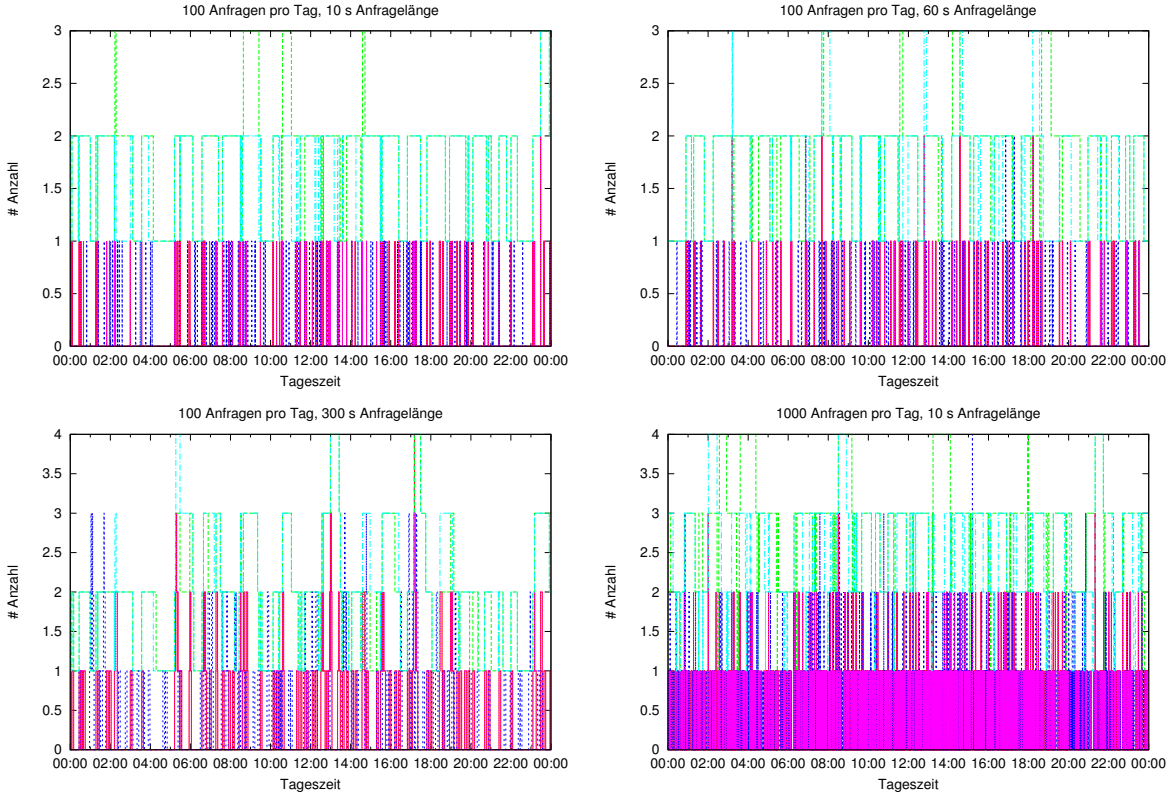
Es wird kein lokaler Festplattenspeicher benutzt. Sämtliche Daten inklusive der VM-Images liegen auf einem SAN. Deshalb muss Festplattenkapazität in zwei Ausprägungen extra bezahlt werden. Einerseits werden pro GB pro Monat 5 *Units* berechnet. Außerdem werden pro GB Datentransfer zu/von der Festplatte weitere 2 *Units* berechnet. Sämtlicher Datentransfer von und nach außen wird mit 5 *Units* pro GB berechnet.

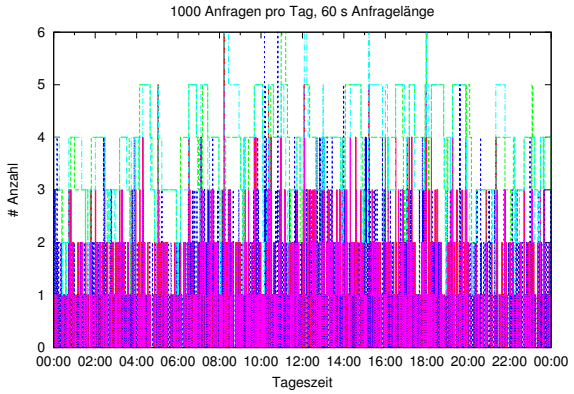
Stand 6.3.2014 `http://www.flexiscale.com/products/flexiscale/pricing/`

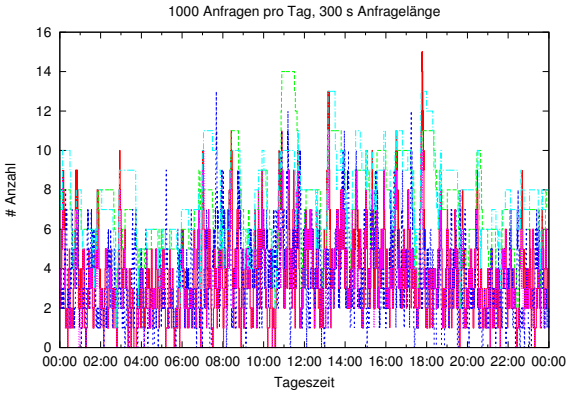
Anhang C

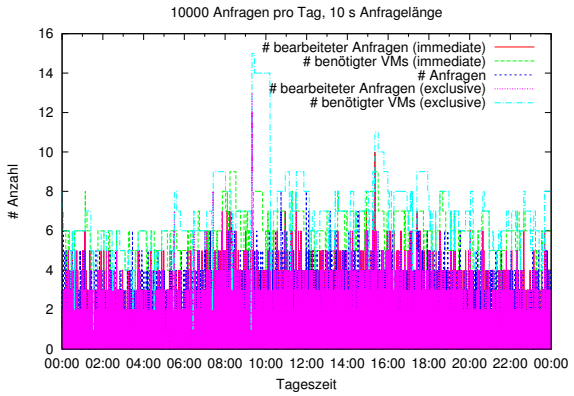
Simulationsergebnisse der Ressourcenallokationsstrategien

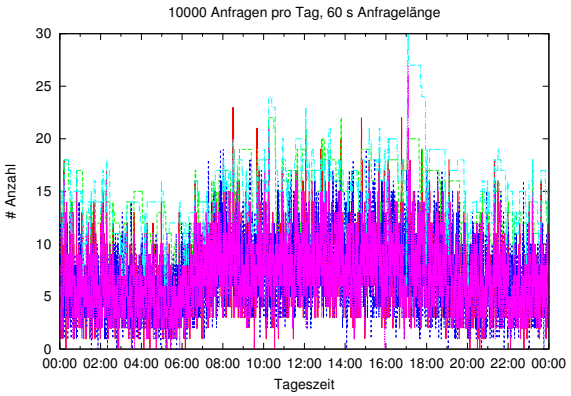
In Kapitel 5.4 wurde durch Simulation das Verhalten zweier Allokationsstretegien für Cloud-Ressourcen untersucht. Dazu wurden unter Verwendung einer möglichst realistischen Anfrageverteilung Simulationen mit verschiedenen Anzahlen von Anfragen und Anfragelängen durchgeführt. Nachfolgend sind die erhaltenen Verteilungen je eines stabilen Simulationslaufs für alle simulierten Konfigurationen dargestellt. Die wichtigsten Ergebnisse der Simulationen sind bereits in Tabelle 5.2 zusammengefasst. Variiert wurde die Anfrageanzahl zwischen 100 – 1000000 Anfragen pro Tag und die Anfragelänge zwischen 2 s – 1200 s, wobei nur die Diagramme der Anfragelängen zwischen 10 s – 300 s angegeben sind, da die restlichen Diagramme jeweils einem der dargestellten stark gleichen. Daraus ergeben sich 15 Konfigurationen. Jedes Diagramm zeigt die Anzahl aktuell in Bearbeitung befindlicher Anfragen und die Anzahl dafür benötigter VM-Instanzen in Abhängigkeit der Tageszeit (jeweils 1 Tag) für beide Allokationsstrategien an. Zusätzlich ist auch die optimale Verteilung angegeben.

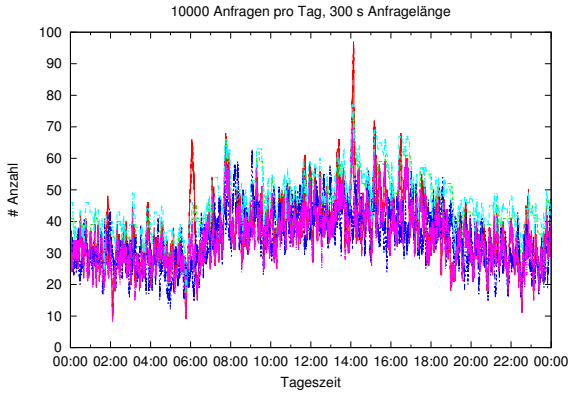


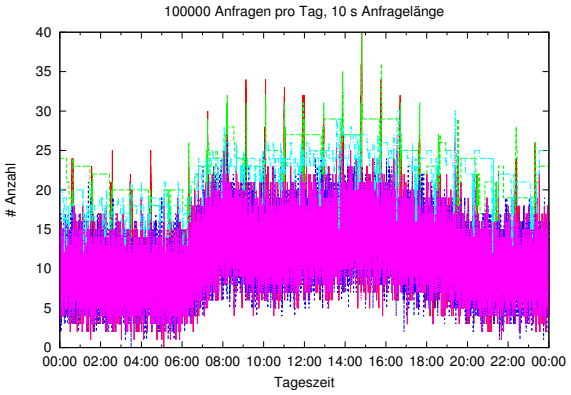

1000 Anfragen pro Tag, 60 s Anfragelänge
Anzahl
Tageszeit
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00

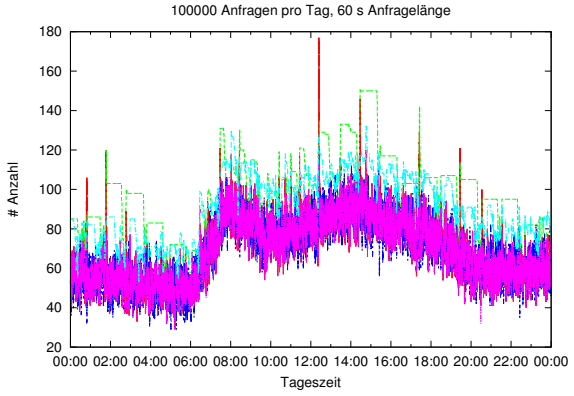

1000 Anfragen pro Tag, 300 s Anfragelänge
Anzahl
Tageszeit
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00

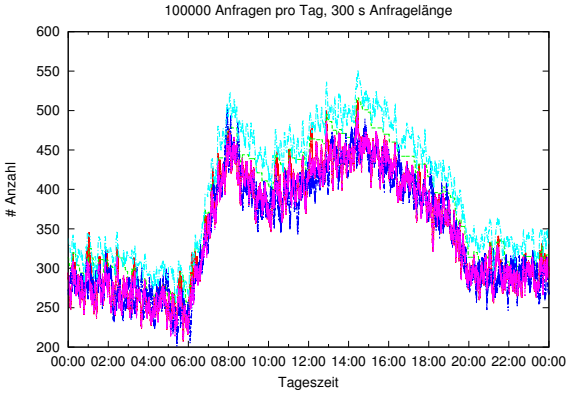

10000 Anfragen pro Tag, 10 s Anfragelänge
bearbeiteter Anfragen (immediate)
benötigter VMs (immediate)
Anfragen
bearbeiteter Anfragen (exclusive)
benötigter VMs (exclusive)
Anzahl
Tageszeit
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00

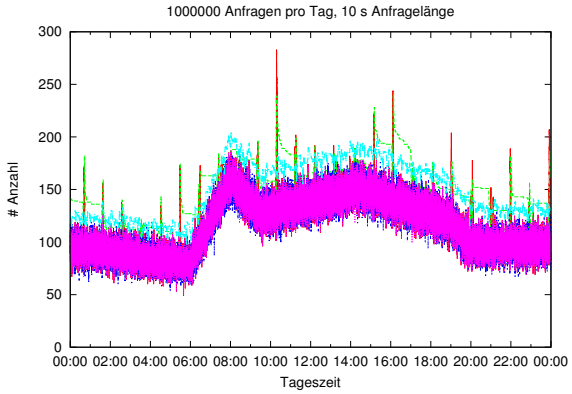

10000 Anfragen pro Tag, 60 s Anfragelänge
Anzahl
Tageszeit
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00

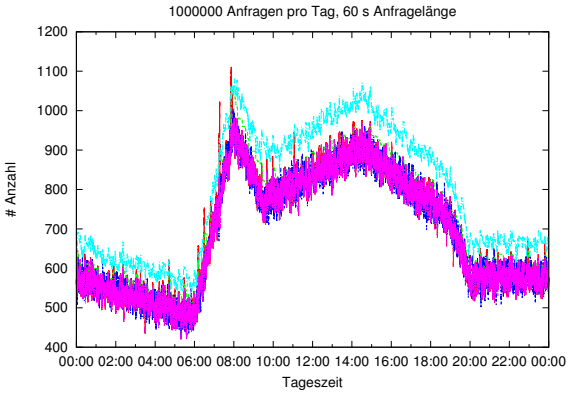

10000 Anfragen pro Tag, 300 s Anfragelänge
Anzahl
Tageszeit
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00

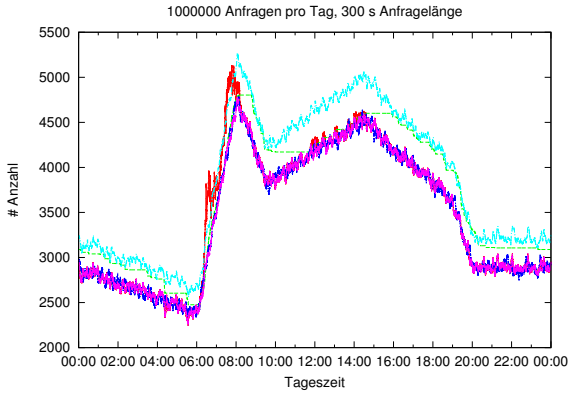

100000 Anfragen pro Tag, 10 s Anfragelänge
Anzahl
Tageszeit
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00


100000 Anfragen pro Tag, 60 s Anfragelänge
Anzahl
Tageszeit
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00


100000 Anfragen pro Tag, 300 s Anfragelänge
Anzahl
Tageszeit
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00


1000000 Anfragen pro Tag, 10 s Anfragelänge
Anzahl
Tageszeit
0
50
100
150
200
250
300
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00


1000000 Anfragen pro Tag, 60 s Anfragelänge
Anzahl
Tageszeit
400
500
600
700
800
900
1000
1100
1200
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00


1000000 Anfragen pro Tag, 300 s Anfragelänge
Anzahl
Tageszeit
2000
2500
3000
3500
4000
4500
5000
5500
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 00:00

Anhang D

WSDL Interface der externen Broker API

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:ns1="http://org.apache.axis2/xsd"
        xmlns:ns="http://master.ferb"
        xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
        xmlns:ax21="http://common.ferb/xsd"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        targetNamespace="http://master.ferb">
    <wsdl:types>
        <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://common.ferb/xsd">
            <xs:complexType name="CloudSession">
                <xs:sequence>
                    <xs:element minOccurs="0" name="endpoint" nillable="true" type="xs:string"/>
                    <xs:element minOccurs="0" name="session_key" nillable="true" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="CloudOption">
                <xs:sequence>
                    <xs:element minOccurs="0" name="cost" nillable="true" type="xs:float"/>
                    <xs:element minOccurs="0" name="remote" nillable="true" type="xs:float"/>
                    <xs:element minOccurs="0" name="type" nillable="true" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
        </xs:schema>
        <xs:schema xmlns:ax22="http://common.ferb/xsd" attributeFormDefault="qualified" elementFormDefault="qualified"
                targetNamespace="http://master.ferb">
            <xs:import namespace="http://common.ferb/xsd"/>
            <xs:element name="deploy">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="name" nillable="true" type="xs:string"/>
                        <xs:element maxOccurs="unbounded" minOccurs="0" name="lauf" nillable="true" type="xs:string"/>
                        <xs:element maxOccurs="unbounded" minOccurs="0" name="index" nillable="true" type="xs:string"/>
                        <xs:element maxOccurs="unbounded" minOccurs="0" name="kosten" type="xs:float"/>
                        <xs:element minOccurs="0" name="VM_image" nillable="true" type="xs:string"/>
                        <xs:element minOccurs="0" name="trafficcost" type="xs:float"/>
                        <xs:element minOccurs="0" name="user" nillable="true" type="xs:string"/>
                        <xs:element minOccurs="0" name="pass" nillable="true" type="xs:string"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="deployResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="return" type="xs:boolean"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="getInstanceForType">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="name" nillable="true" type="xs:string"/>
                        <xs:element minOccurs="0" name="type" nillable="true" type="xs:string"/>
                        <xs:element minOccurs="0" name="user" nillable="true" type="xs:string"/>
                        <xs:element minOccurs="0" name="pass" nillable="true" type="xs:string"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="getInstanceForTypeResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="return" nillable="true" type="ax22:CloudSession"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="undeploy">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="name" nillable="true" type="xs:string"/>
                        <xs:element minOccurs="0" name="user" nillable="true" type="xs:string"/>
                        <xs:element minOccurs="0" name="pass" nillable="true" type="xs:string"/>
                    </xs:sequence>
```

```
            </xs:complexType>
        </xs:element>
        <xs:element name="undeployResponse">
            <xs:complexType>
                <xs:sequence>
                    <xs:element minOccurs="0" name="return" type="xs:boolean"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="calculateMCsingle">
            <xs:complexType>
                <xs:sequence>
                    <xs:element minOccurs="0" name="user" nillable="true" type="xs:string"/>
                    <xs:element minOccurs="0" name="pass" nillable="true" type="xs:string"/>
                    <xs:element minOccurs="0" name="name" nillable="true" type="xs:string"/>
                    <xs:element minOccurs="0" name="subtype" nillable="true" type="xs:string"/>
                    <xs:element maxOccurs="unbounded" minOccurs="0" name="size" type="xs:int"/>
                    <xs:element minOccurs="0" name="t_comm" type="xs:float"/>
                    <xs:element minOccurs="0" name="t_mobil" type="xs:float"/>
                    <xs:element minOccurs="0" name="downloadsize" type="xs:int"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="calculateMCsingleResponse">
            <xs:complexType>
                <xs:sequence>
                    <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="ax22:CloudOption"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="calculateMCmulti">
            <xs:complexType>
                <xs:sequence>
                    <xs:element minOccurs="0" name="user" nillable="true" type="xs:string"/>
                    <xs:element minOccurs="0" name="pass" nillable="true" type="xs:string"/>
                    <xs:element minOccurs="0" name="name" nillable="true" type="xs:string"/>
                    <xs:element minOccurs="0" name="subtype" nillable="true" type="xs:string"/>
                    <xs:element maxOccurs="unbounded" minOccurs="0" name="size" type="xs:int"/>
                    <xs:element minOccurs="0" name="f_comm" type="xs:float"/>
                    <xs:element minOccurs="0" name="f_mobil" type="xs:float"/>
                    <xs:element minOccurs="0" name="downloadsize" type="xs:int"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="calculateMCmultiResponse">
            <xs:complexType>
                <xs:sequence>
                    <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="ax22:CloudOption"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:schema>
</wsdl:types>
<wsdl:message name="deployRequest">
    <wsdl:part name="parameters" element="ns:deploy"/>
</wsdl:message>
<wsdl:message name="deployResponse">
    <wsdl:part name="parameters" element="ns:deployResponse"/>
</wsdl:message>
<wsdl:message name="getInstanceForTypeRequest">
    <wsdl:part name="parameters" element="ns:getInstanceForType"/>
</wsdl:message>
<wsdl:message name="getInstanceForTypeResponse">
    <wsdl:part name="parameters" element="ns:getInstanceForTypeResponse"/>
</wsdl:message>
<wsdl:message name="calculateMCsingleRequest">
    <wsdl:part name="parameters" element="ns:calculateMCsingle"/>
</wsdl:message>
<wsdl:message name="calculateMCsingleResponse">
    <wsdl:part name="parameters" element="ns:calculateMCsingleResponse"/>
</wsdl:message>
<wsdl:message name="undeployRequest">
    <wsdl:part name="parameters" element="ns:undeploy"/>
</wsdl:message>
<wsdl:message name="undeployResponse">
    <wsdl:part name="parameters" element="ns:undeployResponse"/>
</wsdl:message>
<wsdl:message name="calculateMCmultiRequest">
    <wsdl:part name="parameters" element="ns:calculateMCmulti"/>
</wsdl:message>
<wsdl:message name="calculateMCmultiResponse">
    <wsdl:part name="parameters" element="ns:calculateMCmultiResponse"/>
</wsdl:message>
<wsdl:portType name="SOAPServerPortType">
    <wsdl:operation name="deploy">
        <wsdl:input message="ns:deployRequest" wsaw:Action="urn:deploy"/>
        <wsdl:output message="ns:deployResponse" wsaw:Action="urn:deployResponse"/>
    </wsdl:operation>
    <wsdl:operation name="getInstanceForType">
```

```
            <wsdl:input message="ns:getInstanceForTypeRequest" wsaw:Action="urn:getInstanceForType"/>
            <wsdl:output message="ns:getInstanceForTypeResponse" wsaw:Action="urn:getInstanceForTypeResponse"/>
        </wsdl:operation>
        <wsdl:operation name="calculateMCsingle">
            <wsdl:input message="ns:calculateMCsingleRequest" wsaw:Action="urn:calculateMCsingle"/>
            <wsdl:output message="ns:calculateMCsingleResponse" wsaw:Action="urn:calculateMCsingleResponse"/>
        </wsdl:operation>
        <wsdl:operation name="undeploy">
            <wsdl:input message="ns:undeployRequest" wsaw:Action="urn:undeploy"/>
            <wsdl:output message="ns:undeployResponse" wsaw:Action="urn:undeployResponse"/>
        </wsdl:operation>
        <wsdl:operation name="calculateMCmulti">
            <wsdl:input message="ns:calculateMCmultiRequest" wsaw:Action="urn:calculateMCmulti"/>
            <wsdl:output message="ns:calculateMCmultiResponse" wsaw:Action="urn:calculateMCmultiResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="SOAPServerSoap11Binding" type="ns:SOAPServerPortType">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <wsdl:operation name="deploy">
            <soap:operation soapAction="urn:deploy" style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="getInstanceForType">
            <soap:operation soapAction="urn:getInstanceForType" style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="calculateMCsingle">
            <soap:operation soapAction="urn:calculateMCsingle" style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="undeploy">
            <soap:operation soapAction="urn:undeploy" style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="calculateMCmulti">
            <soap:operation soapAction="urn:calculateMCmulti" style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="SOAPServer">
        <wsdl:port name="SOAPServerHttpSoap11Endpoint" binding="ns:SOAPServerSoap11Binding">
            <soap:address location="https://server:8443/BrokerWS/services/SOAPServer.SOAPServerHttpSoap11Endpoint/"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

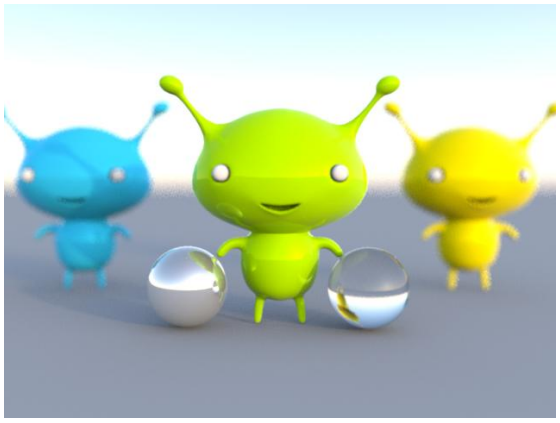
Anhang E

Beispielszenen der Testapplikation Raytracing

In Kapitel 8.2 wird anhand der Software *Sweethome 3D* die Anwendbarkeit und Leistungsfähigkeit der entwickelten Middleware untersucht. Dazu werden via Raytracing fotorealistische Bilder von 3D-Szenen erstellt. Das Raytracing wird dabei durch die separate aber in *Sweethome 3D* integrierte Software *Sunflow* durchgeführt. Nachfolgend sind die verwendeten Szenen visualisiert. Da *Sunflow* auch als eigenständige Software verfügbar ist, wurde eine eigene Testszene benutzt.

Beispielszene Sunflow

Entnommen aus den offiziellen Beispielszenen. (Verfügbar unter Onlineressource `http://sourceforge.net/projects/sunflow/files/sunflow-data/v0.07.1/` (Stand 15.5.2014))

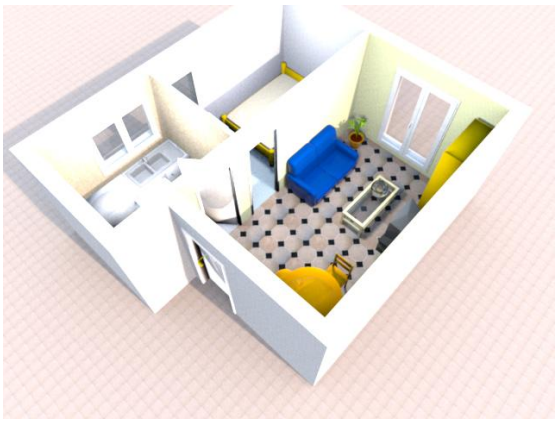


Aliens Shiny

Offizielle Beispielszenen Sweethome 3D

Entnommen aus den offiziellen Beispielszenen. (Verfügbar unter Onlineressource `http://www.sweethome3d.com/gallery.jsp` (Stand 15.5.2014))

SweetHome3DExample



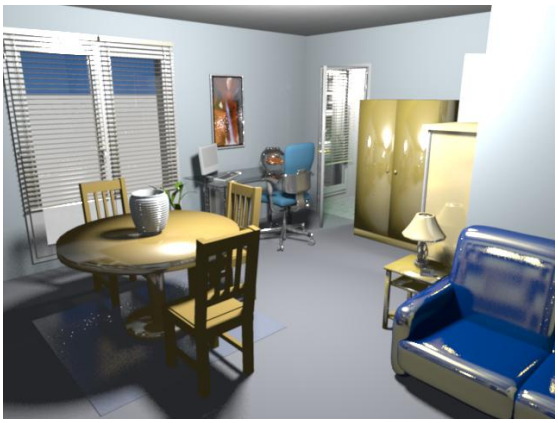
SweetHome3DExample2

SweetHome3DExample3

SweetHome3DExample4

SweetHome3DExample5

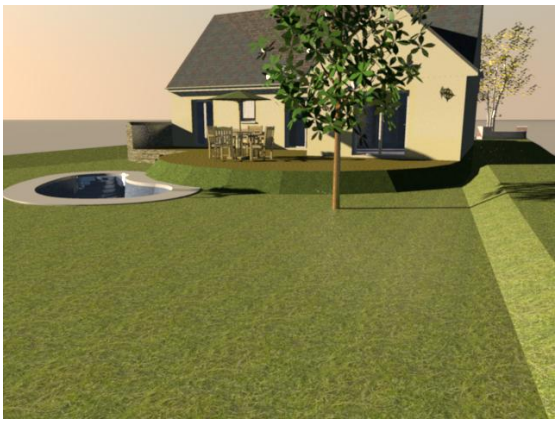
SweetHome3DExample6



userGuideExample

Selbst konstruierte Beispiele

test1



test2

test3

test4

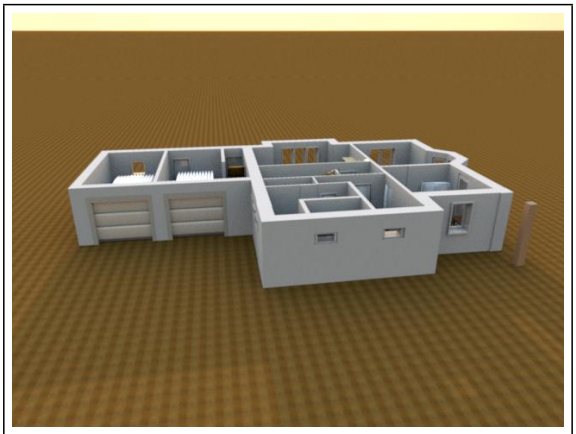
Weitere Beispiele Dritter

extern1

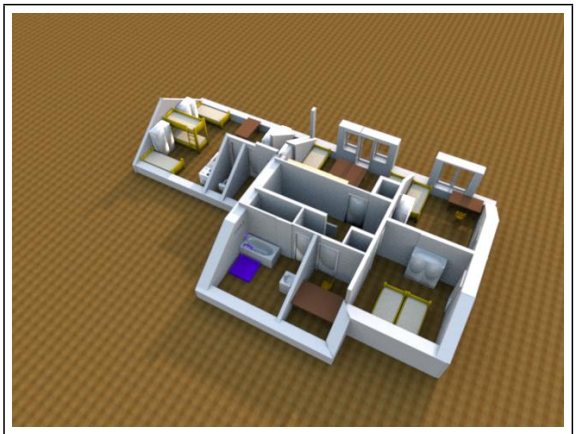
extern2

extern3

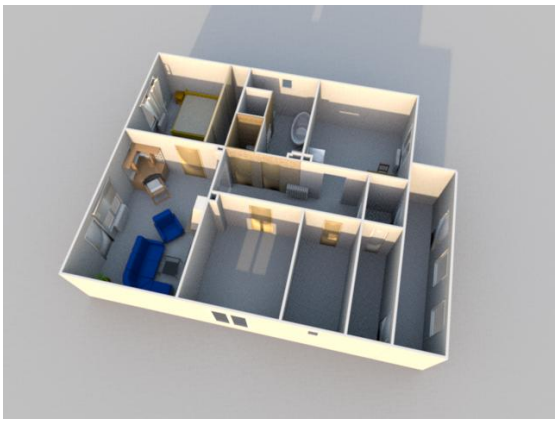
extern4



extern5

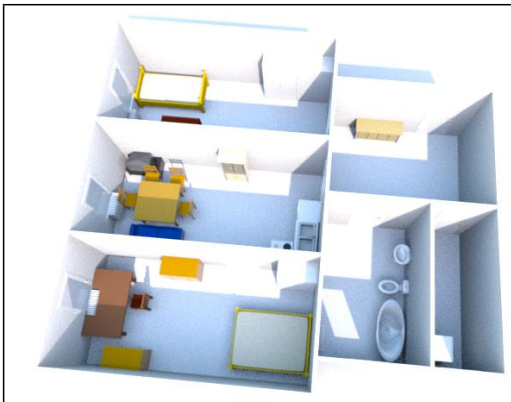


extern6

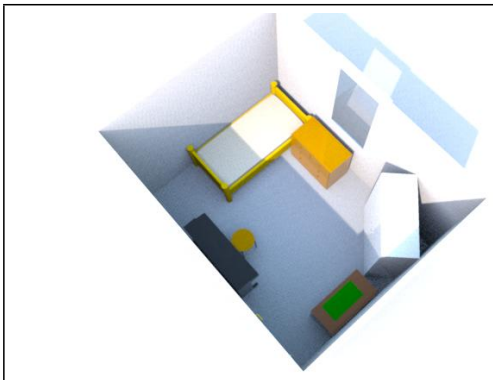


extern7

extern8



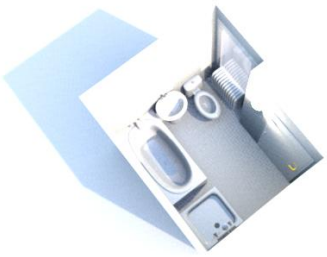
extern9



extern10

extern11

extern12



extern13

extern14

extern15

extern16

extern17

extern18

extern19

extern20

extern21