Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Anna Tertilt
MASTER THESIS

Last- und Performancetest des Open Data Services

Eingereicht am 29.05.2015

Betreuer: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Zirndorf, den 29.05.2015

License

_____

Zirndorf, den 29.05.2015

Abstract

Load and performance testing is an established technique for the performance evaluation of a software system. In this thesis this technique is applied for a quantitative and qualitative evaluation of the performance characteristics of the *JValue Open Data Service* and the *JValue Complex Event Processing Service* used by the mobile application *PegelAlarm*. For performing the evaluation this thesis is answering the questions on how the expected user behavior of the app *PegelAlarm* looks like and how it could be simulated. Starting from these answers a *JMeter* script is implemented that is used to generate the expected load on the services, based on the defined number of users and the intensity of usage. The system under test is observed under different load scenarios in a controlled test environment. Based on the analysis of the test results performance bottlenecks and optimization points are identified. A number of suggestions for further development and operation are derived, which will, when implemented, result in better performance characteristics of the analyzed system.

Zusammenfassung

Last- und Performancetests sind eine etablierte Technik zur Evaluierung der Performance-Eigenschaften eines Softwaresystems. In dieser Arbeit erfolgt der Einsatz eines solchen Last- und Performancetests zur quantitativen und qualitativen Bewertung der Performance-Eigenschaften des *JValue Open Data Services* und des *JValue Complex Event Processing Services* am Anwendungsfall der App *PegelAlarm*. Dafür geht diese Arbeit der Fragestellung nach, wie das erwartete Nutzerverhalten der App *PegelAlarm* aussieht und mit welchen Werkzeugen es für den Last- und Performancetest abgebildet werden kann. Hierzu wird ein *JMeter*-Skript implementiert, das die erwartete Last, basierend auf der erwarteten Benutzerzahl und der Nutzungsintensität, simuliert. Im Rahmen des Last- und Performancetests wird das System in einer kontrollierten Testumgebung den definierten Lastszenarien ausgesetzt und sein Performance- und Ressourcennutzungsverhalten untersucht. Auf Basis der Analyse der Testergebnisse werden Performance-Schwachstellen des Systems sowie Optimierungspunkte aufgezeigt. Eine Reihe an Verbesserungsvorschlägen für die Weiterentwicklung und den Betrieb werden abgeleitet, die nach ihrer Umsetzung zu besseren Performance-Eigenschaften des Systems führen werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AF	Anwendungsfall
API	Application Programming Interface
CEPS	Complex Event Processing Service
EPL	Event Processing Language
GC	Garbage Collection
GCM	Google Cloud Messaging
GUI	Graphical User Interface
HLUG	Hochwasservorhersagezentrale von Hessisches Landesamt für Umwelt und Geologie
HND	Bayerischer Hochwassernachrichtendienst
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JSON	JavaScript Object Notation
KPI	Key Performance Indicators
L&P-Test	Last- und Performance-Test
MBT	Modellbasiertes Testen
NFA	nichtfunktionale Anforderungen
ODS	Open Data Service
OSM	Open Street Map
REST	Representational State Transfer
SUT	System Under Test
TF	Testfall
XML	Extensible Markup Language

1 Einleitung

Die Informationsversorgung der Bevölkerung in Zeiten von Naturkatastrophen ist überlebenswichtig. Mit zunehmender Verbreitung mobiler Endgeräte werden mobile Anwendungen immer bedeutendere Informationsressourcen dabei – die Menschen wenden sich mehr und mehr an ihre mobilen Endgeräte während einer Naturkatastrophe (Wade, 2012). Dieser Trend blieb auch den Entwicklern von mobilen Anwendungen, sogenannten Apps, nicht verborgen. Auf dem Markt sind Lösungen für unterschiedliche Aufgaben erschienen[1]: Warnmeldungen, Sensordaten- und Gefahrenkarten, Nachrichten- oder Kommunikationsservices, Bildungsapps zum Katastrophenverhalten etc. (Sung, 2011).

Der JValue Open Data Service[2] (ODS), entwickelt im Rahmen eines Projekts an der Forschungsgruppe für Open-Source-Software der Friedrich-Alexander-Universität Erlangen-Nürnberg, sammelt und konsolidiert Daten aus mehreren öffentlichen Quellen und stellt sie über eine REST-API in einem einheitlichen Format zur Verfügung. Unter anderem sind die Wasserstandsdaten der großen Flüsse Deutschlands, stammend von *PEGELONLINE* – Gewässerkundliches Informationssystem der Wasser- und Schifffahrtsverwaltung des Bundes[3] –, dort zu finden.

Der JValue Complex Event Processing Service[4] (CEPS) ist ein Benachrichtigungsdienst für ODS mit dem Hauptziel, bestimmte Ereignisse aus großen Datenmengen zu erfassen, zu analysieren und zu extrahieren. Die Benutzer werden mit CEPS in die Lage versetzt, eine Reihe von Bedingungen für die Daten zu beschreiben, und werden durch die Interaktion zwischen ODS und CEPS informiert, wenn diese Bedingungen erfüllt werden.

PegelAlarm[5] ist eine mobile App, die auf Basis von ODS und CEPS gebaut ist. PegelAlarm bietet den Menschen in Deutschland eine Unterstützung in Zeiten potenzieller Hochwassergefahren und versorgt Benutzer mit aktuellen Wasserstandsdaten. Diese Funktionalität wird jedoch auch von vielen anderen Lösungen bereitgestellt: So bietet das Internetangebot des Bayerischen Hochwassernachrichtendienstes (HND) für Smartphones[6] aktuelle Hochwasser-Informationen (unter anderem Rohdaten der Pegelstände) aus Bayern und ist in der Bevölkerung sehr populär. Die Seite des HND wurde in den Zeiten des Hochwasserhöhepunkts im August 2006 mehr als ein halbes Million Mal aufgerufen (LfU, 2006). Während des Junihochwassers 2014 zählte der HND sogar mehr als 800.000 Seitenbesuche (LfU, 2014). Die App PegelAlarm bietet jedoch den Benutzern die zusätzliche attraktive Möglichkeit, für eine oder mehrere Messstationen Alarme zu registrieren, die ausgelöst werden, wenn der Wasserstand des Flusses einen bestimmten Schwellenwert übersteigt. Der Grundgedanke dahinter ist, dass die Menschen, die in den Regionen mit erhöhtem Hochwasserrisiko wohnen, in der Regel wissen, wie hoch der naheliegende Fluss steigen darf, bevor die Gefahr ernsthaft für sie wird.

Aufgrund der spezifischen Funktionalität der App ist das erwartete Nutzerverhalten recht speziell: die

[1] http://www.disaster-relief.org/disaster-mobile-apps.htm
[2] https://github.com/jvalue/open-data-service
[3] http://pegelonline.wsv.de
[4] https://github.com/jvalue/cep-service
[5] https://github.com/jvalue/hochwasser-app
[6] http://m.hnd.bayern.de

meiste Zeit wird eine geringe App-Nutzung erwartet – dann, wenn kein Hochwasser ansteht und die Nutzer gegebenenfalls gelegentlich aus Interesse den Pegelstand ihres Flusses inspizieren oder einen Alarm registrieren. In der Hochwassersaison hingegen muss mit einem massiven Anstieg der Nutzerzahl gerechnet werden. Aus diesem Grund sind die Performance- und Skalierungseigenschaften der App und vor allem der zugrunde liegenden Services von besonderer Bedeutung. So führten die hohen Zugriffszahlen auf das Internet-Angebot des Bayerischen Landesamtes für Umwelt während des Hochwassers im August 2006 zu Laufzeitproblemen beim Abruf der Pegelständen (LfU, 2006).

Performance ist ein wichtiges Qualitätsmerkmal von Softwaresystemen (Williams & Smith, 1998). Nutzer interpretieren schlechte Performance oft als einen Indikator für eine schlechte Quality of Service (Menascé, 2002). Die Entscheidung für oder gegen die Akzeptanz neuer Anwendungen geschieht oft aufgrund ihrer Performance (Forrester Consulting, 2009). Jedoch sind Performance-Fehler auch in schon abgenommener und freigegebener Software weit verbreitet (Jin, Song, Shi, Scherpelz, & Lu, 2012). Sie führen zu beschädigten Kundenbeziehungen, Produktivitätsverlust für die Benutzer bei der Arbeit mit dem System, verlorenen Umsätzen und verpassten Markt-Möglichkeiten (Williams & Smith, 1998), zu Kostenüberschreitungen, Ausfällen im Einsatz und sogar zur Aufhebung des Projekts (Woodside, Franks, & Petriu, 2007). Dem Prozess des Performance-Tests muss daher, zur Identifizierung dieser Performance-Fehler, besondere Beachtung gewidmet werden (Barber, 2004a). So reduziert sich die Anzahl an Performance-Fehlern, die im Betrieb behoben werden müssen, von 100% auf 30% für Unternehmen, die sich Zeit für Performance-Tests (auch erst in den späten Phasen des Software-Lebenszyklus) genommen haben (Molyneaux, 2014). Aus diesen Gründen wird eine rechtzeitige Untersuchung der Performance-Eigenschaften der der App PegelAlarm zugrunde liegenden Services – ODS und CEPS – als besonders wichtig eingestuft.

(Jin et al., 2012) zeigen, dass die Performance-Probleme, die aufgrund eines schnell wechselnden Workloads entstehen, schwer zu vermeiden sind. In Hinblick auf die Vorbereitung auf die Hochwassersaison 2015/2016 ist damit eine rechtzeitige Untersuchung der Performance-Eigenschaften des Systems von großer Bedeutung. Diese geschieht im Rahmen dieser Arbeit.

Zentrale Forschungsgegenstände dieser Arbeit sind:

- Wie muss ein Last- und Performancetest (L&P-Test) für das System (ODS und CEPS) implementiert werden, um zuverlässige Aussagen über sein Performanceverhalten machen zu können?
- Wie ist das gemessene Laufzeit- und Ressourcenverhalten des Systems zu interpretieren und zu bewerten?
- Welche Optimierungsmöglichkeiten des Performanceverhaltens des Systems können auf Basis der Testergebnissen hergeleitet werden?

Die Arbeit ist strukturiert wie folgt: Kapitel 2 präsentiert die für die Arbeit notwendigen Grundlagen und stellt L&P-Tests als das ausgewählte Evaluierungsvorgehen vor. In Kapitel 3 werden die Funktionalität und die Architektur des Systems unter Test (SUT) beschrieben. Kapitel 4 liefert Informationen zur Testvorbereitung und Testdurchführung. Im Kapitel 5 werden die Testergebnisse zusammengefasst. Kapitel 6 stellt eine Reihe von Verbesserungsvorschlägen für die App und die ihr zugrundeliegenden Services dar,

diskutiert die Einschränkungen des Vorgehens und gibt einen Ausblick auf mögliche weitere Arbeiten. Die Zusammenfassung beinhaltet eine Übersicht über die Ergebnisse der Arbeit.

2 Grundlagen

Die in diesem Kapitel durchgeführte Literaturanalyse verortet das Thema *Software-Performance* im Kontext der Softwarequalität und bildet eine Grundlage für die Gestaltung und Durchführung der L&P-Tests in dieser Arbeit.

2.1 Softwarequalität

Qualität ist ein komplexer und vielfältiger Begriff (Garvin, 1984) und damit schwer zu bestimmen. Die Qualität eines Softwaresystems ist jedoch ein kritischer Faktor in Softwareprojekten (Chow & Cao, 2008; Reel, 1999) und Basis für deren Erfolg. Softwarequalität wird nach ISO 9126 als „Gesamtheit der Funktionalitäten und Merkmale eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen“, definiert. Die Qualität eines Softwaresystems ist nicht so einfach zu messen (Lochmann & Goeb, 2011). Es wurden verschiedene Qualitätsmodelle entwickelt, mit dem Ziel, Softwarequalität zu beschreiben. Die bekanntesten sind die bereits erwähnte ISO 9126 und ihre Nachfolgenorm ISO 25010. Sie beschreiben High-Level-Qualitätsmerkmale von Software aus den beiden folgenden Perspektiven und werden bei der Bewertung der Softwarequalität verwendet:

- Die Perspektive der *Produktqualität*, die aus acht Qualitätsmerkmalen besteht (Funktionale Tauglichkeit, Performance, Kompatibilität, Gebrauchstauglichkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit, Übertragbarkeit), die sich auf die statischen Eigenschaften der Software sowie auf die dynamischen Eigenschaften des Computersystems beziehen (Bautista, Abran, & April, 2012).
- Die Perspektive der *Nutzungsqualität*, die sich aus fünf Eigenschaften zusammensetzt (Effektivität, Effizienz, Zufriedenheit, Freiheit von Risiken, Kontextabdeckung), die durch den Endbenutzer während der Ausführung der Software wahrgenommen werden (Castillo, Losavio, Matteo, & Bøegh, 2010).

Diese Arbeit beschäftigt sich mit den internen Performanceeigenschaften eines Softwaresystems und konzentriert sich damit auf die Produktqualität. Die Qualitätsmerkmale, die sich auf die Performance eines Softwaresystems beziehen, werden folgend detaillierter dargestellt.

Performance ist als die erbrachte Leistung, relativ zu der Menge an Ressourcen, die unter festgelegten Bedingungen verwendet werden (z.B. andere Softwareprodukte, Software- und Hardwarekonfiguration des Systems, Speicher- und Verbrauchsmaterialien) definiert. Sie setzt sich aus drei Untermerkmalen zusammen (beschrieben nach (Maier, Schmitt, & Rost, 2014)):

- Zeitverhalten – Grad, zu dem die Antwort- und Bearbeitungszeiten sowie die Durchsatzraten eines Produkts oder Systems bei der Ausführung seiner Funktionen die Anforderungen erfüllen.
- Ressourcennutzung – Grad, zu dem die Menge und Typen von Ressourcen, die von einem Produkt oder System bei der Ausführung seiner Funktionen genutzt werden, die Anforderungen erfüllen. Menschliche Ressourcen werden hierbei normalerweise nicht berücksichtigt.
- Kapazität – als Höchstgrenze, bis zu der ein Produkt oder System die Anforderungen erfüllt.

Performance ist ein wichtiger Teil der Softwarequalität, da sie die Nutzerzufriedenheit stark beeinflusst.

2.2 Performance-Evaluierungsvorgehen

Allgemein sind folgende drei Performance-Evaluierungsverfahren nach (Jain, 1991) bekannt und beschrieben:

- *Analytische Modellierung* dient der Vorhersage des Performance-Verhaltens des Systems mittels mathematischer Berechnungen, z. B. mit dem Ansatz von Warteschlangen.
- *Simulation* ermöglicht die Prognose zukünftiger Situationen unter angepassten Bedingungen.
- *Messung* als Überwachung des tatsächlichen Verhaltens eines realen Softwaresystems, während es kontrolliert einer Last ausgesetzt ist.

Im Grunde unterscheiden sich die Verfahren maßgeblich darin, in welchem Stadium seines Lebenszyklus sich das zu untersuchende Softwaresystem befindet: Während die analytische Modellierung und Simulation gute Techniken für die Einschätzungen der Performance-Eigenschaften auch in den früheren Phasen des Entwicklungsprozesses sind, benötigen Messverfahren ein lauffähiges Artefakt. Zudem streben die drei Verfahren auch unterschiedliche Primärziele bei ihrem Einsatz an: analytische Modelle und Simulation eignen sich gut für die Analyse von Änderungen an verschiedenen System-Konfigurationsparametern, die Messung dagegen dient im Allgemeinen dem Vergleich zwischen dem Ist- und dem Soll-Stand oder zwischen mehreren Systemen (Jain, 1991).

Die Zeit, die in eine Performance-Untersuchung investiert werden muss, variiert je nach dem ausgewählten Verfahren: Am schnellsten können die ersten Ergebnisse aus den Modellierungsverfahren extrahiert werden, Messungen hingegen bringen vielen Kosten (wie z. B. Testumgebungsaufbau) und Risiken (Testumgebung ist nicht repräsentativ, da der Produktionsumgebung nicht gleich ist) mit sich, was in einer große Zeitvarianz resultiert. Simulationen nehmen im Allgemeinen viel Zeit in Anspruch, bis die ersten Ergebnisse zur Verfügung stehen. Ein weiteres Unterscheidungsmerkmal der drei Vorgehen ist die Genauigkeit der Evaluierungsergebnisse. Die Messverfahren weisen in der Regel die beste Präzision auf, da sie im Gegensatz zur Modellierung und zur Simulation keine Annahmen und Vereinfachungen über das System vornehmen, sondern das System so wie es ist prüfen (Jain, 1991). Nichtdestotrotz hängt die Exaktheit der Messergebnissen von vielen Faktoren ab, wie z.B. Testumgebung, Systemkonfiguration, Workload etc.

2.3 Last- und Performance-Test

Ein Performance-Test ist eine analytische Software-Qualitätssicherungsmaßnahme und wird nach IEEE Standard[7] als eine Art von Test definiert, um „die Einhaltung der für ein Softwaresystem oder eine Komponente festgelegten Leistungsanforderungen zu bewerten“. Der Performance-Test bestimmt damit die Performance-Eigenschaften eines Softwareprodukts.

Werkzeuggestützt wird das zu untersuchende System in einer kontrollierten Umgebung einer erwarteten Menge an Last ausgesetzt, während sein Verhalten mithilfe weiterer Werkzeuge beobachtet wird. Die erhobenen Messdaten werden aggregiert und analysiert und gegen die Akzeptanzkriterien geprüft.

[7] IEEE Std 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology, http://dictionary.ieee.org/index/p-5.html

Die grundsätzliche Herangehensweise an die Durchführung von Performance-Tests einer Anwendung umfasst nach (Meier, Farre, Bansode, Barber, & Rea, 2007) die Identifikation der folgenden zentralen Aspekte:

- Testziele,
- wichtigste Nutzungsszenarien des Systems,
- Workload, der die erwartete Last beschreibt und sie realistisch auf die Nutzungsszenarien verteilt,
- Metriken, die erhoben werden (in Hinblick auf die definierten Testziele),
- Werkzeuge, die eingesetzt werden.

Im folgenden Abschnitt werden diese Aspekte näher erläutert.

Mit der Durchführung eines Performance-Tests werden unterschiedliche Ziele verfolgt. (Meier et al., 2007) unterscheiden zwischen folgenden:

- Untersuchung der Bereitschaft für den produktiven Einsatz,
- Evaluierung der Einhaltung von Akzeptanzkriterien,
- Vergleich der Performance-Eigenschaften von unterschiedlichen Systemen oder von unterschiedlichen Konfigurationen eines Systems,
- Ermittlung der Ursachen von Performance-Problemen,
- Unterstützung bei der Durchführung der System-Optimierungen,
- Bestimmung des möglichen Durchsatzes.

Entsprechend den Testzielen werden in (Meier et al., 2007) folgende Variationen des Performance-Tests definiert:

- Der *Lasttest* ist eine Form des Performance-Tests, die durchgeführt wird, um das Verhalten eines Systems oder einer Komponente unter Normal- und Hochlastbedingungen zu ermittelt.
- Der *Stresstest* ist eine Form des Performance-Tests, die durchgeführt wird, um „ein System oder eine Komponente an oder über den Grenzen, die in den Anforderungen spezifiziert wurden, zu bewerten“[8].
- Ein *Kapazitätstest* ergänzt den Lasttest und stellt fest, wie viel Last ein System oder eine Komponente unterstützen kann ohne dabei die in den Anforderungen spezifizierten Performanceziele zu verletzen.

Jede mögliche Benutzeraktivität in einem Performance-Test zu simulieren ist unpraktisch, wenn nicht gar unmöglich (Meier et al., 2007). In der Praxis wird oft aus Gründen der Aufwandsreduktion nur ein Teil aller möglichen Szenarien simuliert. Hierbei sind die aus Performance-Gesichtspunkten kritischsten Szenarien zu wählen. Zu diesen kritischsten Anwendungsszenarien werden die häufigsten, geschäftskritischsten und performance-intensivsten Szenarien gezählt.

Eine der wichtigsten Herausforderungen bei der Vorbereitung von L&P-Tests ist die Erzeugung der richtigen Last (Abbors, Ahmad, Truscan, & Porres, 2012). Sollten die Benutzer in der realen Welt sich

[8] Nach IEEE 610

anderes verhalten als die während des Tests simulierten Benutzer, entspricht die im Test erzeugte Last nicht der realen Last auf dem System. Ein Rückschluss aus den Testergebnissen auf die zu erwartende Performance des Systems ist in diesem Fall nicht nur nicht möglich, sondern kann gegebenenfalls sogar zu Fehlentscheidungen in Form unnötiger, aufwändiger Performanceoptimierungen, oder dem Ausbleiben notwendiger Performanceoptimierungen, führen.

Die Performance eines Softwaresystems kann in unterschiedlichen Performance-Kennzahlen beschreiben werden. Die relevantesten Performance-Kennzahlen nach (Fowler, 2002) sind:

- *Antwortzeit*, definiert als die Zeitspanne vom Absenden des ersten Bits einer Anfrage bis zum Empfang des letzten Bits der Antwort. Somit schließt sie die Netzlaufzeiten der Anfrage zum Server und zurück mit ein.
- *Latenz* wird durch die Netzlaufzeiten von Client-Anfrage und Server-Antwort definiert. Sie ist ein Overhead, der auf jede fachlich-bedingte Antwortzeit des Servers dazuzurechnen ist und enthält Routing, Authentifizierung und Verschlüsselung.
- *Durchsatz* ist eine Rate (Anzahl an Arbeitsschritten pro Zeiteinheit) in der die Anfragen vom System bearbeitet werden können: z. B. Anfrage/Sek, Hits/Sec, etc.
- *Lastsensitivität* beschreibt, wie stark sich das Antwortzeit- oder Durchsatzverhalten verändern, wenn die Last auf der Anwendung zunimmt, und ist ein Maß der Skalierbarkeit des Systems.
- *Effizienz* wird als die erreichte Performance dividiert durch die verwendeten Ressourcen verstanden.
- *Kapazität* beschreibt den maximalen Durchsatz oder die maximale Last, die das System erreichen oder vertragen kann, bis es weitere Anfragen ablehnt oder die Grenzwerte einer Performance-Kennzahl überschreitet.
- *Skalierbarkeit* ist ein Maß für die Kapazitätserweiterung, die durch das Hinzufügen weiterer Ressourcen (z. B. Hardware) erreicht werden kann.

Diese Liste kann mit weiteren Hardware-bedingten Hauptleistungsindikatoren (engl.: Key Performance Indicators (KPI)) vervollständigt werden (nach (Meier et al., 2007)): Prozessorkapazität, Speichernutzung, Festplattenkapazität, Netzwerkkapazität.

Die Auswahl der passenden Kennzahlen ist ein wichtiger Schritt bei einer Performance-Untersuchung (Jain, 1991): basierend auf diesen Kennzahlen wird auf die Performance-Eigenschaften des Systems geschlossen. Die Auswahl an aussagekräftigen Metriken muss in Hinblick auf den Einsatzkontext und die Randbedingungen gemacht werden.

Im Gegensatz zu funktionalen Tests, die auch manuell durchgeführt werden können, ist eine L&P-Test-Durchführung immer automatisiert. Für die Performance-Untersuchung eines Softwaresystems werden mindestens zwei Werkzeugtypen nötig (Jain, 1991):

- Ein Werkzeug, um die Last auf dem System zu produzieren (Lasttreiber-Werkzeug zur Simulation virtueller Benutzer)
- Ein Werkzeug, um die Ergebnisse zu überwachen (Monitoring-Werkzeug)

Auf dem Markt sind inzwischen viele Werkzeuge präsent (siehe auch Abschnitt 4.5.1.2). Je nach Herangehensweise und Implementierung liefern sie jedoch teilweise unterschiedliche Testergebnisse (vergleiche (Suffian & Fahrurazi, 2012)). Bei der Auswahl von Werkzeugen ist besonders wichtig zu verstehen, wie sich die Last zusammensetzt und wie die Daten erhoben werden. Nur auf diese Weise ist eine valide Interpretation der gemessenen Ergebnisse möglich.

Der Prozess der Durchführung von L&P-Tests ergibt sich aus folgenden Aktivitäten (nach (Meier et al., 2007)):

1. Bestimmung der Testumgebung,
2. Bestimmung der Akzeptanzkriterien für den Performance-Test,
3. Testplanung und –design,
4. Einrichtung der Testumgebung,
5. Umsetzung des Testdesigns,
6. Testausführung,
7. Testanalyse, -auswertung und ggf. Wiederholung der Testausführung.

2.4 Bottleneck

Der Hauptgrund für die Notwendigkeit von L&P-Tests besteht darin, dass Softwaresysteme nicht linear skalieren, d.h. mit zunehmender Anzahl an Anfragen nicht gleichmäßig stärker belastet werden. Vielmehr bilden einzelne Komponenten, z.B. die Datenbank bei komplexen Anfragen oder die Netzwerkkomponenten bei großem Transfervolumen, sogenannte Performance-Bottlenecks. Diese Komponenten erreichen ihre maximale Auslastung bereits vor den anderen Komponenten und bilden damit ein limitierendes Element für die Maximallast des Gesamtsystems. (Jain, 1991) definiert den Begriff des Performance-Bottlenecks als eine Ressource mit höchster Auslastung. (Barber, 2004a) unterscheidet zwischen einem Performance-bedingten Ausfall, einem Performance-Bottleneck und einer sogenannten „langsamen Stelle“ (engl.: slow spot). Ein kompletter Stillstand des Systems ist danach ein Performance-bedingter Ausfall. Ein Bottleneck ist dagegen eine Verlangsamung, die nicht nur unter Last zu beobachten ist. Ein Bottleneck beeinflusst die Nutzung eines Systems und wirkt sich auch auf die Arbeit anderer Benutzer aus. Ist dies nicht der Fall und die beobachtete Verlangsamung löst keine weiteren spürbaren Folgen oder Effekte aus, so handelt es sich nur um eine „langsame Stelle“ und kein Bottleneck.

In (Woodside et al., 2007) wird ein Bottleneck-Muster (engl.: bottleneck pattern) wie folgt beschreibt: Eine Ressource *R* ist ein Kandidat für ein Bottleneck, falls:

- sie in der Mehrzahl aller Szenarien verwendet wird,
- viele Szenarien, die diese Ressource nutzen, zu langsam sind,
- die Ressource an ihrer Sättigungsgrenze ist (mehr als 80% der Ressource sind beschäftigt),
- Ressourcen, die vorab akquiriert oder anschließend freigegeben werden, ebenfalls an ihrer Sättigungsgrenze sind.

L&P-Testergebnisse sind jedoch nur als Indizien für ein mögliches Bottleneck zu sehen und sind von den Testdurchführungsbedingungen stark abhängig (Workload, Systemparameter, Umgebungskonfiguration).

Die im L&P-Test identifizierten Bottleneck-Kandidaten müssen im nächsten Schritt in Form einer Code- oder Architekturanalyse bestätigt werden. Ein Schlüsselkriterium für die Identifizierung eines Bottlenecks ist die Reproduzierbarkeit der L&P-Testergebnisse (Barber, 2004a).

Aus der White-Box-Sicht auf die Software-Performance spricht man bei Auffälligkeiten von Performance-Bugs. (Jin et al., 2012) definieren diese als ineffiziente Codesequenzen, die zu erheblichen Performance-Einbußen und Ressourcenverschwendung führen und vom Compiler nicht wegoptimiert werden können. Durch zumeist relativ einfache Quellcode-Änderungen an solchen Stellen kann oft eine erhebliche Beschleunigung der Software erreicht werden, unter Beibehaltung der ursprünglichen Funktionalität. Die White-Box-Sicht wird in dieser Arbeit teilweise eingenommen, indem im L&P-Test auffallende Schwachstellen einer Codeanalyse unterzogen werden.

2.5 Workload

Der Workload ist einer der wichtigsten Konfigurationsparameter eines L&P-Tests, da er die Anzahl und das Verhalten der erwarteten Nutzer auf dem betrachteten System abbildet. Die Aussagekraft der L&P-Testergebnisse hängt damit grundlegend von der Definition des Workloads ab. (Jain, 1991) definiert den Workload als die von den Benutzern an das System gerichteten Anfragen. (Meier et al., 2007) präzisieren die Definition und verstehen unter dem Workload einen auf ein System, eine Anwendung oder eine Komponente gerichteten Stimulus, der in Bezug auf die konkurrierenden Zugriffe und Dateneingaben die Systemnutzung simuliert. Nach (Meier et al., 2007) umfasst die Workload-Charakterisierung die absolute Anzahl an Nutzern, die Anzahl parallel aktiver Nutzer, Daten- und Transaktionsmengen, das Transaktionsmixmuster sowie die Wartezeiten (engl.: think times) zwischen einzelnen Aktionen. Ein realer Benutzer überlegt zwischen Aktionen, was er als nächstes macht, und belastet das System in dieser Zeit nicht. (Denaro, Polini, & Emmerich, 2004) fügen noch die Frequenz der Anfragen, die Ankunftsrate der Client-Anfragen und die Testlaufzeit als Workload-Charakteristika dazu.

Die Varianz der getesteten Workloads ermöglicht das Verhalten des Systems unter verschiedenen im Betrieb erwarteten Lastszenarien zu untersuchen. Dabei muss die auf das System gerichtete Last möglichst realistisch abgebildet werden, damit aus den Ergebnissen des L&P-Tests direkt auf das zu erwartende Performance-Verhalten der Anwendung im Betrieb geschlossen werden kann. Das erfordert ein tiefes Verständnis des Businesskontextes der Anwendung.

Offener und geschlossener Workload

(Jin et al., 2012) haben in einer umfangreichen Studie über fünf repräsentative Software-Suiten festgestellt, dass Performance-Bugs dann am häufigsten entstehen, wenn die Workload-Vorstellungen der Entwickler mit den realen Workloads im Betrieb nicht übereinstimmen. (Schroeder, Wierman, & Harchol-Balter, 2006) heben den Gedanken hervor, dass die Genauigkeit der Performance-Untersuchungen davon abhängt, wie exakt der Workload des Systems unter Test modelliert wird und unterscheiden zwischen einem offenen Workload (engl.: open workload), geschlossen Workload (engl.: closed workload) und teilweise offenem Workload (engl.: partly-open workload) (Abbildung 1).

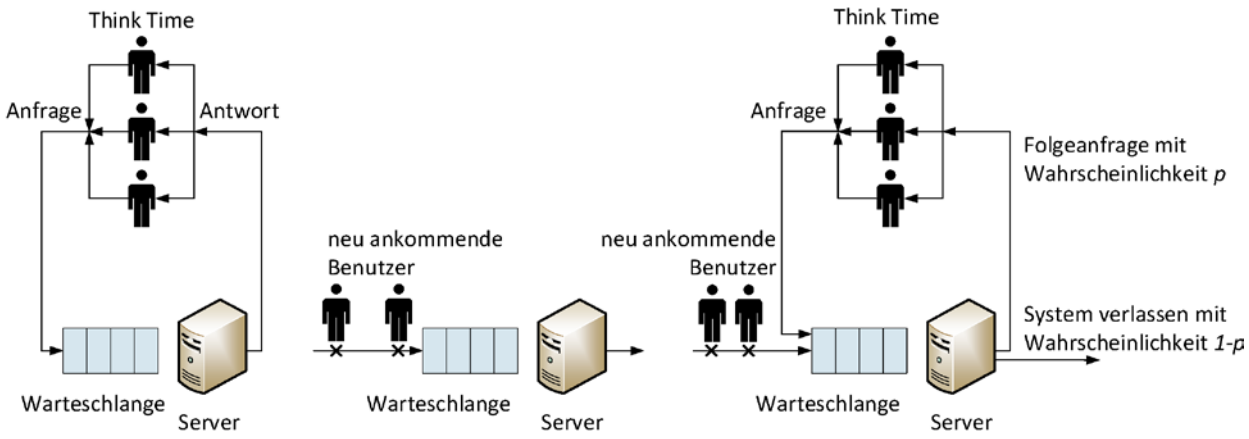

Abbildung 1 (a) geschlossener Workload, (b) offener Workload, (c) teilweise offener Workload. (Quelle: Eigene Darstellung, nach (Schroeder et al., 2006))

In einem geschlossenen Workload wird die Ankunft von neuen Nutzeranfragen nur von der Fertigstellung bereits laufender Anfragen ausgelöst, wie in Abbildung 1 (a) dargestellt. Es wird von einer fixen Anzahl an Benutzern ausgegangen, die das System kontinuierlich nutzen. Jeder dieser Benutzer wiederholt die gesamte Testlaufzeit hinweg zwei Schritte: zuerst wird eine Anfrage ausgeführt, anschließend wird auf die Antwort gewartet. Danach pausiert er für eine gewisse Zeit bis er eine neue Anfrage initiiert.

In einem offenen Workload ist das Auftreten neuer Anfragen unabhängig von der Fertigstellung der bestehenden Anfragen (Abbildung 1 (b)). Es gibt einen Strom von neu ankommenden Benutzern mit einer durchschnittlichen Ankunftsrate. Es wird angenommen, dass jeder Benutzer eine Anfrage beim System einreicht, auf die Antwort wartet und anschließend das System verlässt. Das Unterscheidungsmerkmal eines offenen Systems ist, dass eine Fertigstellung einer Anfrage keine neue Anfrage auslösen wird: eine neue Anfrage wird nur durch die Ankunft eines neuen Benutzers initiiert.

Das Nutzerverhalten der meisten Anwendungen wird am besten über eine Kombination der beiden Modelle abgebildet (Abbildung 1 (c)). Benutzer kommen neu in das System mit einer definierten Ankunftsrate. Jedes Mal, wenn eine Anfrage bearbeitet wird, bleibt der Benutzer mit einer Wahrscheinlichkeit p im System und macht eine Folgeanfrage (evtl. nach einer Think Time). Mit der Wahrscheinlichkeit $1-p$ verlässt der Benutzer das System.

3 System unter Test

Eine wichtige Anforderung für die meisten Arten des Testes eines Softwaresystems ist ein gutes Verständnis dessen, was die Software leistet (Elfar & Whittaker, 2001). Ziel dieser Arbeit ist die Vorbereitung und Durchführung eines L&P-Tests des JValue Open Data Service (ODS) im Zusammenspiel mit dem JValue Complex Event Processing Service (CEPS) unter Einsatz in der App PegelAlarm. Auf Basis von Primärliteratur zu ODS, CEPS und PegelAlarm (Eichhorn, 2014; Reischl, 2014; Tsysin, 2015) werden im Folgenden die für die Arbeit relevanten Aspekte der Architektur des Systems unter Test (SUT) beschrieben Die Architekturbeschreibung folgt im Wesentlichen dem Template des arch42-Projekt[9].

3.1 Zweck des Systems

Die für die Android-Plattform 4.4 entwickelte App PegelAlarm ist eine native mobile Web-Anwendung (im Gegensatz zu mobilen Standalone-Anwendungen), d. h. um vollständig funktionieren zu können benötigt sie eine kontinuierliche Verbindung zu den ihr zugrundeliegenden Services ODS und CEPS, die sie mit den benötigten Daten versorgen: das Zusammenspiel zwischen ODS, CEPS und der PegelAlarm-App stellt die Funktionalität eines Hochwasser-Alarms bereit. PegelAlarm nimmt Messstationen und Wasserstandswerte als Parameter vom Anwender entgegen und benachrichtigt sie, wenn der Wasserstand auf den entsprechenden Stationen über diese voreingestellten Werte gestiegen ist. Außerdem kann die App Auskünfte über die aktuellen Wasserstände geben und sie über ein Zeitintervall hinweg aufzeichnen.

Die beiden Services ODS und CEPS, die die Basis für diese Funktionalitäten bereiten, bieten jedoch Lösungen für viele weitere generische Probleme an: ODS dient als Lösung für die zunehmende Datenflut, indem er heterogene Daten aus verschiedenen für die Öffentlichkeit zur Verfügung stehenden externen Datenquellen konsumiert und sie in einem einheitlichen Format über eine zentrale REST-Schnittstelle zur Verfügung stellt. Einen weiteren Mehrwert der Datennutzung aus dem ODS bietet zudem auch das in ODS implementierte Konzept der Datenqualitätsverbesserung, sowie die durch die Anbindung des CEPS-Services zur Verfügung stehende Benachrichtigungs-Funktionalität. CEPS konsumiert die Daten aus ODS und dient dem Ziel, beliebige Ereignisse aus diesen großen Datenmengen zu erfassen, zu analysieren und zu extrahieren. Somit bietet CEPS seinen Clients eine Möglichkeit, die Verarbeitung und Evaluation der Ereignissen aus großen Datenmengen auszulagern und nur über das Ergebnis informiert zu werden. CEPS ist ein separater Service und abstrahiert ODS von der direkten Kommunikation mit der Esper-Engine, um mögliche Lizenz-Konflikte zu vermeiden.

3.2 Kontextabgrenzung

Abbildung 2 zeigt das Zusammenspiel von PegelAlarm, ODS und CEPS mit den wichtigsten Akteuren. Dabei bilden ODS und CEPS gemeinsam das SUT, das von der App PegelAlarm benutzt wird, und unterliegen den L&P-Tests.

Die App PegelAlarm selber ist nicht Bestandteil des Tests und ist in Rahmen dieser Arbeit vor allem im Kontext der von ihr verursachten Aktivitäten auf ODS und CEPS wichtig. Die von ODS benutzten

[9] www.arc42.de

Datenquellen, v. A. *PEGELONLINE*, sowie weitere Software-Systeme, die die Funktionalitäten der beiden Services theoretisch benutzen könnten, stehen ebenfalls nicht im Fokus. Die Interaktion mit dem Google Cloud Messaging Service[10] (GCM), der von CEPS für die Server-Client-Kommunikation benutzt wird, wird ebenfalls nur bis zu dem Punkt, an dem eine Nachricht CEPS verlässt, betrachtet.

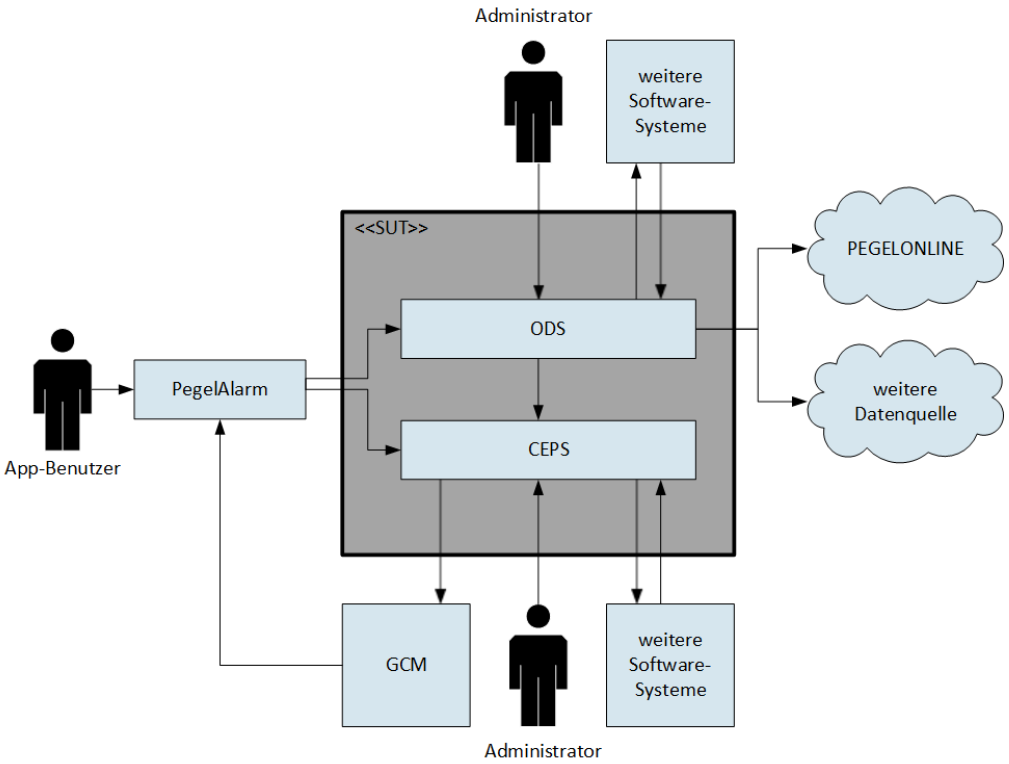

Abbildung 2 PegelAlarm, ODS und CEPS im Zusammenspiel mit wichtigen Akteuren, Pfeile stellen Initiierung eines Aufrufs dar. (Quelle: Eigene Darstellung)

Im Folgenden werden die dargestellten Benutzer und Fremdsysteme kurz erläutert.

Während hinter jedem App-Benutzer ein Mensch steht, der über die grafische Oberfläche der App die bereitgestellten Funktionen in Anspruch nimmt, sind die ODS- und CEPS-Services primär für eine Maschine-zu-Maschine-Kommunikation designed: ODS konsumiert die Daten von externen Datenquellen, aktualisiert regelmäßig seinen Datenbestand und stellt die Daten über eine REST-Schnittstelle weiter zur Verfügung. Die Anzahl an Fremdsystemen, mit denen ODS kommuniziert, ist variabel und hängt von der vom Systemadministrator vorgenommenen Konfiguration ab. Für eine solche Konfiguration bietet ODS eine weitere REST-Schnittstelle an. Je nach Datenquelle liest ODS die Daten über verschiedene Schnittstellen (z. B. die Quelle *PEGELONLINE*[11] stellt die Daten über REST oder SOAP-Schnittstellen im JSON-Format bereit).

CEPS nimmt ebenfalls Konfigurationen vom Administrator oder anderen Fremdsystemen entgegen: die Datenquellen, aus deren Daten der Service Ereignisse extrahieren wird, sowie die entsprechenden EPL-Anweisungen, die die Muster, nach denen die Ereignisse gesucht werden, definieren (jedoch als EPL-Adapter, ohne konkrete Parameter, diese werden im Allgemeinen von End-Benutzer erwartet). Sobald ein

[10] https://developer.android.com/google/gcm/index.html
[11] https://www.pegelonline.wsv.de

Ereignis erkannt wird, kann CEPS Benachrichtigungen an die entsprechenden Benutzer oder Fremdsysteme verschicken (HTTP-Benachrichtigung) bzw. weiterleiten (GCM-Benachrichtigung). Zudem bietet CEPS die Möglichkeit, die Daten, die eine Benachrichtigung ausgelöst haben, über eine REST-Schnittstelle abzufragen.

Die funktionalen Anforderungen an beide Services sind in der Primärliteratur (Eichhorn, 2014; Reischl, 2014; Tsysin, 2015) beschrieben und stehen in dieser Arbeit nicht im Fokus. Es wird davon ausgegangen, dass das SUT die geforderte Funktionalität bietet. Hingegen stellen die nichtfunktionalen Anforderungen (NFAs) die behandelten Themen dieser Arbeit dar. Die vorhandenen Formulierungen der NFA in der Primärliteratur (Eichhorn, 2014; Reischl, 2014; Tsysin, 2015) verletzen jedoch das Testbarkeits-Kriterium, das eine gute Anforderung erfüllen soll, und lassen sich nicht direkt testen. Die NFA müssen daher im Rahmen dieser Arbeit konkretisiert werden (siehe Abschnitt 4.2).

3.3 Architekturziele

Die Grundlagen für die wichtigsten Design-Entscheidungen bei der Entwicklung des Systems bietet die Übersicht der Architekturziele (Tabelle 1), die aus der genannten Primärliteratur abgeleitet wurden. Diese sind für das Verständnis des Systemaufbaus, der Wahl der Anwendungsstruktur und der eingesetzten Technologien von großer Bedeutung.

Tabelle 1 Architekturziele von PegelAlarm, ODS und CEPS.

Priorität	Qualitätsmerkmal	Architekturziel	Beschreibung
PegelAlarm			
1	Effizienz	Effizienz	Ressourcennutzung (Batterie) muss minimiert werden.
ODS			
1	Funktionalität	Interoperabilität	Das System arbeitet mit anderen vorhandenen Systemen erfolgreich zusammen und erlaubt den Austausch von Daten.
2	Benutzbarkeit	Bedienbarkeit	Der Aufwand für den Benutzer/die andere Anwendung, ODS zu bedienen, muss minimiert werden.
3	Erweiterbarkeit	Erweiterbarkeit	Die Architektur muss an neue Anforderungen angepasst werden können (z. B. Hinzufügen von neuen Datenquellen, neue API anbieten).
4	Effizienz	Performance	Die Anfragen an ODS sollen schnell verarbeitet werden. Ein Wachstum der Daten und der Nutzerzahl soll bewältigt werden können.
5	Wartbarkeit	Wartbarkeit	Der ODS soll einfach auf Veränderungen in den Anforderungen angepasst werden können.
CEPS			
1	Funktionalität	Interoperabilität	Das System soll mit anderen vorhandenen Systemen erfolgreich zusammenarbeiten und den Austausch von Daten erlauben.
2	Funktionalität	Sicherheit	Client-EPL-Anweisungen dürfen keinen Einfluss auf die EPL-Anweisungen von anderen Clients nehmen.

3.4 Schnittstellenbeschreibung

In der Abbildung 3 sind die wichtigsten Schnittstellen des SUT dargestellt.

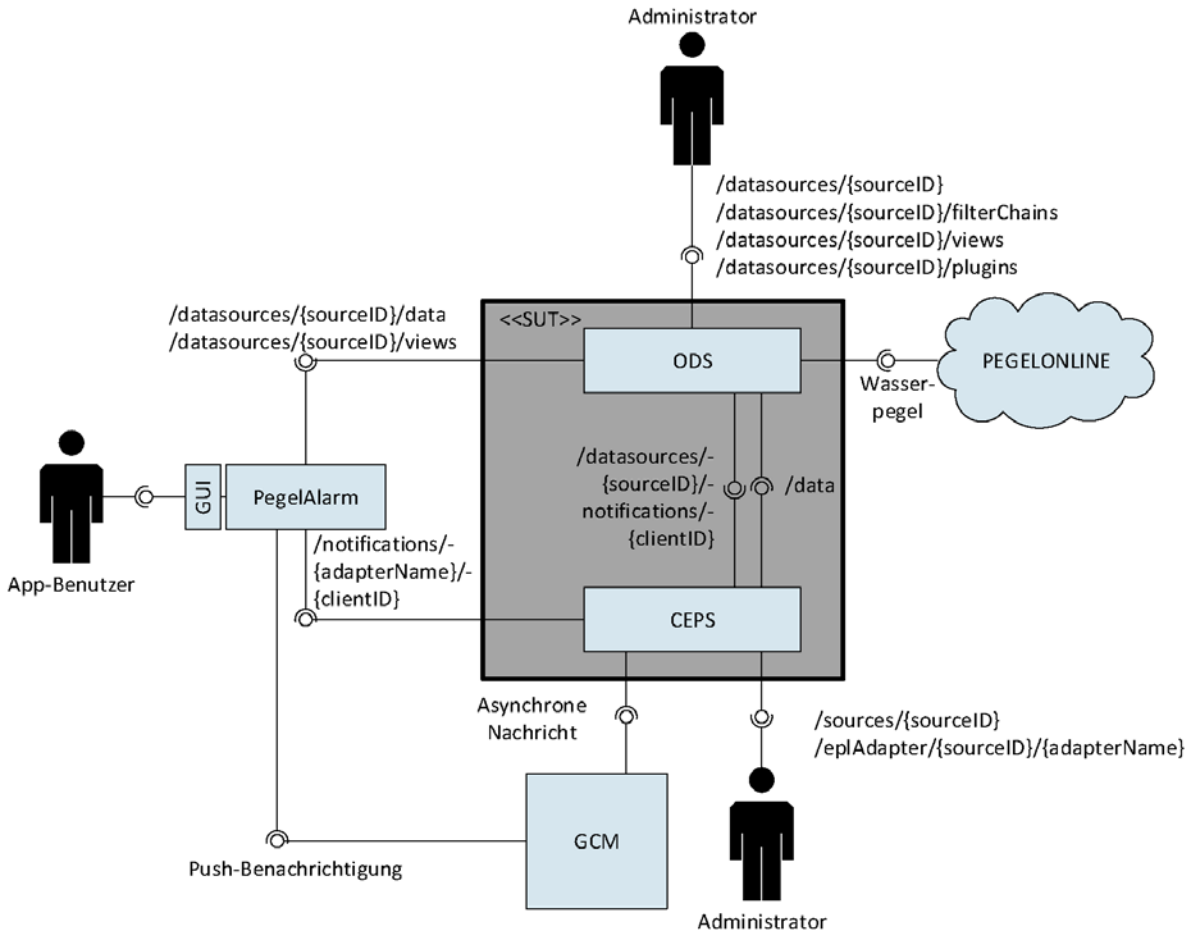

Abbildung 3 Die relevanten Schnittstellen des SUT. (Quelle: Eigene Darstellung)

Die gesamte Kommunikation zwischen der App *PegelAlarm* und ODS – das Abfragen der aktuellen Daten und der periodische Synchronisierungsvorgang – erfolgt über die REST-Schnittstelle.

Die Client-Server-Kommunikation zwischen der App und CEPS – Übergabe der Client-spezifischen Parameter für die Registrierung eines Alarms – erfolgt über die REST-Schnittstelle, die Server-Client-Kommunikation nutzt GCM für das Verschicken von asynchronen Nachrichten an mobile Endgeräte. Aktuell verpflichtet sich GCM jedoch zu keinen Dienstgütevereinbarungen (Service-Level-Agreements) und garantiert eine Nachrichtenzustellung nicht (Stand: März 2015). Aus diesem Grund ist es unmöglich, das Verhalten von GCM im Rahmen eines L&P-Tests korrekt nachzubilden. Das SUT endet somit an der Schnittstelle zu GCM, letzteres wird von Test explizit ausgeschlossen.

Die Kommunikation zwischen ODS und CEPS erfolgt über eine einfache Push-basierte Benachrichtigungs-Architektur, die in den ODS integriert ist: ODS agiert als Publisher und leitet neuen Daten von entsprechenden Datenquellen sofort an allen registrierten Clients weiter. CEPS wird als Subscriber über eine REST-Schnittstelle bei ODS registriert und bekommt die neuen Daten automatisch, ebenfalls über eine REST-Schnittstelle, übergeben.

3.5 Laufzeitsicht

Ein typischer Arbeitsablauf beginnt mit einem App-Benutzer, der die Anwendung *PegelAlarm* startet. Beim ersten Start wird ein einmaliger Synchronisierungsvorgang mit dem ODS ausgelöst, so dass der Benutzer sofort mit dem Erkunden der Wasserstände anfangen kann. ODS auf seiner Seite aktualisiert in gewissen Zeitabständen seine Datenbestände, indem er periodisch frische Daten aus den registrierten Datenquellen anfordert. In Abbildung 4 sind die Abläufe der zwei häufigsten Anwendungsfälle der App zu sehen:

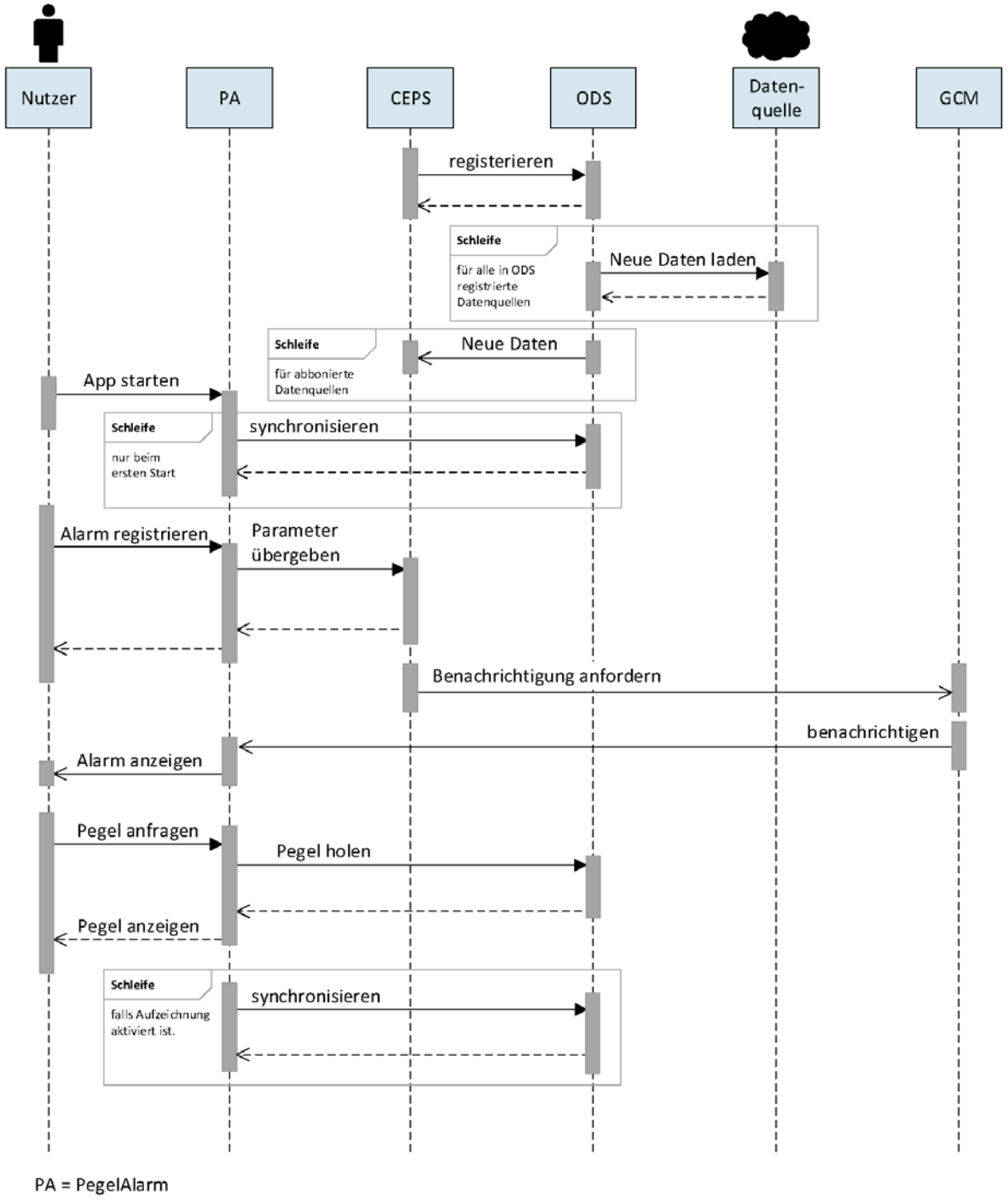

Abbildung 4 Laufzeitsicht der häufigsten Anwendungsfälle. (Quelle: Eigene Darstellung)

App-Benutzer erkundet aktuelle Wasserstände und App-Benutzer registriert einen Alarm. Eine Abfrage eines Wasserstandes wird direkt an ODS adressiert. Alarm-Registrierungsdaten, die Ereignis-Bedingungen

beschreiben, gehen an CEPS, der bei ODS als Subscriber registriert ist und automatisch neue Daten bekommt, sobald ODS frische Daten von den Datenquellen hat. Diese Daten werden an eine Esper Engine[12] weiterleitet, die für die Verarbeitung komplexer Ereignisse verantwortlich ist. Wenn die Ereignis-Bedingungen erfüllt sind, und ein Ereignis eingetreten ist, initiiert CEPS eine Benachrichtigung an den App-Benutzer. Die Daten, die diese Benachrichtigung verursacht haben, werden dabei gespeichert, jedoch werden sie nicht von der App PegelAlarm abgeholt und werden im Laufe eines expliziten CEPS Garbage Collection periodisch gelöscht.

Falls die Aufzeichnungs-Funktionalität der App aktiviert ist (der Benutzer kann die Wassertrends über die Zeit hinweg beobachten) wird ein periodischer Synchronisierungsvorgang mit dem ODS gestartet: eine Kopie der aktuellen Daten von ODS der Datenquelle *PEGELONLINE* wird angefragt und auf dem Gerät lokal gespeichert.

3.6 Übergreifende Architekturaspekte und technische Konzepte

Die folgenden Abschnitte stellen die querschnittlichen Aspekte des SUT dar, die im Rahmen dieser Arbeit relevant sind.

3.6.1 Persistenz

ODS speichert seine Daten in einer zentralen Datenbank. Die Datenbank enthält nur die aktuellsten Daten der registrierten Datenquellen, alte Werte werden regelmäßig überschrieben. Zusätzlich zu den eigentlichen Nutzdaten werden zu jeder Quelle auch Metadaten gespeichert (z. B. Daten-Schema, das das Format der einzulesenden Daten beschreibt). In ODS wird die dokumentenorientierte Datenbank CouchDB[13] eingesetzt, die im Kontext von vielen verschieden Datenquellen einen Vorteil gegenüber relationalen Datenbanken aufweist – es wird kein Schema benötigt, so dass es möglich ist, beliebige Rohdaten in der Datenbank abzulegen. Der interne Zugriff auf die Daten erfolgt bei CouchDB ebenfalls über eine REST-HTTP-Schnittstelle. Die aktuelle Implementierung ermöglicht ein gleichzeitiges Einfügen oder Aktualisieren von mehreren Dokumenten in die Datenbank mithilfe der *executeBulk*-Methode. Dies stellt die bevorzugte Methode zum Einfügen von Dokumenten in CouchDB dar (Reischl, 2014).

CEPS verwendet nach den gleichen Prinzipien auch eine dokumentenorientierte CouchDB.

3.6.2 Esper Engine

Die Esper-Engine, die von CEPS benutzt wird, übernimmt die eigentliche Analyse der Daten, um voneinander abhängige Ereignisse (engl.: events) zu erkennen. Esper hält alle Datenstrukturen im Speicher und benötigt keine Datenbank oder Laufwerk. Anstatt der Speicherung der Daten und dem Ausführen von Abfragen auf diesen gespeicherten Daten bietet Esper die Abfragen in Form von Filterketten an und lässt einkommende Daten durch diese zur Analyse laufen. Die Ereignisse werden in Echtzeit ausgelöst, sobald mit den gespeicherten Abfragen übereinstimmende Bedingungen auftreten.

3.6.3 CEPS Explizite Garbage Collection

Sowohl die Daten, die einen Alarm ausgelöst haben (sogenannte Ereignisdaten), als auch registrierten

[12] http://esper.codehaus.org
[13] http://couchdb.apache.org/

Clients werden von CEPS in der Datenbank gespeichert. In beiden Fällen gibt es jedoch keine Garantie, dass diese Daten jemals entsorgt werden. Dies übernimmt die explizite Garbage Collection (GC) von CEPS: eine Garbage Collection von Ereignisdaten läuft in regelmäßigen Abständen und verwirft alle Ereignisse, die älter als ein definierter Schwellwert sind.

So entsorg die App *PegelAlarm* zum Beispiel nie die Daten, die einen Alarm ausgelöst haben, die CEPS jedoch explizit speichert: eine Benachrichtigung von GCM mit einsprechender Ereignis-ID ist für das Verständnis, was für ein Alarm getriggert wurde, ausreichend. Die Ereignisdaten werden nicht explizit abgeholt und der expliziten Garbage Collection überlassen. Wie oft der *GarbageCollector* zum Einsatz kommt hängt von der vorgenommenen CEPS-Konfiguration ab. Aktuell hat sich dieser Wert bei 60 Min. bewährt.

Eine Garbage Collection von registrierten Benutzern wird über eine Heartbeat-Nachricht an das Zielgerät mithilfe des GCM-Frameworks implementiert und ist im Rahmen dieser Arbeit nicht von Interesse, da dies außerhalb des SUT sowie des betrachteten Zeitraums geschieht.

3.6.4 Logging

ODS und CEPS bieten ein Logging von Aktivitäten und geben zur Laufzeit Auskunft über den Status des Programms auf 3 unterschiedliche Stufen:

- INFO – zum Speichern von Zugriffen.
- DEBUG – zum detaillierten Speichern von Aktionen, Zugriffen und Fehlervorgängen.
- ERROR – zum Speichern von Fehlervorgängen.

3.6.5 Konfigurierbarkeit

Die Konfigurationsmöglichkeiten von ODS und CEPS geben ihnen eine gewisse Flexibilität: Die Anzahl an ODS-Datenquellen sowie die gewünschte Komplexität der Datenverarbeitung (Zusammensetzung der Filter-Kette) bilden die wichtigsten Konfigurationsparameter des ODS:

- Datenquellen mit entsprechenden Filterketten und Aktualisierungsintervallen,
- Daten-Views der CouchDB, die abgefragt werden können,
- Subscriber, die neue Daten von ODS automatisch bekommen möchten,
- Maximale Anzahl an *Connections* im Connection-Pool der Datenbank.

Folgend sind die wichtigsten Konfigurationsparameter des CEPS beschrieben:

- Dem ODS entsprechende Datenquellen,
- EPL-Adapter, der die von Benutzer übergebenen Parameter in eine gültige EPL-Anweisung umwandelt,
- Maximale Anzahl an *Connections* im Connection-Pool der Datenbank.

Ausgehend von verschiedenen Konfigurationen von ODS und CEPS ist ein entsprechend unterschiedliches Zeit- und Ressourcennutzungsverhalten dieser Services zu erwarten. In dieser Arbeit wird das SUT entsprechend den Anforderungen der App PegelAlarm konfiguriert (siehe Abschnitt 4.5.7).

3.7 Entwurfsentscheidungen und Trade-Offs

Bei der Umsetzung von ODS und CEPS mussten Designentscheidungen getroffen werden, die einen Trade-Off zwischen verschiedenen Architekturzielen (vgl. Abschnitt 3.3) verlangten. Hierzu wurden Kompromisse eingegangen, um die bestmögliche Ausprägung der Ziele zu erreichen. Diese Kompromisse sind im Folgenden aufgeführt:

- Benutzerfreundlichkeit und Netz-Performance vs. Lizenzeinschänkung: obwohl CEPS enge Kommunikation mit ODS betreibt und von ihm mit Daten versorgt wird, ist er als ein separater Service entworfen und gebaut. Auf diese Weise wird ODS von der direkten Kommunikation mit der Esper-Engine abstrahiert, um mögliche Lizenz-Konflikten zu vermeiden. Dies bringt jedoch zusätzliche Komplexität in das System und zwingt Anwender, mit zwei Services, statt mit einem, über das Netz zu kommunizieren.
- Informationssicherheit vs. Benutzerfreundlichkeit und Performance: CEPS führt eine zusätzliche Abstraktionsschicht zwischen benutzerdefinierten CEPS Regeln und Esper event processing language (EPL)-Anweisung, die an die Esper-Engine übergeben werden: EPL-Anweisungen können von böswilligen Benutzer ausgenutzt werden, um die Daten von anderen Benutzer zu beeinflussen. Dies erfordert jedoch die Konfiguration eines speziellen Adapters (EPL-Adapter) für jede Datenquelle und gewünschtes Ereignis-Muster. Dieser Adapter wandelt die Benutzer-Parameter in eine gültige EPL-Anweisung um.
- Batterielebensdauer vs. Aktualität der Daten: PegelAlarm hält lokale Daten synchron mit den ODS-Daten unter der Prämisse, die Gerätebatterie so gut wie möglich zu schonen, was sich in teilweise längeren Synchronisierungsintervallen auswirkt.

4 Forschungsansatz

Dieses Kapitel fasst das ausgewählte Testvorgehen zusammen und bietet eine Übersicht über den durchgeführten Testprozess – von der Definition der Testzielen und Akzeptanzkriterien bis zu der Testdurchführung. Hierbei werden die Lastmengen definiert, Testdaten vorbereitet und Testskripte implementiert.

4.1 Testziele

In Abschnitt 2.3 wurde bereits beschrieben, dass unter dem Sammelbegriff L&P-Test eine Reihe von Testausprägungen zusammengefasst werden, die unterschiedliche Ziele verfolgen. Aus diesem Grund ist eine Klarstellung der Testziele vor jeder Testdurchführung, im Rahmen der Testvorbereitung, notwendig. In (Barber, 2004b) werden drei Aspekte eines in der Entwicklung befindlichen Systems für die Untersuchung in den L&P-Tests besonders hervorgehoben:

- Ermittlung der tatsächlichen Performance-Eigenschaften des Systems;
- Validierung der Designentscheidungen, die in Bezug auf die Performance-Eigenschaften von besonderer Relevanz sind;
- Planung der Behandlung zunehmender Last durch die Erhöhung der Kapazität.

Angelehnt an diese Aspekte werden folgende Testziele für die in dieser Arbeit durchgeführten L&P-Tests definiert:

1. **Ermittlung des tatsächlichen Performance-Verhaltens von ODS und CEPS im Einsatzfall der App PegelAlarm unter den erwarteten Niedrig-, Normal- und Hochlastbedingungen:**
   - Ermittlung der maximalen Anzahl an parallelen Benutzern, die die Services unter definierten Bedingungen und Laufzeiten unterstützen können.
   - Ermittlung der Bottlenecks des Systems.
   - Ermittlung der Performance-Bugs, die erst unter Last oder längeren Betriebszeiten auftreten.
2. **Bewertung der getroffenen Designentscheidungen bei der Entwicklung von ODS und CEPS am Beispiel des Einsatzfalls der App PegelAlarm auf Basis des beobachteten Verhaltens:**
   - Bewertung der Auswirkungen der Entscheidung, CEPS in einen externen Service zu separieren, auf Ressourcenverbrauch und Laufzeitverhalten des Gesamtsystems.
   - Bewertung der Performance-Eigenschaften der dokumentenorientierten Datenbank, die als eine schemalose Alternative zu relationalen Datenbanken eingesetzt wurde. Dabei insbesondere eine Bewertung, inwieweit der Einsatz von dokumentenorientierten Datenbanken den Qualitätsanforderungen entspricht.
3. **Skalierbarkeitsanalyse:**
   - Planung einer erhöhten Kapazität, um zunehmende Last zu unterstützen: Bewertung, wie gut ODS und CEPS unter verschiedenen Lastszenarien die Herausforderungen – mehr Benutzer und mehr Datenquellen – bewältigen kann.

Ausgehend von den bekannten Implementierungsdetails des Systems ist es zudem interessant, auch weitere

Kennzahlen zu erheben:

- Auswirkungen der expliziten Garbage Collection von CEPS (wie im Abschnitt 3.6.3 beschrieben) auf den Ressourcen-Verbrauch (vor allem die CPU-Auslastung) und das Laufzeit- und Antwortzeitverhalten.

4.2 Nicht-funktionale Anforderungen

Die Qualität der Software-Architektur zeigt sich primär in der Erfüllung von nicht-funktionalen Anforderungen (Hruschka & Starke, 2012). Der Term „nicht-funktionale Anforderungen“ (NFA) selber kann jedoch unterschiedlich definiert werden (mehr dazu in (Chung & do Prado Leite, 2009; Glinz, 2007)). In dieser Arbeit werden darunter solche Anforderungen verstanden, die sich nicht auf die Funktionalität des Systems, sondern auf Merkmale wie Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit beziehen.

Für die Durchführung des L&P-Tests und die Auswertung der Ergebnisse müssen vorab Akzeptanzkriterien definiert werden, mit denen die Testergebnisse verglichen werden. Diese Akzeptanzkriterien werden oft von NFAs abgeleitet und prüfen damit deren Einhaltung.

Wie im Abschnitt 3.2 bereits angemerkt sind die für ODS formulierten Performance-bezogenen NFA nicht prüfbar und damit nicht testbar, sondern müssen präzisiert werden. Für CEPS wurden keine Performance-relevanten NFA dokumentiert, so dass diese ebenfalls im Rahmen der Testvorbereitung definiert werden müssen.

Bei der Definition der NFAs an ODS und CEPS müssen die Randbedingungen ihres aktuellen Einsatzes berücksichtigt werden: als Daten-Lieferanten für die App PegelAlarm müssen die Anforderungen an diese Services vor allem ein reibungsloses Funktionieren der App aus Anwendersicht garantieren. Die von ODS und CEPS angebotene Funktionalität ist jedoch viel generischer und ermöglicht ein breites Einsatzfeld. Daher dürfen die in dieser Arbeit definierten NFAs nicht absolut gesehen werden: sie sind dem Anwendungsszenario bedingt und müssen in einem anderen Einsatzfall neu definiert werden. Für den in dieser Arbeit beschriebenen L&P-Test hingegen sind die im Folgenden definierten NFAs passend.

In (Mannion & Keepence, 1995) wird das Modell der SMART-Anforderungen vorgestellt, das in dieser Arbeit für die Formulierung von NFAs herangezogen wird. Anforderungen sollten danach spezifisch (klar und konsistent), messbar (auf Basis von quantitativen Größen), anspruchsvoll (um nützlich zu sein), realistisch (um erreichbar zu sein) und nachverfolgbar (nachvollziehbar) sein.

Um eine hohe Akzeptanz bei den Anwendern zu erzielen muss das Zeitverhalten der App von Anfang an mit den Erwartungen der Anwender übereinstimmen. Es ist ein allgemeiner Konsensus in der Literatur, dass lange Wartezeiten sich schädlich auf die Zufriedenheit der Anwender auswirken (Martin und Corl, 1986). Die Bestimmung des Grenzwertes, ab wann die Wartezeiten einem Benutzer als zu lang vorkommen, ist keine triviale Sache und hängt stark von dem Anwendungsfall und dem Fluss der Mensch-Maschine-Interaktion ab. Die Ergebnisse der Untersuchungen von (Card, Robertson, & Mackinlay, 1991) fassen die folgenden Grenzen der Zeitwahrnehmung zusammen:

- Eine Verzögerung von weniger als 0,1 Sekunde wird vom Benutzer nicht gemerkt.
- Eine Verzögerung von weniger als 1 Sekunde wird vom Benutzer wahrgenommen, unterbricht jedoch nicht seinen Gedankenfluss.
- Eine Verzögerung bis zu 10 Sekunden wird von Benutzer noch geduldet, darf aber kein Dauerzustand sein.
- Nach 10 Sekunden verlieren Benutzer den Fokus und fangen im Allgemeinen an, etwas anderes zu tun.

Von (Martin & Corl, 1986) wurden die Antwortzeiten zwischen 1 und 10 Sekunden detaillierter untersucht: Die Autoren zeigen, dass eine 2-Sekunden-Antwortzeit einen kritischen Grenzwert für die Produktivität der durchschnittlichen Anwender darstellt. Antwortzeiten über 2 Sekunden mindern die Konzentration und behindern die Erfüllung komplexer Aktivitäten. Angelehnt an diese Studien wurden die erträglichen Antwortzeiten für einzelne Anwendungsfälle des SUT festgelegt (Tabelle 2). Diese Antwortzeiten werden als Akzeptanzkriterien für die L&P-Tests gesehen und bilden die NFA für das SUT.

Tabelle 2 ODS- und CEPS- relevante NFA für PegelAlarm.

Anwendungs-fall	Beschreibung	Maximale Antwortzeit
PA-AF 1	PegelAlarm erster Startvorgang (Synchronisierung mit ODS)	10 s.
PA-AF 2a	Wasserstand abfragen	2 s.
PA-AF 2b	Wasserstände entlang eines Flusses abfragen	4 s.
PA-AF 3	Alarm registrieren	2 s.
PA-AF 4	Wasserstände aufzeichnen (Synchronisierung mit ODS)	1 Min. (da im Hintergrund)

Die Zeitangaben sind aus der Endbenutzer-Sicht zu sehen und beinhalten durchschnittlich angenommene Netzlatenzen über eine mobile oder stationäre Internetverbindung: Die Benutzer der App PegelAlarm werden sich größtenteils außerhalb des Universitätsnetzes aufhalten, in dem der Server gehostet wird, und über ein WLAN-Netz oder ein hochwertiges mobiles Netz mit dem Internet verbunden sein.

4.3 Modellbasiertes Testvorgehen

Der klassische Testansatz impliziert eine manuelle Erstellung einzelner Testfälle. Dies ist nicht nur zeitintensiv und fehleranfällig, sondern benötigt, um effektiv zu sein, eine exakte Feststellung, was genau getestet werden soll (das Testauswahlkriterium) und wie viel (das Testendekriterium) (Prowell, 2005). Eine mögliche Antwort auf diese Herausforderung stellt das modellbasierte Testen (MBT) dar: Ein Testverfahren, das auf einem Modell der zu testenden Komponente oder des zu testenden Systems basiert, und Testfälle automatisch daraus ableitet.

Ein Modell ist eine Abstraktion und stellt nur bestimmte Eigenschaften des Systems dar, entsprechend seinem Einsatzzweck. In Rahmen des MBT spezifiziert so ein Modell vor allem die Nutzung der Software (Black-Box-Sicht): Das Systemverhalten wird mit allen möglichen Eingaben und den zu erwarteten Ausgaben dargestellt (z. B. in Form eines endlichen Automaten, UML-Diagramms, etc.). Da so ein Modell nicht auf das interne Verhalten der Software eingeht, kann es schon früh im Software-Entwicklungszyklus

erstellt werden. Testfälle werden aus dem Modell automatisch generiert und prüfen die Gesamtfunktionalität des Systems gegen die Spezifikation. Zudem werden die Modelle auch für die Analyse der erwarteten Nutzung im Betrieb verwendet sowie zur Bestimmung des Testendekriteriums.

Eine Voraussetzung für die Gültigkeit eines L&P-Tests nach (Menascé, 2002) ist, dass das Verhalten der virtuellen Benutzer ähnlich dem Verhalten der tatsächlichen Benutzer ist. Eine Menge starrer Testfälle wird jedoch nie die Flexibilität und Varianz des tatsächlichen Nutzerverhaltens nachbilden können. Aus diesem Grund wird der Einsatz von probabilistischen Modellen für die Ableitung von L&P-Testfällen bevorzugt. So ein modellbasiertes Testentwurfsverfahren kann auf Basis von Markov-Ketten erfolgen, die probabilistisch beschreiben, wie, von wem und über welche Schnittstellen das Testobjekt benutzt wird. Diskrete Markov-Ketten sind Zustandsautomaten mit diskreten Übergangswahrscheinlichkeiten an den Transitionen. Sie beschreiben die Wahrscheinlichkeit, dass eine bestimmte Transition ausgeführt wird, jedoch nicht den Zeitpunkt (Liggesmeyer, 2002).

Eine zentrale Aufgabenstellung des modellbasierten Testens ist die Erstellung eines solchen formalen Modells des Systems – des Testmodells. (Van Hoorn, Rohr, & Hasselbring, 2008) stellen einen Ansatz für die Erzeugung von Testmodellen auf Basis von Nutzerverhaltensmodellen, die für die automatisierte Ableitung von Testfällen und somit die Generierung des probabilistischen Workloads im Rahmen eines L&P-Tests benutzt werden, vor. Zwei mathematische Modelle werden dabei erstellt: ein Anwendungsmodell und ein Nutzerverhaltensmodell.

Ein Anwendungsmodell (engl.: application model) ist ein Modell, das die zulässigen Sequenzen von Nutzerinteraktionen mit dem betrachteten System beschreibt, z. B. in Form einer Zustandsmaschine (engl. state machine). Eine der größten Herausforderungen bei der Modellierung eines komplexen Systems ist, das Modell entsprechend seines Einsatzzwecks mit allen relevanten Informationen zu versehen und trotzdem handhabbar zu behalten. Im Rahmen der L&P-Tests kann das Anwendungsmodell entsprechend auf performance-relevante Aspekte eingeschränkt werden. In (Meier et al., 2007) wird eine Liste an Einsatzszenarien aufgeführt, die in einem L&P-Test abgedeckt werden müssen, je nach Randbedingungen:

- Häufigste Nutzungsszenarien – die Einsatzszenarien, die bei den Anwendern besonders beliebt sind.
- Geschäftskritische Nutzungsszenarien – die Einsatzszenarien, die für die Erfüllung der Business-zwecke des Systems besonders relevant sind.
- Performance-intensive Nutzungsszenarien – die Einsatzszenarien, die besonders viele Ressourcen in Anspruch nehmen.
- Nutzungsszenarien von großer Bedeutung für Stakeholder.
- Vertraglich verpflichtende Nutzungsszenarien.

Ein Nutzerverhaltensmodell (engl.: user behavior model) hingegen ist ein Modell, dass das Verhalten von Benutzern mit der Anwendung beschreibt. Es wird auf Basis des Anwendungsmodells erstellt, indem die Wahrscheinlichkeiten, mit denen bestimmte Aktionen in der Anwendung durch den Benutzer ausgeführt werden, annotiert werden (als Markov-Kette spezifiziert). Ein gängiger Ansatz für die Anreicherung der Anwendungsmodelle mit Wahrscheinlichkeiten und Think Times basiert auf der Auswertung von

empirischen Daten: z. B. von Log-Dateien. In (Savoia, 2001) wird ein Workload durch Analysieren von Protokolldateien von laufenden Versionen der Anwendung generiert.

Für ein neuentwickeltes System, das noch nicht im Produktiveinsatz war und demgemäß keine Produktivdaten hat, ist diese Methode nicht anwendbar. (Barber, 2004b) schlägt ein Vorgehen für die Erstellung von validen Lastmodelle für L&P-Test mit unvollständigen empirischen Daten vor, das auf der Bewertung der Anwendungsfälle und anderen Design-Dokumenten sowie Informationen von Stakeholder basiert. Nach Verfügbarkeit werden auch Informationen von der Benutzung ähnlicher Systeme (z.B. Vorgängersysteme) und die Ergebnisse von Akzeptanz- oder Usability-Tests herangezogen. Es werden pro Benutzertyp zwei Modelle erstellt: das erwartete Nutzerverhaltensmodell und ein Worst-Case Nutzerverhaltensmodell, die auf Basis von L&P-Testergebnissen verbessert werden können.

Zusätzlich wird das Testmodell mit zwei ausgezeichneten Zuständen versehen: einem Startzustand, in dem sich das System zu Beginn der Ausführung eines Testfalls befindet, und einem Endzustand, der das Ende eines Testfalls markiert. Ein Testfall ist ein zufälliger Pfad durch das Modell vom Startzustand zum Endzustand (Zimmermann, Eschbach, Kloos, & Bauer, 2009). Daher ist gefordert, dass jeder Zustand der Markov-Kette vom Startzustand erreichbar ist, und dass von jedem Zustand der Ausgangszustand erreicht werden kann.

Werkzeuggestützt kann aus dem Testmodell nun automatisch eine (theoretisch) unbegrenzte Anzahl von generierten Testfällen abgeleitet werden. Diese Testfälle sind zufällig, aber realistisch (sie entsprechen der erwarteten Nutzung) und umfassen (theoretisch, in der Praxis ist ein L&P-Test immer zeitlich begrenzt) alle zulässigen Navigationspfade. Die potentiell hohe Anzahl an Duplikaten unter ihnen muss jedoch berücksichtigt werden.

Der beschriebene Prozess ist in der Abbildung 5 dargestellt.

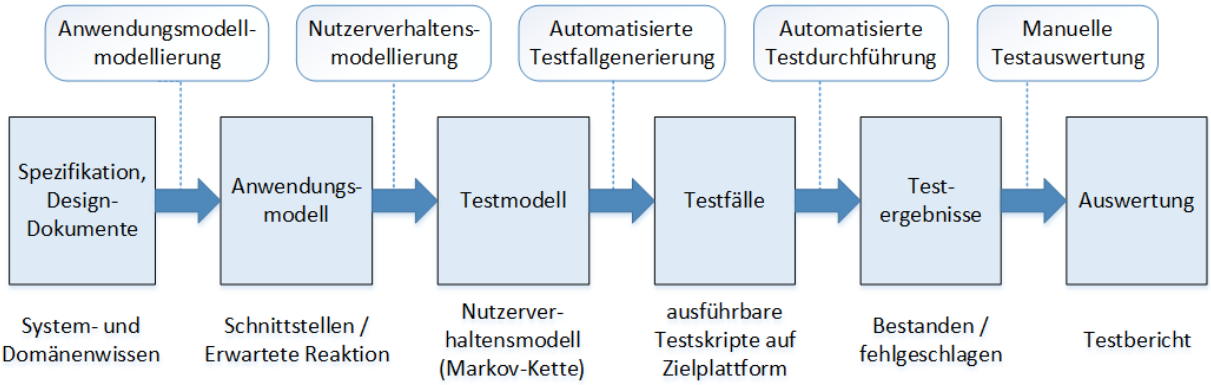

Abbildung 5 Hauptschritte des MBT: von der Modellerstellung zu Testauswertung. (Quelle: Eigene Darstellung, angelehnt an (Zimmermann et al., 2009)).

4.3.1 Testmodell

Für die Modellierung des Anwendungsmodells (wie im Abschnitt 4.3 vorgestellt) wurde eine Analyse der relevanten Anwendungsfälle durchgeführt, um das Anwendungsmodell auf die für den L&P-Test entscheidenden Zustände zu beschränken.

Aufgrund des Einsatzes der beiden Services in der App PegelAlarm stehen die Anwendungsfälle der App

im Vordergrund. Aus der ganzen Vielfältigkeit der Anwendungsfälle der App sind jedoch nur solche relevant, die Aktivitäten auf den beiden Services – ODS und CEPS – anstoßen. Folgend werden die einzelnen Anwendungsfälle identifiziert und nach (Meier et al., 2007) in drei Gruppen – *häufige*, *geschäftskritische*, *performance-intensive* Einsatzszenarien – klassifiziert (PegelAlarm Tabelle 3, CEPS Tabelle 4, ODS Tabelle 5), mit dem Ziel, die zu modellierende Anwendungsfälle zu identifizieren. Die Kriterien der Klassifizierung der Anwendungsfälle als *häufige* und / oder *geschäftskritische* basiert auf den ersten Erfahrungen mit der App im Rahmen des Beta-Tests und den erwarteten Nutzungsszenarien. Das Kriterium der Klassifizierung der Anwendungsfälle als *performance-intensiv* basiert auf der Erwartung aus der Black-Box-Sicht, insbesondere bei der Übertragung oder Verarbeitung großer Datenmengen bei der Durchführung dieser Anwendungsfälle.

Tabelle 3 Klassifikation der wichtigsten Anwendungsfälle von PegelAlarm für die Erstellung des Testmodells.

Anwendungs-fall	Beschreibung	ODS/CEPS relevant	Häufige ES	Geschäfts-kritische ES	Performance-intensive ES
PA-AF 1	PegelAlarm erster Startvorgang (Synchro-nisierung mit ODS)	X	X	X	X
PA-AF 2	Wasserstand abfragen	X	X	X	
PA-AF 3	Alarm registrieren	X	X	X	
PA-AF 4	Wasserstände auf-zeichnen (Synchroni-sierung mit ODS)	X			X
PA-AF 5	Aufzeichnung beenden	X			

ES = Einsatzszenario.
X = trifft zu.
(X) = trifft teilweise zu.

Tabelle 4 Klassifikation der wichtigsten Anwendungsfälle von CEPS für die Erstellung des Testmodells.

Anwendungs-fall	Beschreibung	Häufige ES	Geschäfts-kritische ES	Performance-intensive ES
CEPS-AF 1	Datenquelle registrieren		X	
CEPS-AF 2	EPL-Adapter registrieren		X	
CEPS-AF 3	EPL-Adapter löschen			
CEPS-AF 4	Client-Regel registrieren	X	X	(X)
CEPS-AF 5	Client-Regel löschen	X		
CEPS-AF 6	Benachrichtigung verschicken	X	X	(X)
CEPS-AF 7	Aggregierte Daten, die die Client-Regel ausgelöst haben, abfragen.	(X)	(X)	
CEPS-AF 8	Regel löschen	X	X	

ES = Einsatzszenario.
X = trifft zu.
(X) = trifft teilweise zu.

Tabelle 5 Klassifikation der wichtigsten Anwendungsfälle von ODS für die Erstellung des Testmodells.

Anwendungs-fall	Beschreibung	Häufige ES	Geschäfts-kritische ES	Performance-intensive ES
ODS-AF 1	Datenquelle registrieren		X	
ODS-AF 2	Filter-Kette für Datenquelle anlegen		X	
ODS-AF 3	View für Datenquelle anlegen		X	
ODS-AF 4	View abfragen	(X)		(X)
ODS-AF 5	Daten einer Datenquelle abfragen	X	X	(X)
ODS-AF 6	Bei der Benachrichtigungsfunktion registrieren, um die neuen Daten von ODS unmittelbar zu bekommen	(X)		(X)
ODS-AF 7	Daten an Subscriber weiterleiten	(X)		X

ES = Einsatzszenario.
X = trifft zu.
(X) = trifft teilweise zu.

Basieren auf dieser Klassifikation wird die Modellierung des Systems als Zustandsautomat vorgenommen, mit zwei Pseudo-Zuständen: einem Start- und einem End-Zustand (Abbildung 6). Hinter jedem Zustandswechsel stehen entsprechende HTTP-Anfragen an ODS und CEPS. Manche Transitionen sind mit Verhaltensspezifikationen und Wächterausdrücken (engl. guards) versehen: wird der Wächterausdruck logisch wahr ausgewertet, kann die Transition durchlaufen und ihr zugeordnetes Verhalten ausgeführt werden. Das Verlassen des Systems ist zu jeder Zeit aus jedem Zustand möglich.

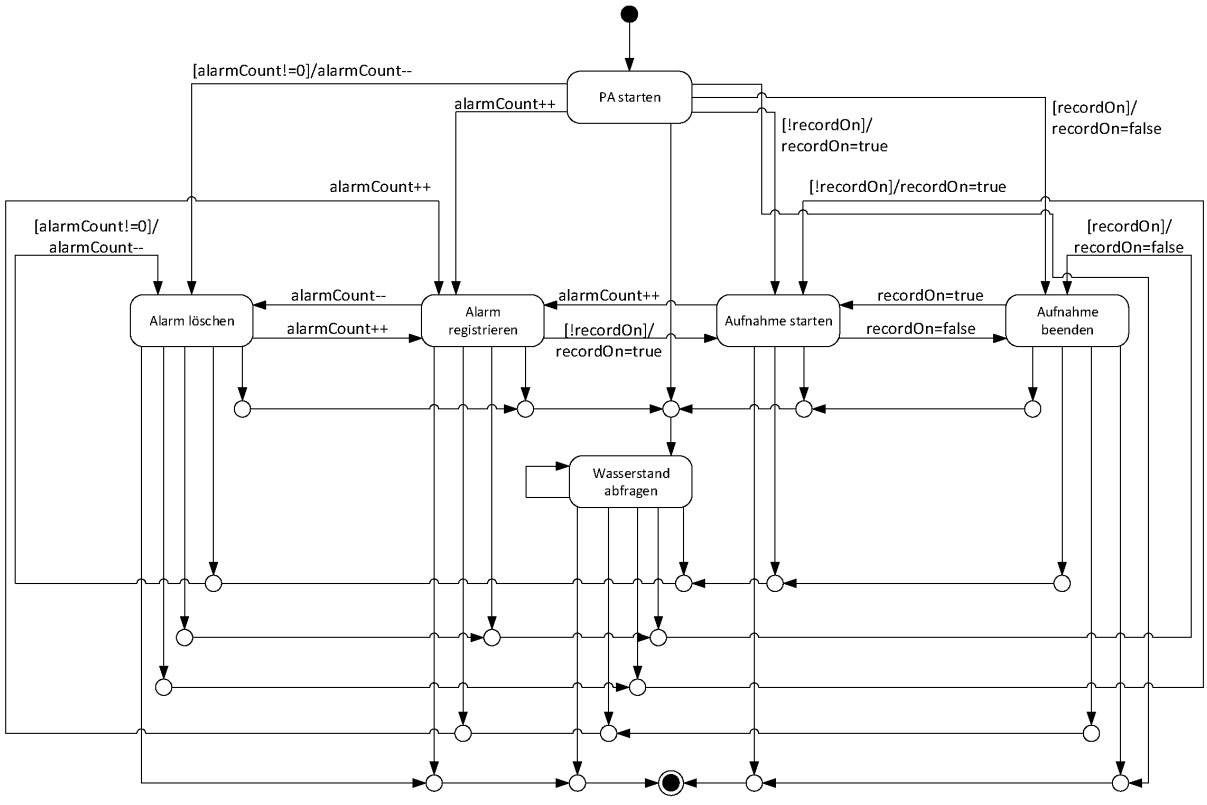

Abbildung 6 Anwendungsmodell des Systems. Nach dem Start der Anwendung ist ein Übergang von jedem Zustand in jeden Zustand möglich (unter Berücksichtigung der Wächterausdrücke). (Quelle: Eigene Darstellung).

Wird dieses Modell mit Übergangswahrscheinlichkeiten von jedem Zustand in einen anderen angereichert, entsteht ein Benutzerverhaltensmodell. Das Benutzerverhaltensmodell beschreibt, welche Aktion mit welcher Wahrscheinlichkeit vom simulierten Benutzer als nächstes durchgeführt wird. Dieser Schritt wird in Abschnitt 4.4.1 vorgestellt.

4.4 Test Workload

Beim Einsatz von ODS und CEPS im Rahmen der App PegelAlarm muss berücksichtigt werden, dass sich die auf das System gerichtete Last über die Zeit nicht gleichmäßig verteilen wird: in den Hochwassersaisons ist eine deutlich höhere Last auf dem System zu erwarten, als in den Zeiten, in denen keine Hochwassergefahr besteht. Außerdem spielen die Popularität und die Verbreitung der App eine entscheidende Rolle bei der Abschätzung der erwarteten Lastmengen. Ausgehend von diesen Randbedingungen wurden drei Arten an Lastsituationen definiert: *Niedriglast*, *Normallast* und *Hochlast*. Das System wird entsprechen dieser Testszenarien unter eine entsprechende Menge und Art der Last gesetzt und gemessen:

- Unter *Niedriglast* wird eine Situation verstanden, in der die App eine erste Verbreitung in der Bevölkerung erzielen konnte und keine Hochwassergefahr besteht.
- Unter *Normallast* wird eine Situation verstanden, in der die App eine erste Verbreitung in der Bevölkerung erzielen konnte und eine akute Hochwassergefahr besteht.
- Unter *Hochlast* wird eine Situation verstanden, in der die App eine hohe Popularität besitzt und akute Hochwassergefahr besteht.

Für jedes dieser Testszenarien muss ein entsprechender Workload erstellt werden, der die erwartete Lastmenge definiert. Nach dem Ansatz von (Van Hoorn et al., 2008) wird pro Testszenario ein probabilistischer Workload auf Basis von Benutzerverhaltensmodellen erzeugt. Dies erfordert eine Spezifizierung der folgenden Parameter:

- Definition der Anwendertypen und Erstellung eines ihrem Verhalten entsprechenden Benutzerverhaltensmodells.
- Definition des Benutzerverhaltensmixes, der die Wahrscheinlichkeiten für das Auftreten der einzelnen Benutzerverhaltensmodelle während des L&P-Tests angibt.
- Definition der Workload-Intensität, die die während des L&P-Tests simulierte Benutzerzahl und ihre Änderung über die Testzeit hinweg spezifiziert.

4.4.1 Benutzertypen

Alle PegelAlarm-Anwender werden auf zwei Nutzerverhaltensmodelle abgebildet, die auf Basis einer Abschätzung der App-Bedienung entstanden sind:

- *Neuer Benutzer*
- *Wiederkehrender Benutzer*

Diese Profile repräsentieren zwei Arten der Anwendungsnutzung: Das Profil *Neuer Benutzer* repräsentiert die App-Anwender, die zum ersten Mal die Anwendung verwenden. Sie erkundigen die App-Möglichkeiten

und probieren die Funktionalitäten aus. Ihre Sitzung wird dadurch länger ausfallen (im Vergleich zu den *Wiederkehrenden Benutzern*). *Wiederkehrende Benutzer* sind mit den Funktionalitäten der App vertraut und nehmen sie bewusst in Anspruch. Sie führen weniger Aktionen durch und verlassen die Anwendung relativ schnell. Die Profile werden durch Übergangsmatrizen dargestellt (Abbildung 7, Abbildung 8). Vom initialen Zustand *App starten* aus verlässt ein *Neuer Benutzer* das System im Durchschnitt nach 14 Schritten mit einer Wahrscheinlichkeit von ca. 94%. Ein *Wiederkehrender Benutzer* verlässt das System ausgehend vom initialen Zustand *Wasserstand abfragen* im Durchschnitt nach 5 Schritten mit ca. 94% Wahrscheinlichkeit.

(a)

$$\begin{pmatrix} 0 & 0.3 & 0.6 & 0.05 & 0 & 0 & 0.05 \\ 0 & 0.2 & 0.3 & 0.2 & 0.1 & 0.1 & 0.1 \\ 0 & 0.2 & 0.4 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0 & 0.2 & 0.4 & 0 & 0.1 & 0.2 & 0.1 \\ 0 & 0.1 & 0.2 & 0.1 & 0.1 & 0.1 & 0.4 \\ 0 & 0.1 & 0.2 & 0.1 & 0.1 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(b)

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.5 \\ 0 & 0.1 & 0.2 & 0.1 & 0.1 & 0.1 & 0.4 \\ 0 & 0.1 & 0.3 & 0 & 0.15 & 0.05 & 0.4 \\ 0 & 0.1 & 0.15 & 0.1 & 0.15 & 0.1 & 0.4 \\ 0 & 0.15 & 0.1 & 0.05 & 0.1 & 0.1 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Abbildung 7 Übergangsmatrizen der Nutzerverhaltensmodelle, entsprechend den Markov-Zuständen in der Reihenfolge *App erstmalig starten, Alarm registrieren, Wasserstand abfragen, Aufnahme starten, Alarm löschen, Aufnahme beenden, App verlassen*. Normale Bedingungen: (a) *Neuer Benutzer*, (b) *Wiederkehrender Benutzer* (Der Zustand *App erstmalig* starten ist für dieses Profil irrelevant. Startzustand ist *Wasserstand abfragen*). (Quelle: Eigene Darstellung).

Es ist jedoch zu erwarten, dass beide Benutzerklassen sich leicht unterschiedlich unter Normalbedingungen und im Falle eines Hochwassers verhalten. Während in den normalen Zeiten die Benutzer die App eher aus reinem Interesse benutzen, sind die aktuellen Pegelstand-Informationen in den Hochwasserzeiten überlebenswichtig: die Benutzer werden viel mehr Abfragen zum aktuellen Wasserstand abschicken. Daraus ergeben sich zwei weitere Profilvariationen (in der Abbildung 8 als Übergangsmatrizen dargestellt).

(a)

$$\begin{pmatrix} 0 & 0.3 & 0.6 & 0.07 & 0 & 0 & 0.03 \\ 0 & 0.2 & 0.3 & 0.2 & 0.1 & 0.1 & 0.1 \\ 0 & 0.2 & 0.4 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0 & 0.2 & 0.4 & 0 & 0.1 & 0.2 & 0.1 \\ 0 & 0.1 & 0.3 & 0.1 & 0.1 & 0.1 & 0.3 \\ 0 & 0.1 & 0.2 & 0.1 & 0.1 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(b)

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.5 \\ 0 & 0.1 & 0.3 & 0.1 & 0.1 & 0.1 & 0.2 \\ 0 & 0.1 & 0.2 & 0 & 0.15 & 0.05 & 0.5 \\ 0 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.5 \\ 0 & 0.15 & 0.2 & 0.05 & 0.1 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Abbildung 8 Übergangsmatrizen der Nutzerverhaltensmodelle, entsprechend den Markov-Zuständen in der Reihenfolge *App erstmalig starten, Alarm registrieren, Wasserstand abfragen, Aufnahme starten, Alarm löschen, Aufnahme beenden, App verlassen*. Hochwasser-Bedingungen: (a) *Neuer Benutzer*, (b) *Wiederkehrender Benutzer* (Der Zustand *App erstmalig starten* ist für dieses Profil irrelevant. Startzustand ist *Wasserstand abfragen*). (Quelle: Eigene Darstellung).

Von dem initialen Zustand *Wasserstand abfragen* verlässt ein *Neuer Benutzer* das System im Durchschnitt nach 16 Schritten mit ca. 94% Wahrscheinlichkeit. Ein *Wiederkehrender Benutzer* verlässt das System im Durchschnitt nach 6 Schritten mit ca. 94% Wahrscheinlichkeit.

Die vorgestellten Profile werden je nach Testszenario eingesetzt. Die Festlegung des gewählten Profils

erfolgt bei der Parametrisierung des Testskripts (vgl. Abschnitt 4.5.8).

4.4.2 Nutzerverhaltensmix

Der Nutzerverhaltensmix definiert, welche Benutzertypen mit entsprechenden Nutzerverhaltensmodellen mit welcher Wahrscheinlichkeit bei der Testausführung in der auf das SUT gerichtete Gesamtlast auftreten.

Für die Bestimmung des Nutzerverhaltensmixes werden die vorgestellten Profile mit Gewichten versehen, die ihren Anteil an der Gesamtlast definieren. Für alle drei Testszenarien wird eine unterschiedliche Verteilung der Anwendertypen erwartet (Tabelle 6). Diese Überlegung basiert vor allem auf der Annahme, dass erst in den Hochwasserzeiten eine stärkere Verbreitung der App unter neuen Anwendern zu erwarten ist.

Tabelle 6 Nutzerverhaltensmix für definierte Testszenarien.

Profil	Testszenarien		
	Niedriglast	Normallast	Hochlast
Neuer Benutzer	0,1	0,2	0,3
Wiederkehrender Benutzer	0,9	0,8	0,7

4.4.3 Benutzerzahl PegelAlarm

Die Abschätzung der maximalen Anzahl an App-Benutzer berücksichtigt, dass die von der App angebotenen Daten ausschließlich von Wasserstationen innerhalb Deutschlands stammen. Dies ist der Ausgangspunkt der Erwartung, dass die App vor allem für Benutzer aus Deutschland interessant sein wird. Die Abschätzung der maximalen Anzahl an App-Benutzern basiert auf folgenden statistischen Daten:

- 63% aller Deutschen nutzen ein Smartphone (Stand: März 2015)[14].
- Der Anteil der Android-Smartphones beträgt ca. 68,2% (Stand: Juni 2014)[15].
- Die Anzahl an Android-Smartphones mit einer Betriebssystemversion über der minimal geforderten Android-Version Ice Cream Sandwich (4.0) beträgt ca. 95% (Stand: September 2014)[16].
- In Regionen mit Hochwassergefahr leben in Deutschland ca. 1,66 Millionen Menschen[17].

Die maximale Anzahl an potenziellen Anwendern deutschlandweit beträgt damit ca. 660.000.

Ausgehend von der maximalen Anzahl an potenziellen Anwendern – 660.000 – erfolgt die Abschätzung einer tatsachlich erwarteten Benutzerzahl. Diese ist stark abhängig von der App-Verbreitung und dem Bekanntheitsgrad. Eine Vielzahl an im Google Play Store vertriebenen Apps bietet eine ähnliche Funktionalität wie die App PegelAlarm, jedoch scheint die Alarm-Funktionalität der PegelAlarm-App, kombiniert mit dem effizienten Umgang mit der Geräte-Batterie, einzigartig auf dem Android-Markt zu sein (Eichhorn, 2014). In (Molyneaux, 2014) werden einige Beispiele der Unterschätzung der Popularität

[14] http://www.bitkom-research.de/epages/63742557.sf/de_DE/?ObjectPath=/Shops/63742557/Categories/Presse/Pressearchiv_2015/44_Millionen_Deutsche_nutzen_ein_Smartphone

[15] http://de.statista.com/statistik/daten/studie/170408/umfrage/marktanteile-der-betriebssysteme-fuer-smartphones-in-deutschland/

[16] http://www.mobiflip.de/android-versionsverteilung-deutschland-zeigt-ein-anderes-bild/

[17] http://www.climatecentral.org/news/new-analysis-global-exposure-to-sea-level-rise-flooding-18066

einer neuen Anwendung aus der Praxis aufgeführt und der Rat ausgesprochen, die Popularität einer Anwendung mit neuen Funktionalitäten eher optimistischer zu planen. Da keine weiteren Informationen über den Bekanntheits- und Verbreitungsgrad vorliegen, wird die in dieser Arbeit getroffene Abschätzung der zu erwartenden Anzahl an App-Benutzern über einen Vergleich mit der Popularität einer App mit einer ähnlichen Zielgruppe gemacht, am Beispiel der App *ADAC Pannenhilfe*.

Der ADAC umfasst 2015 ca. 19 Mio. Mitgliedern bundesweit[18]. Die App *ADAC Pannenhilfe* aus dem Google Store zählt zum Zeitpunkt dieser Arbeit ca. 1 Mio. Installationen. Die Anzahl an Installationen zu der Anzahl an potentiellen Anwendern resultiert damit in einem Koeffizient von ca. 0,05. Das würde für PegelAlarm in Falle einer in etwa vergleichbar erfolgreichen Werbekampagne ca. 32.000 Installationen bundesweit bedeuten.

Um an die maximale Anzahl an parallel aktiven App-Benutzern zu kommen ist eine weitere Einschränkung dieser Zahl notwendig: die geografische Lokalisation eines Hochwassers. Es ist sehr unwahrscheinlich, dass eine Hochwassergefahr für die gesamte Fläche Deutschlands aktuell wird. Historische Daten zu den Hochwassern im August 2002 und im Juni 2013 sind dabei die Extrembeispiele: das Hochwasser hat sich zur gleichen Zeit über mehrere Bundesländer und mehrere Flüsse erstreckt. So betraf das Hochwasser an der Elbe und an der Mulde im August 2002 etwa 370.000 Menschen (Baumgarten et al., 2012). Dies beträgt ca. 22% der 1,66 Mio. Menschen, die in Deutschland potenziell betroffen sein könnten und kann als ein Richtwert für die geografische Einschränkung angenommen werden. Damit liegt die maximale Abschätzung der parallel aktiven Benutzer im Falle einer Hochwassergefahr bei ca. 7.000, unter der Voraussetzung einer sehr hohen Popularität der App in der Bevölkerung. Eine weniger erfolgreiche Bekanntmachung der PegelAlarm führt dazu, dass mit ca. 20% dieser Zahl gerechnet wird, und resultiert in 1400 parallelen Anwendern.

Auswertung der historischen Daten geben einen Eindruck über die Ausmaße des gestiegenen Interesses der Menschen an Informationen währen eines Hochwassers. Aus den Zugriffszahlen des Hochwassers 2013 auf das Informationsangebot des Bayerischen Hochwassernachrichtendienstes (Abbildung 9) und das Informationsangebot der Hochwasservorhersagezentrale des Hessischen Landesamts für Umwelt und Geologie (Abbildung 10) ist zu sehen, dass im Falle eines Hochwassers mit bis zu einem 10-fachen Wachstum der Benutzerzahl bzw. der Aktivitäten gerechnet werden muss. Eine Rückrechnung auf die Benutzerzahl unter normalen Bedingungen kann damit gemacht werden und resultiert in etwa 140 parallelen Anwendern unter Normallast-Bedingungen. Somit ergibt sich die Abschätzung an Benutzerzahlen für alle drei Testszenarien (Tabelle 7).

18 https://www.adac.de/sp/presse/regional/nordrhein_westfalen/pm_mitgliederversammlung_adac_nordrhein_2015.aspx

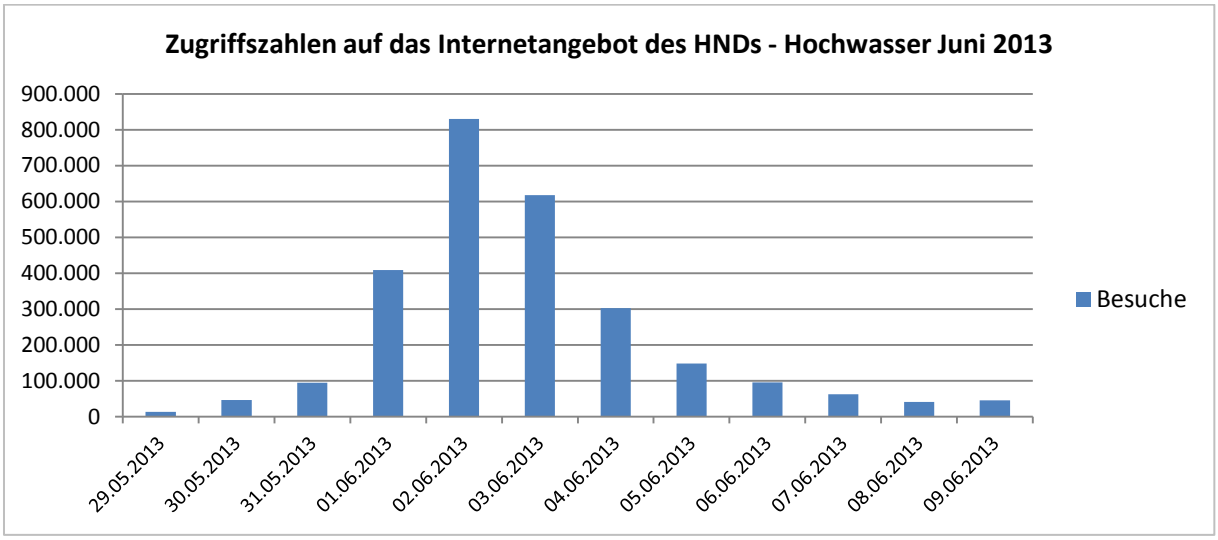

Abbildung 9 Zugriffszahlen auf das mobile Internetangebot des HND pro Tag. (Quelle: Eigene Darstellung, basierend auf Daten von (LfU, 2014)).

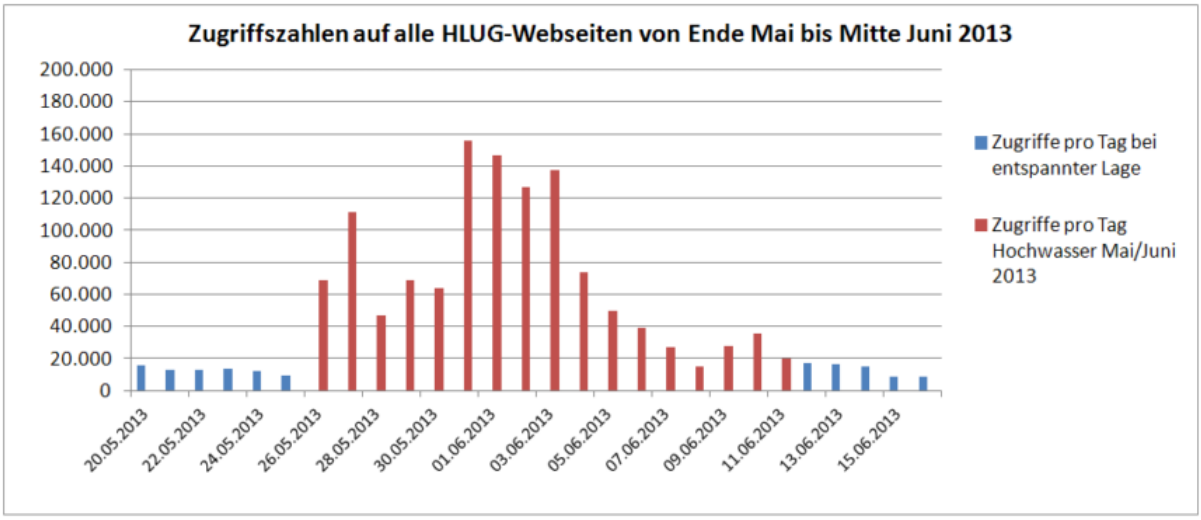

Abbildung 10 Zugriffszahlen auf das Internetangebot der Hochwasservorhersagezentrale des Hessischen Landesamts für Umwelt und Geologie (HLUG) während des Hochwassers Mai/Juni 2013. (Quelle: (Geologie, 2013)).

4.4.4 Benutzerzahl Wasserpegel-Aufzeichnung im Hintergrund

Die Wasserstands-Aufnahme ist eine Funktionalität der App, die standardmäßig vollständig ausgeschaltet ist und von Anwender manuell gestartet werden muss. Wird diese Funktionalität in Anspruch genommen, so wird eine regelmäßige Aufzeichnung von Wasserständen in einem definierten Zeitintervall vorgenommen: die App fordert in regelmäßigen Abständen von ODS aktuelle Daten der *PEGELONLINE*-Datenquelle und speichert sie lokal auf dem Gerät. Die Standardeinstellung schlägt eine Synchronisierung mit ODS alle 30 Minuten vor und nur, wenn der Benutzer sich in einem Wi-Fi-Netz befindet. Dies kann jedoch vom Benutzer auf andere vordefinierten Werte angepasst werden, zum Beispiel kann er das Aufzeichnungsintervall auf 15 Minuten verkürzen.

Da diese Funktionalität im Hintergrund ausgeführt wird muss die Anzahl an parallelen Benutzer, die diese Funktionalität benutzen, separat abgeschätzt werden. Diese Abschätzung basiert auf folgenden Annahmen:

- **Keine Hochwassergefahr:** Es wird angenommen, dass die Anzahl an Anwendern, die diese Funktionalität eingeschaltet haben, ca. 10% aller App-Benutzer ausmacht – die Wasserstände ändern sich kaum, so dass eine Aufzeichnung kaum Mehrwert bringt. Unter der Grundannahme, dass die meisten Benutzer die Standardeinstellung so belassen, wie sie sind (so hat beispielsweise (Spool, 2011) gezeigt, dass neun von zehn Menschen alle Standardeinstellungen in einem beliebten Textverarbeitungspaket beibehalten), wird damit gerechnet, dass die Anzahl an Benutzern, die das Synchronisierungsintervall verkürzen, sich mit der Anzahl an Benutzer, die das Intervall entsprechend vergrößern, ausgleicht. Damit resultiert diese Abschätzung in einer durchschnittlichen Zahl von ca. 10% aller App-Anwender, die sich gleichverteilt in 30 Minuten mit ODS synchronisieren.
- **Hochwassergefahr:** Es wird angenommen, dass die Anzahl an Benutzern, die diese Funktionalität einschalten, ca. 75% aller erwarteten App-Benutzer beträgt. Weiter wird angenommen, dass nur 10% der Benutzer die Standard-Einstellungen ändern, jedoch diesmal verkürzen die Benutzer, die eine Änderung der Standard-Einstellungen vornehmen, das Intervall auf die minimal möglichen 15 Minuten.

Tabelle 7 Benutzerzahlen für definierte Testszenarien.

Benutzerzahl	Testszenarien		
	Niedriglast	Normallast	Hochlast
Max. Anzahl an parallelen Pegel-Alarm-Benutzern	140	140	1.400
Anzahl an Benutzern der Wasserpegel-Aufzeichnung über das Synchronisierungsintervall	14	105	1.050

4.4.5 Anzahl an Alarmen

Ein Wasserstandalarm (im Folgenden als Alarm bezeichnet) wird vom App-Benutzer registriert und informiert ihn, falls der Wasserstand an einer für den Benutzer relevanten Wasserstandsmessstation den vom Benutzer definierten kritischen Wert erreicht oder überschritten hat. Ein App-Benutzer kann für verschiedene Messstationen verschiedene Alarme registrieren und ist dabei in ihrer Anzahl theoretisch nicht eingeschränkt. Die erwartete Gesamtzahl an dauerhaft angelegten Alarmen befindet sich in Korrelation mit der Gesamtzahl an App-Installationen. Die Abschätzung, wie populär diese Funktionalität bei den Anwendern ist, basiert auf den Überlegungen, was einen Anwender zum Anlegen eines Alarms motivieren kann. Diese sind in Tabelle 8 vorgestellt.

Da die App aktuell keine Hinweise zu den für eine ausgewählte Station „*kritischen*“ Wasserstandwerten angibt, wird eine zufällige Alarmregistrierung als unwahrscheinlich eingestuft. Ein bewusstes Alarmanlegen ist die Primärintention hinter dieser Funktionalität – die Menschen, die in den Regionen mit erhöhtem Hochwasserrisiko wohnen, wissen in der Regel, wie hoch der nahe gelegene Fluss steigen darf, bevor die Gefahr ernsthaft für sie wird. Die durchschnittliche Anzahl an angelegten Alarmen pro App-Benutzer beträgt nach dieser Berechnung ca. 1,26.

Tabelle 8 Abschätzung der Anzahl an angelegten Alarmen pro Benutzer.

Motivation für den Benutzer, einen Alarm für eine Messstation anzulegen	Wahrscheinlichkeit	Absolute Anzahl an Alarmen	
		Große Verbreitung der App (32.000 Installationen)	Geringe Verbreitung der App (3.200 Installationen)
Messstation in der Nähe des Wohnorts	80%	25.600	2.560
Messstation in der Nähe weiterer wichtiger Orte (Arbeitsplatz, Elternhaus, Ferienhaus, etc.)	30%	9.600	960
Messstation 20 km. flussaufwärts (um rechtzeitig informiert zu sein)	5%	1.600	160
Zufällige Messstation / unterwegs in der Region / sonstige Interesse	3%	960	96
Summe		**37.760**	**3.776**
Stopplimits nach oben und nach unten (im Falle einer ungeklärten Situation – für die gleiche Messstation zwei oder mehrere Alarms anlegen, um informiert zu sein, falls die Situation sich entschärft oder richtig kritisch wird.	7% aller Vorhaben, informiert zu sein	2.644	264
Gesamtsumme		**40.404**	**4.040**
Alarme/Benutzer		**1,26**	**1,26**

Diese Erwartung ist in den Übergangswahrscheinlichkeiten der Benutzerverhaltensmodelle (Abbildung 7, Abbildung 8) wiederzufinden, als Differenz zwischen der mittleren Anzahl der Besuche des Zustandes *Alarm anlegen* vor der Absorption (*App verlassen*) und der mittleren Anzahl der Besuche des Zustandes *Alarm löschen* vor der Absorption (*App verlassen*).

In den Hochwasserzeiten wird eine hohe Anzahl an ausgelösten Alarmen erwartet. Tabelle 9 fasst die Korrelation zwischen aktiven parallelen Benutzern, der Wahrscheinlichkeit, dass ein Alarm getriggert wird und der resultierende Anzahl an gefeuerten Alarmen zusammen. Diese Anzahl wird bei der Testdatenerstellung (Abschnitt 4.5.6) berücksichtigt.

Tabelle 9 Anzahl an getriggerten Alarmen für definierte Testszenarien.

	Testszenarien		
	Niedriglast	Normallast	Hochlast
Prozentsatz an gefeuerten Alarmen insgesamt	5%	75%	75%
Gesamtzahl an gefeuerten Alarmen (mit Koeffizient 1,26)	9	132	1.218

4.4.6 Workload-Intensität über die Zeit

Wie in Abschnitt 4.4.3 beschrieben wird für PegelAlarm im Hochwasser-Fall eine erhöhte Benutzung der Aufzeichnungsfunktionalität erwartet. Betrachtet man eine Ereignis-getriggerte Situation – z. B. nach einer Hochwasserwarnung durch die Medien – dann kann damit gerechnet werden, dass viele App-Benutzer diese Aufzeichnungsfunktionalität für Wasserstände mehr oder weniger gleichzeitig anschalten. Unter Berücksichtigung der Beibehaltung der Standard-Einstellungen nach (Spool, 2011) kann dies zu einem wellenartigen Anfragenaufkommen an ODS führen (Abbildung 11). Je mehr Benutzer die App hat, desto

größer werden die Amplituden der Anfragen sein und desto mehr Last muss ODS punktuell verarbeiten können.

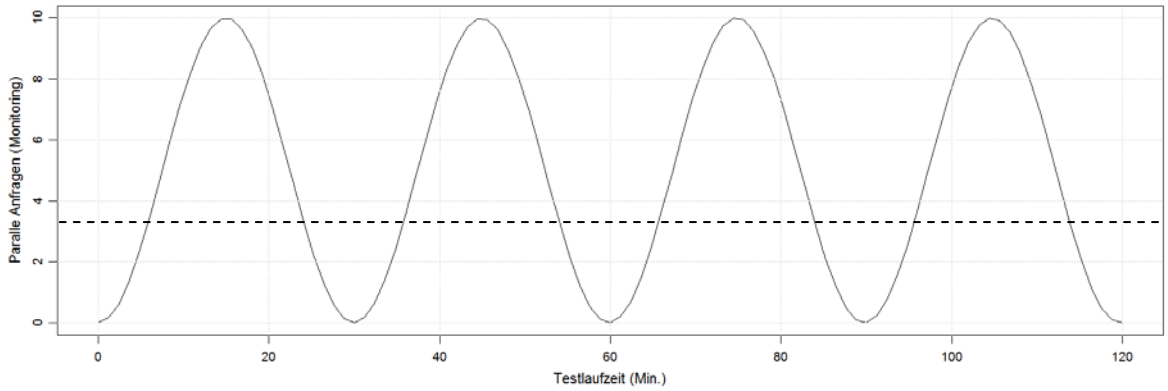

Abbildung 11 Wellenartiges Anfragenaufkommen an ODS am Beispiel von 105 Anwendern der Aufzeichnungsfunktionalität (30 Min.-Aufzeichnungsintervall). Gleichverteilt, wird mit ca. 3 parallelen Anfragen gerechnet werden, ein wellenartiges Lastaufkommen führt punktuell zu 10 parallelen Anfragen. (Quelle: Eigene Darstellung).

4.4.7 Zusätzliche Last

Die bis jetzt beschrieben Szenarien berücksichtigen nur die von der App PegelAlarm kommende Last auf ODS und CEPS. Es ist jedoch nicht klar, ob die App exklusive Instanzen von ODS und CEPS für sich zur Verfügung haben wird. Diese Überlegung wird durch eine konstante Last auf dem System durch weiteren Systeme nachgebildet: in ODS werden weiteren Datenquellen (primär Open Street Map (OSM)[19]) angebunden und regelmäßig abfragt. Damit wird ein zusätzliches Testszenario definiert: Das System wird von der App PegelAlarm unter Normallast-Bedingungen benutzt und ODS wird zusätzlich einer weiteren Last ausgesetzt.

4.5 Testvorbereitung für die L&P-Tests

Im vorliegenden Abschnitt werden die für die Testvorbereitung notwendigen Schritte beschrieben, die der eigentlichen Testdurchführung vorausgehen und die Testrandbedingungen schaffen.

4.5.1 Eingesetzte Werkzeuge

Im Rahmen der Durchführung der L&P-Tests kamen mehrere Werkzeuge zur Messung der Laufzeiten und Ressourcennutzung sowie zur Lasterzeugung zum Einsatz. Im Folgenden werden diese Werkzeuge sowie die jeweiligen Einsatzszenarien beschrieben.

4.5.1.1 Monitoring-Werkzeuge

Um die Lastsituationen auf dem SUT während der Ausführung der L&P-Tests festzustellen sind Monitoring-Werkzeuge notwendig. Ein Monitorwerkzeug misst die Ressourcennutzung des unter Last stehenden Systems und liefert die wichtigsten Kennzahlen zum Performance-Verhalten. Ein ideales Monitoring-Werkzeug beeinflusst die Ergebnisse der Messung nicht, was jedoch in der Praxis nicht möglich ist: Die Monitoring-Werkzeuge produzieren Overhead, der zu einigen Verfälschungen der

[19] http://www.openstreetmap.de

Messwerte führen kann (Jain, 1991). Ein ideales Monitoring-Werkzeug aus der Praxis ist leichtgewichtig, sodass die von ihm verursachten Ungenauigkeiten vernachlässigt werden können. Monitoring-Werkzeuge werden nicht nur für die Überwachung der Ressourcenausnutzung durch das Testobjekt eingesetzt. Auf Seiten der Lasttreiber muss auch sichergestellt werden, dass die Lastgenerierung planmäßig verläuft und nicht selber unter Performance-Problemen leidet.

Für die Überwachung der Ressourcenausnutzung wurden folgende Werkzeuge eingesetzt:

- Auf der Windows-Plattform: Windows Performance Monitor (*PerfMon*)[20].
- Auf der Linux-Plattform: *dstat*[21] und *pidstat*[22].

PerfMon

PerfMon ist ein Standard Performance-Monitoring-Werkzeug der Windows-Plattform für das Überwachen der Systemstatistiken: verschiedener Aspekte der Systemleistung (z.B. CPU- und Speicherauslastung, I/O-Raten) werden in diversen Kennzahlen erfasst. Diese Kennzahlen können in Echtzeit beobachtet oder in eine Datei geschrieben werden. Die Kennzahlen-Messungsgranularität beträgt 1 Sekunde, innerhalb dieses Zeitraums werden die Werte gemittelt und abgerundet, was zu kleinen Ungenauigkeiten führen kann und bei der Analyse berücksichtigt wurde. Um den Overhead zu minimieren wird empfohlen, die Monitoringdaten direkt in eine Datei zu schreiben und nicht die grafische Oberfläche zu verwenden. Außerdem wird empfohlen, anstatt des CSV-Formats das Binär-Format zu verwenden, wodurch der Overhead um etwa den Faktor 10 reduziert wird (Schwartz, 2006). Dies wirkt sich positiv auf die Gesamtsystemleistung und damit auf die Untersuchungsgenauigkeit aus, erschwert jedoch die Analyse der Daten: eine Betrachtung und Auswertung der gemessenen Daten ist in diesem Format nur in PerfMon selber möglich.

Dstat

Das Tool *dstat* gibt einen aggregierten Überblick über die aktuelle Systemsituation auf einem Linux-System. Alle System-Ressourcen (CPU-Auslastung, Arbeitsspeicherallokation, etc.) werden in Echtzeit überwacht. Während der Testdurchläufe wird *dstat* genutzt, um zu analysieren, welche Gesamtauslastung auf dem System verursachen wird. Die Rohdaten können auch in eine csv-Datei geschrieben werden für eine spätere grafische Analyse mithilfe eines Tabellenverarbeitungsprogramms.

Pidstat

Das Tool *pidstat* gibt Ressourcen-Statistiken zu aktiven Prozessen auf einem Linux-System aus: welche Prozesse aktiv sind, wie viel Speicher und CPU ein Prozess allokiert, etc. Während der Testdurchläufe wird *pidstat* genutzt, um die Ressourcennutzung der einzelnen Komponenten – ODC, CEPS und CouchDB, zu analysieren. Die Rohdaten können auch in eine csv-Datei geschrieben werden für eine spätere grafische Analyse mithilfe eines Tabellenverarbeitungsprogramms.

[20] https://technet.microsoft.com/en-us/library/dd744567(v=ws.10).aspx
[21] http://dag.wiee.rs/home-made/dstat/
[22] http://sebastien.godard.pagesperso-orange.fr/

4.5.1.2 Performance-Testwerkzeug

Unter einem Performance-Testwerkzeug wird ein Werkzeug zur Unterstützung der L&P-Tests verstanden (Hamburg & Löwer, 2014). Es muss im Wesentlichen zwei Funktionen bereitstellen: Lastgenerierung und Messung der Testtransaktionen. Über die Simulation vieler Benutzer und/oder hohe Eingabedatenvolumen wird die erforderliche Last auf dem SUT produziert. Die durchgeführten Transaktionen und gemessene Verhaltensstatistiken (z. B. Antwortzeiten) müssen dabei protokolliert werden. Auf Grundlage dieser Testprotokolle werden Analysen und Berichte erstellt.

Auf dem Markt sind inzwischen viele Werkzeuge präsent, die die Durchführung von L&P-Tests unterstützen (vgl. eine Übersicht von über 50 verschiedenen Open Source-Werkzeugen für L&P-Tests[23]). Diese Vielfältigkeit ist mit einem hohen Spezialisierungsgrad der Werkzeuge zu erklären.

Die Wahl eines Performance-Testwerkzeugs für den Einsatz im Projekt ist ein wichtiger Schritt: ein Werkzeugwechsel ist mit hohem Aufwand verbunden und zumeist nicht trivial. Die Entscheidung für ein Werkzeug muss nicht nur an die generellen Anforderungen geknüpft werden, wie z. B. Support, verfügbare Dokumentation, Preis, sondern auch an die relevanten Randbedingungen des Einsatzfalls. In Rahmen dieser Arbeit wurden folgende Anforderungen an ein Performance-Testwerkzeug gestellt:

- Die Möglichkeit, REST-API zu testen (HTTP und HTTPS),
- die Möglichkeit, probabilistische Modelle abzubilden,
- manuelle als auch skriptbasierte Testausführung,
- Aktualität des Werkzeuges (Weiterentwicklung/Wartung/aktive Community),

Eine Reihe von Open-Source-Werkzeugen wurde in Hinblick auf diese Anforderungen auf ihre Eignung für den Einsatz in der vorliegenden Arbeit analysiert. Kommerzielle Werkzeuge wurden aufgrund der Rahmenbedingungen dieser Arbeit nicht in die Analyse mit einbezogen. Die beiden in die engere Auswahl genommenen Werkzeuge sind *Grinder*[24] und *JMeter*[25].

Der folgende Abschnitt stellt einen kurzen Vergleich zwischen den beiden Werkzeugen dar und zeigt die Gründe für die Entscheidung für die Verwendung von *JMeter*.

Grinder

Grinder ist ein open-source Java-basiertes Lasttest-Framework, das unter einer BSD-artigen Lizenz zur Verfügung gestellt wird. *Grinder* bietet einen generischen Ansatz für L&P-Tests von beliebigen Softwaresystemen, die eine Java-API haben. Test-Skripte werden in den dynamischen Skriptsprachen *Jython*[26] und *Clojure*[27] geschrieben. *Grinder* verfügt über die Möglichkeit, einen verteilten L&P-Test zu starten (mit vielen Last-generierenden Maschinen). Die Simulation von unterschiedlichem Benutzerverhalten kann über einzelne Last-generierende Maschinen implementiert werden, die durch eine zentralisierte Instanz verwaltet werden. Die Tests können nur skriptbasiert ausgeführt werden.

[23] http://www.opensourcetesting.org/performance.php
[24] http://grinder.sourceforge.net/
[25] http://jmeter.apache.org/
[26] http://www.jython.org/
[27] http://clojure.org/

JMeter

JMeter ist ein in Java geschriebenes Open-Source-Werkzeug zum Ausführen von L&P-Tests für Client/Server-Anwendungen, das von der Apache Software Foundation entwickelt wurde und unter der Apache License, Version 2.0, zur Verfügung steht. *JMeter* kann in 2 Modi benutzt werden: der GUI-Modus ist für die Erstellung des Testplans sowie Überprüfung und Debugging, der Non-GUI-Modus hingegen für die eigentliche Testdurchführung. Die Testskripte werden als JMX-Dateien (*JMeter* spezifisches XML-Format) gespeichert. JMeter skaliert gut und stellt eine Möglichkeit des verteilten Testens zur Verfügung: die *JMeter*-Tests können auf mehreren Maschinen gestartet werden, um eine hohe Last zu simulieren. *JMeter* hat eine aktive Community und ist durch eine zur Verfügung gestellte API sehr erweiterbar, was in einer großen Anzahl an externen Plug-Ins[28] resultiert (inkl. *Markov4JMeter*, das die Abbildung der Nutzerverhaltensmodelle unterstützt; diverse Komponenten erleichtern die Auswertung der Testergebnisse, etc.).

Die beiden Werkzeuge unterscheiden sich vor allem in ihrer Architektur – eine Skript-basierte Natur von *Grinder* im Vergleich zu einer Komponenten-basierten Struktur von *JMeter* – und werden oft gegenübergestellt (Vgl. Artikel von (Bear, 2006)). Die Entscheidung für den Einsatz von *JMeter* in dieser Arbeit basiert vor allem auf seiner aktiven Community und der großen Anzahl an Plug-Ins, inkl. *Markov4JMeter*, das das modellbasierte Testen ermöglicht.

4.5.2 Testumgebung

Die Hardwarekomponenten, Netzkonfigurationen und notwendigen Werkzeuge bilden die Testumgebung (Meier et al., 2007), in der die L&P-Tests ausgeführt werden. Eine ideale Testumgebung bildet die Produktivumgebung exakt nach, mit dem Zusatz von Lasttreiber- und Monitoring-Werkzeugen. Ist dies nicht der Fall, so ist es notwendig, dass die Testumgebung zumindest in ihren Hauptmerkmalen der Produktivumgebung ähnlich ist: die Ausstattung der eingesetzten Hardwarekomponenten und die zur Verfügung stehende Bandbreite müssen realistisch abgebildet werden. Dies ist eine notwendige Voraussetzung dafür, dass valide Schlüsse aus den Testergebnissen auf den Produktiveinsatz des SUT gezogen werden können. Die Gültigkeit von L&P-Tests kann außerdem nur unter Voraussetzung einer stabilen Testumgebung und eines stabilen Testobjekts gewährleistet werden: mögliche Umgebungsschwankungen, verursacht durch auf der Testumgebung parallel laufende Software, sollten minimiert werden.

Für die in dieser Arbeit beschriebene L&P-Testdurchführung wurden zwei Testumgebungen aufgebaut. Während die Testumgebung #1 sehr flexibel ist und für die ersten Smoke-Tests und Sizing-Tests eingesetzt wird, ist die Testumgebung #2 leistungsfähig und der Produktivumgebung ähnlich. Sie wird für die Last- und Stresstests einsetzt. Folgend werden die wichtigsten Charakteristika der beiden Umgebungen zusammengefasst.

4.5.2.1 Testumgebung #1

Testumgebung #1 besteht aus folgenden Hardware-Komponenten:

[28] http://jmeter-plugins.org/

- Rechner A mit 1 CPU (2 Cores, Hyper-Threading), 2,3 GHz, 8GB RAM, Betriebssystem: Windows 8.1 mit installierten ODS und CEPS sowie einer CouchDB-Instanz.
- Rechner B mit 1 CPU (2 Cores, Hyper-Threading), 2,5 GHz, 12GB RAM, Betriebssystem: Windows 7, wurde als Lastgenerator mit JMeter eingesetzt.
- Rechner C mit 1 CPU (2 Cores, Hyper-Threading), 2,3 GHz, 8GB RAM, Betriebssystem: Windows 8.1, wurde als HTTP-Server-Stub für Benachrichtigungen verwendet.

Diese Deployment-Konfiguration führt zu keiner Latenz der Netzwerkverbindung zwischen den beiden Services, sowie zwischen den Services und CouchDB. Außerdem befinden sich Rechner A und B im kabelgebundenen lokalen Netz mit 100MBit/Sek. (Ethernet), Rechner A und C kommunizieren über WLAN mit 144 MBit/Sek. Die Kommunikation zwischen allen Maschinen erfolgte über das HTTP-Protokoll.

Der Testumgebungsaufbau ist auf der Abbildung 12 zu sehen.

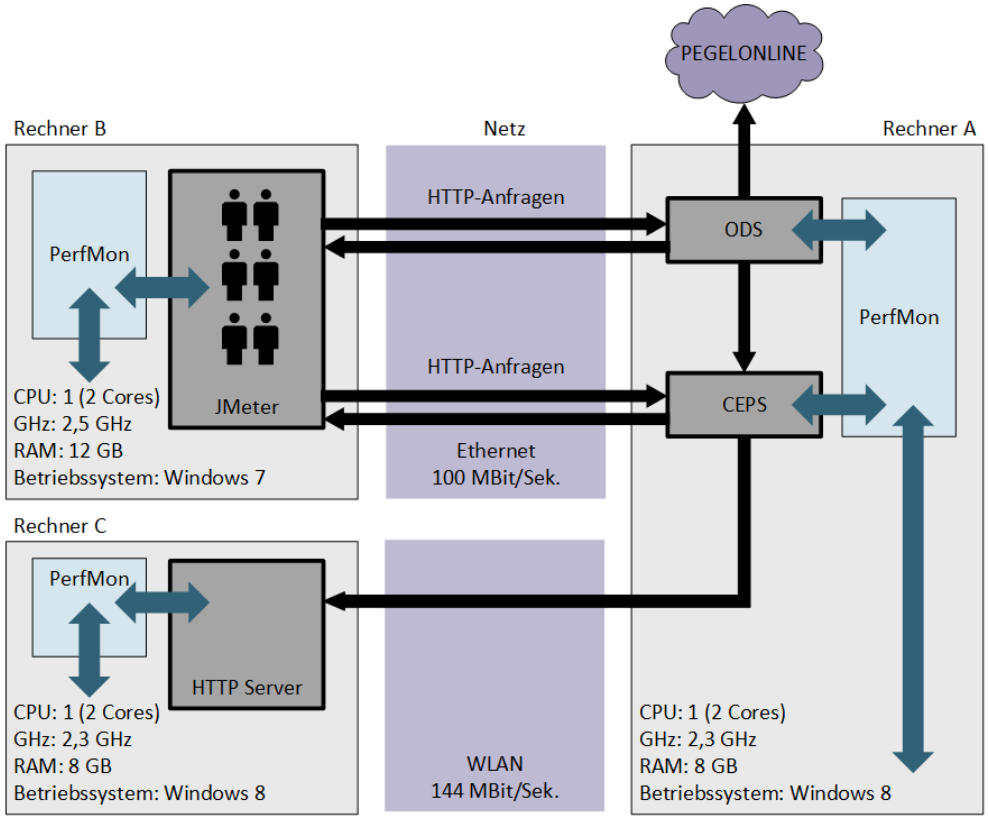

Abbildung 12 Testumgebung #1. (Quelle: Eigene Darstellung).

Das Ressourcenausnutzungsverhalten aller Maschinen wurde mit dem Monitoring-Werkzeug *PerfMon* (wie im Abschnitt 4.5.1.1 beschrieben) überwacht: Der System-Ressourcenverbrauch des Rechners A wurde in eine Datei für eine spätere Analyse umgeleitet. Die Überwachung des System-Ressourcenverbrauchs des Rechners B verlief im Echtzeitmodus über das User Interface und diente der Sicherstellung, dass die geplanten Lastmengen produziert werden konnten und an sich kein Bottleneck bei der Testdurchführung bereiteten. Die Überwachung des System-Ressourcenverbrauchs des Rechners B stellte sicher, dass der HTTP-Server-Stub für die ankommenden Anfragen zur Verfügung stand.

Um die Umgebungsschwankungen auf den Rechnern zu minimieren wurden alle Ressourcen-intensiven

Prozesse heruntergefahren (z. B. Antivirus, Dropbox). Die Ressourcennutzung durch das Betriebssystem, das Monitoring-Werkzeug selbst und die restlichen laufenden Prozesse wurde auf dem Rechner A im Leerlauf wiederholt protokolliert: jedes Mal nach dem Rechnerneustart und ab dem Moment, in dem sich das System in einem stabilen Zustand befand, erfolgte eine Systemverhaltens-Beobachtung über 30 Minuten. Unter der Annahme, dass der beobachtete Leerlauf-Ressourcenverbrauch sich konstant über die gesamte Testlaufzeit verhält, besteht damit keine Umgebungsschwankungsgefahr. Die Testergebnisse wurden entsprechend dieser Annahme analysiert.

4.5.2.2 Testumgebung #2

Für den L&P-Test wurden in Testumgebung #2 drei Rechner eingesetzt:

- Rechner A mit 4 CPU (8 Cores), 2,8 GHz, 8GB RAM, Betriebssystem: Debian GNU/Linux 8.0 (jessie) mit installierten ODS und CEPS sowie einer CouchDB-Instanz.
- Rechner B mit 1 CPU (2 Cores), 2,5 GHz, 12GB RAM, Betriebssystem: Windows 7 wurde als Lastgenerator mit JMeter eingesetzt.
- Rechner C mit 1 CPU (2 Cores), 2,3 GHz, 8GB RAM, Betriebssystem: Windows 8.1 wurde als HTTP-Server-Stub verwendet.

Diese Deployment-Konfiguration erzeugt ebenfalls keine Latenz der Netzwerkverbindung zwischen den beiden Services, sowie zwischen den Services und CouchDB. Rechner A befindet sich in dem Universitätsnetz, Rechner B und C befinden sich außerhalb des Universitätsnetzes mit 16 MBit/Sek. Die Kommunikation zwischen allen Maschinen erfolgt über das HTTPS-Protokoll.

Der Testumgebungsaufbau ist in Abbildung 13 dargestellt.

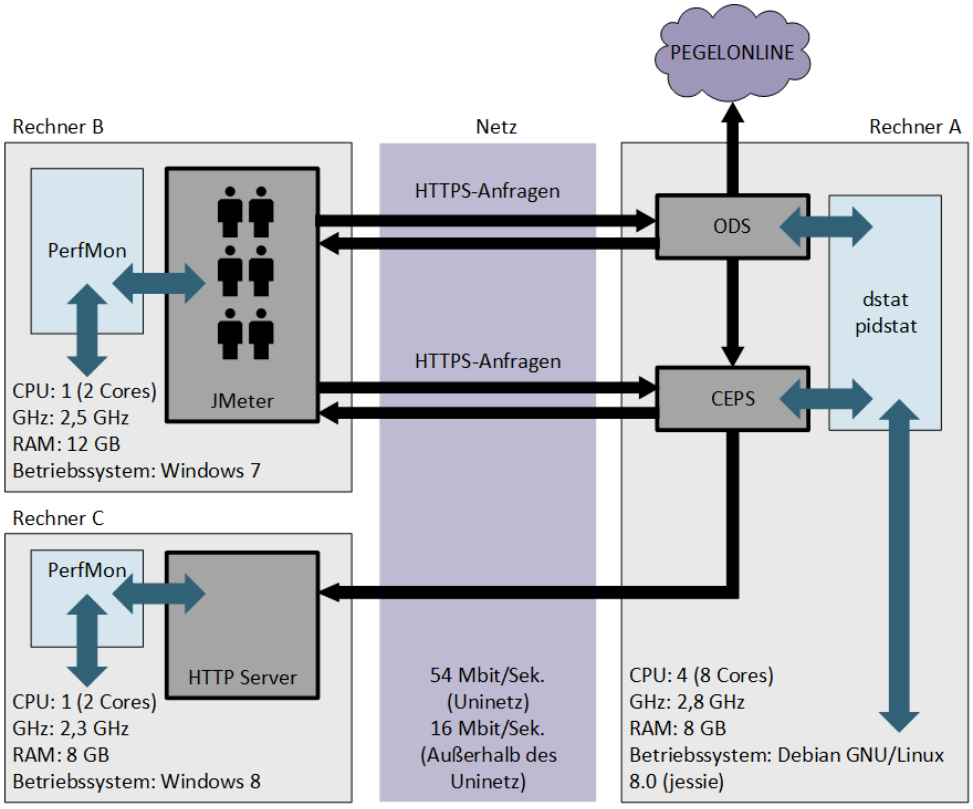

Abbildung 13 Testumgebung #2. (Quelle: Eigene Darstellung).

Rechner A ist vom Lehrstuhl zur Verfügung gestellt. Es handelt sich um eine virtuelle Maschine, die auf der *ESX*[29]-Virtualisierungslösung (*VMware*[30]) läuft, auf der ODS, CEPS und CouchDB jeweils in einem eigenen Docker-Container deployed sind – was keine Latenz der Netzwerkverbindung zwischen ODS, CEPS und CouchDB bedeutet. Sie sind das Einzige, das auf der Maschine läuft, somit werden keine Umgebungsschwankungen erwartet. Auf Rechner A laufen die Monitoring-Werkzeuge *dstat* und *pidstat* um die aktuelle Systemsituation zu überwachen: Aus dem Ressourcenverbrauch des Rechners A kann auf den Ressourcenverbrauch des SUT geschlossen werden. Das Monitoring-Werkzeug *PerfMon* auf Rechner B gibt Hinweise darauf, ob die geplante Lastmenge produzieren werden kann und an sich kein Bottleneck bei der Testdurchführung darstellt. Die Ressourcen-Anforderungen an den Lastgenerator (wie viel Rechenkapazität JMeter für die Lastgenerierung benötigt) basieren auf Erfahrungswerten der Community: als Richtwert für einen durchschnittlichen Rechner hat sich als Obergrenze die Menge von ca. 1.000 JMeter-Threads[31] (je nach Komplexität und Intensivität) bewährt. Für einen Stresstest, bei dem mehr als 1.000 Benutzer simuliert werden sollen, wurde ein verteiltes Testnetz aus 3 JMeter-Agents mit der Ausstattung des Rechners C aufgebaut.

Eine Bestimmung der Ressourcennutzung seitens des Betriebssystems und der Monitoring-Werkzeuge selber erfolgte über eine Reihe von Systemmonitoring-Messungen im Leerlauf auf dem Rechner A. Unter der Annahme, dass der festgestellte Ressourcenverbrauch sich in dem beobachteten Ausmaß konstant über die gesamte Testlaufzeit verhalten wird, können die Testergebnisse entsprechend analysiert werden.

4.5.3 Testskripte zur Lasterzeugung

Die Durchführung der L&P-Tests wurde mittels des Einsatzes des Lastgenerators JMeter 2.13 automatisiert (siehe auch Abschnitt 4.5.1.2). Mithilfe eines JMeter-Testkripts wurde gemäß den in Abschnitt 4.4 vorgestellten Testszenarien Last generiert und auf das System gerichtet. Die Erstellung von Testskripten ist keine triviale Aufgabe und erfordert ein tief gehendes Verständnis dessen, wie die Anwender mit dem SUT interagieren werden. Die ganze Vielfalt der Benutzerinteraktionen mit dem SUT wurde in einem JMeter-Skript abgebildet. Folgend werden die wichtigsten Informationen zu der Skript-Implementierung aufgeführt.

Der Testplan (*TestPlan* in JMeter) ist das Wurzelelement jedes JMeter-Testskripts. Er steuert den Ablauf der Lastgenerierung und besteht aus einer oder mehreren Thread-Gruppen (*ThreadGroup* in JMeter). Eine Thread-Gruppe repräsentiert eine Menge von Benutzern, die sich gleich verhalten und einen bestimmten Ablauf an Aktionen ausführen. Eine Thread-Gruppe bildet damit all die Benutzer ab, die ein Anwendungsszenario durchspielen. Alle Thread-Gruppen werden gleichzeitig gestartet und in ihrer Gesamtheit bilden sie die auf das System gerichtete Gesamtlast.

Die Generierung der in Abschnitt 4.4 vorgestellten Lastmengen wurde über folgende Thread-Gruppen erreicht:

- Thread-Gruppe *PegelAlarm Users*,

[29] http://www.vmware.com/products/esxi-and-esx/overview.html
[30] http://www.vmware.com/de
[31] http://wiki.apache.org/jmeter/HowManyThreads

- Thread-Gruppe *Monitoring Users,*
- Thread-Gruppe *Artificial DataPush,*
- Thread-Gruppe *Extra Users.*

Die Thread-Gruppe *PegelAlarm Users* repräsentiert den auf den Nutzerverhaltensmodellen (wie im Abschnitt 4.4.1 beschrieben) basierenden Interaktionsablauf zwischen Benutzer und der App. In JMeter wird diese Abbildung von Nutzerverhaltensmodellen auf die Testskripte durch das von (Van Hoorn et al., 2008) vorgestellten Plugin Markov4JMeter ermöglicht. Das Plugin stellt einen zusätzlichen Controller zur Verfügung – *Markov Session Controller* – der die Nutzerverhaltensmodelle (in Form von Übergangsmatrizen, wie im Abschnitt 4.4.1 dargestellt) mit entsprechenden Gewichten referenziert. Ihm untergeordnet ist eine Reihe von *Markov State*-Testelementen, die die eigentlichen HTTP-Anfragen (*HTTP-Sampler* in JMeter) an ODS und CEPS seitens der Benutzer kapseln. Jeder Übergang von einem Markov-Zustand in einen anderen geschieht gemäß den Wahrscheinlichkeiten in den Übergangsmatrizen. Die Abbildung der Nutzerverhaltensmodelle auf JMeter Elemente ist in Abbildung 14 dargestellt.

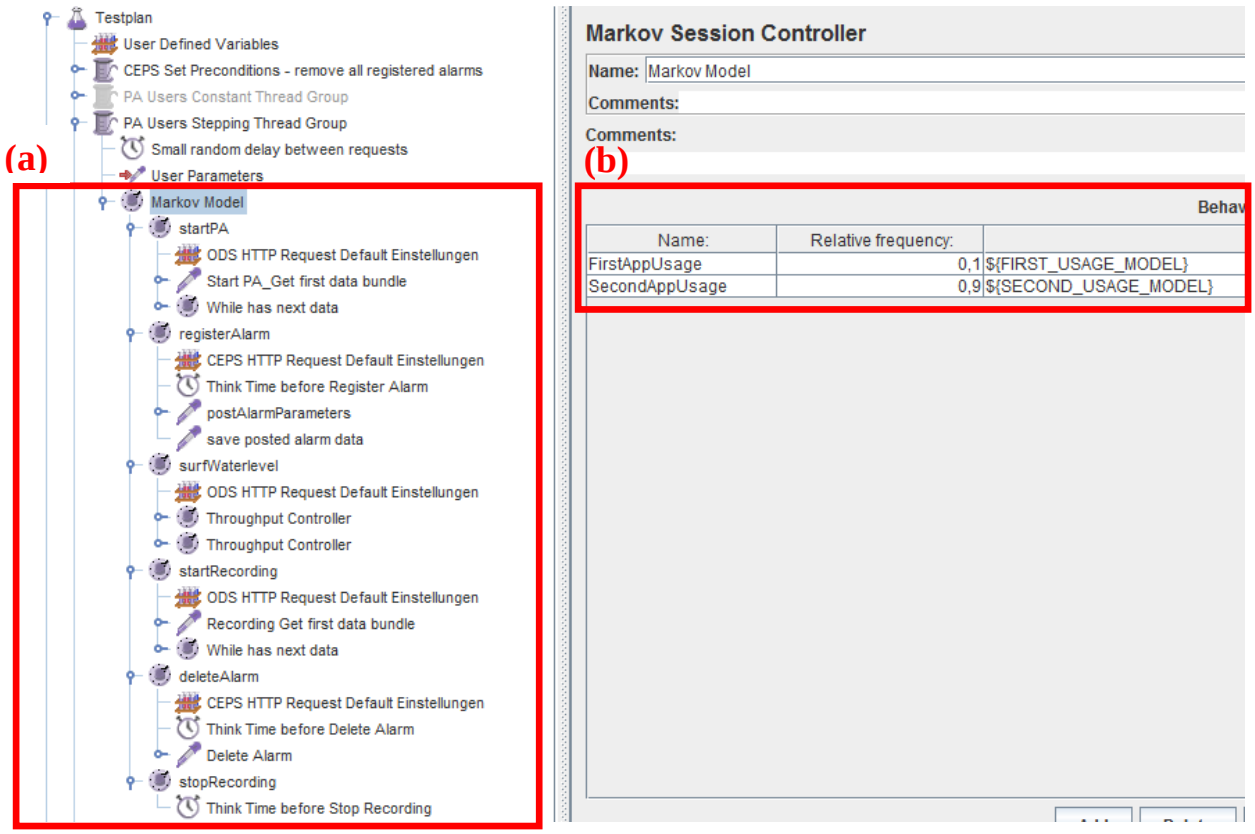

Abbildung 14 (a) Abbildung des Markov-Modells auf eine JMeter Thread-Gruppe: die Zustände und Kanten sind sequentiell angeordnet, die Reihenfolge ihrer Ausführung beruht jedoch auf den in Nutzerverhaltensmodellen definierten Wahrscheinlichkeiten. (b) Nutzerverhaltensmix mit Gewichten. (Quelle: Eigene Darstellung).

Die Thread-Gruppe *Monitoring Users* repräsentiert die Benutzer der App PegelAlarm, die die Funktionalität „Aufzeichnung des Wasserpegels über die Zeit“ benutzen. Diese Funktionalität initiiert eine periodische Abfrage des kompletten Datenbestands der Datenquelle *PEGELONLINE* durch ODS und passiert auch im Hintergrund – selbst wenn der Benutzer die App verlassen hat.

Die Thread-Gruppe *Artificial DataPush* ist zuständig für die Herstellung von Situationen, in denen CEPS

selber aktiv agiert und Benachrichtigungen über getriggerte Alarme verschickt. Wegen der Testreproduzierbarkeit ist es notwendig, die Kontrolle über diese Aktivitäten zu übernehmen und sie steuern. Für den in dieser Arbeit betrachteten Einsatzfall der App PegelAlarm ist dies vor allem CEPS beim Verschicken der Benachrichtigungen an registrierten Clients, sobald Alarme getriggert werden. Um dieses aktive Verhalten von CEPS kontrolliert hervorzurufen ist es notwendig, die Bedingungen, auf denen dieses Verhalten beruht, im Testverlauf nachbilden zu können. Eine Benachrichtigung wird von CEPS verschickt sobald zwei aufeinander folgende Pegelstände einer Wasserstation beobachtet werden und der erste Pegelstand unter dem registrierten Alarm-Wert liegt, der zweite Pegelstand über diesem. Die Thread-Gruppe *Artificial DataPush* ruft dieses Verhalten hervor, indem sie entsprechend manipulierte Testdaten mit Wasserpegeln an CEPS verschickt. CEPS kann diese Daten von echten Daten, die er regelmäßig von ODS übergeben bekommt, nicht unterscheiden, Alarme feuern und CEPS verschickt Benachrichtigungen an die betroffenen Clients.

Die Thread-Gruppe *ExtraUsers* ist für die Generierung zusätzlicher Last auf dem ODS zuständig. Diese Thread-Gruppe repräsentiert eine gleichverteilte Abfrage der OSM-Daten (Anzahl an Dokumenten und Think Times als Skript-Parameter). Sie ist standardmäßig deaktiviert und wird nur in manchen Testläufen eingesetzt (wie im Abschnitt 4.4.7 vorgestellt).

4.5.4 Workload-Intensität

Die Workload-Intensität spezifiziert, wie die Anzahl an Benutzern, die simulieren werden, sich über die Zeit des L&P-Tests ändert. Diese Schwankungen über die Testlaufzeit können über eine mathematische Formel abgebildet werden. In Markov4JMeter ist der *Session Arrival Controller* dafür zuständig und reguliert die Anzahl aktiver Sitzungen basierend auf der Auswertung der vorgegebenen mathematischen Formel (diese kann z. B. als ein BeanShell[32]-Skript angegeben werden, siehe Abbildung 15). Die maximale Anzahl an aktiven Anfragen beschränkt sich auf die maximale Anzahl an parallel laufenden Nutzern. Falls die Anzahl aktiver Anfragen die vorgegebene maximale Anzahl überschreitet werden überschüssige Threads in einer Warteschlange blockiert (Van Hoorn et al., 2008).

Die Möglichkeit, den Workload in Form einer mathematischen Formel zu definieren, ist insbesondere für komplexere Workloadszenarien wie z.B. das Zugriffsverhalten im Rahmen eines Hochwasserverlaufs nützlich: hier entsteht teilweise, wie in Abschnitt 4.4.6 beschrieben, ein wellenartiges Lastaufkommen auf ODS durch die im Hintergrund ablaufenden Wasserpegelaufzeichnungen.

```
import org.apache.jmeter.util.JMeterUtils;

private static double numRequests (double x){
  return -5*Math.cos(x*0.21)+5;
}

long startMs = Long.parseLong(JMeterUtils.getPropDefault("TEST.START.MS",""));
float expMin = (float)(System.currentTimeMillis()-startMs)/(double)(1000*60);
return (int) numRequests(expMin);
```

Abbildung 15 BeanShell-Skript zur Abbildung der variierenden Anzahl an aktiven Nutzern in Abhängigkeit von der verstrichenen Testlaufzeit während der Testdurchführung (für 105 Benutzer). (Quelle: Eigene Darstellung).

[32] http://www.beanshell.org/

4.5.5 HTTP-Stub

CEPS verschickt Benachrichtigen an die betroffenen Clients, sobald ein registrierter Alarm feuert. Zur Abbildung der Client-Schnittstelle, die Nachrichten von CEPS mit den Informationen zu gefeuerten Alarmen (primär die *EventID*) entgegennimmt, wurde ein HTTP-Server-Stub verwendet. Er bietet dem CEPS eine gültige HTTP-POST Schnittstelle an und verhindert damit das Auftreten der Fehler „Client nicht erreichbar“. Außerdem sammelt er die Statistiken zu der Anzahl an angenommenen Benachrichtigungen.

4.5.6 Testdatenerstellung

Unter Testdaten werden die Daten verstanden, die vor der Ausführung eines Tests existieren und die Ausführung der Komponente bzw. des SUT beeinflussen bzw. dadurch beeinflusst werden (Hamburg & Löwer, 2014). Im Rahmen des Testdatenmanagements werden die Anforderungen an die Testdaten analysiert, die Testdatenstrukturen entworfen und die entsprechenden Testdaten, oft mithilfe eines Testunterstützungswerkzeuges, erstellt. Diese Werkzeuge (auch Testdateneditoren und -generatoren genannt) generieren, verändern oder selektieren die für die Testdurchführung benötigten Daten.

Testdaten, die für ein funktionales Testen verwendet werden, können nur selten den Anforderungen der L&P-Tests genügen: entweder ist die Menge an vorhandenen Daten nicht ausreichend oder der Inhalt der vorhandenen Daten spiegelt nicht die kritischen Performance-Szenarien wider (Bernardo & Hillston, 2007). Die Erstellung der Testdaten ist daher im Rahmen der Vorbereitung der L&P-Tests eine wichtige Aktivität. Die Projekterfahrung zeigt folgende Herausforderungen bei der Vorbereitung der Testdaten für einen L&P-Test:

- Es wird eine große Menge an Testdatensätzen benötigt, oft abhängig von der Anzahl an simulierten Benutzern, die fachlich auf die dem Test zugrundeliegenden Abläufe passen.
- Testdaten werden im Verlaufe der L&P-Tests „verbraucht“: sie werden verändert und können nach dem ersten Gebrauch nicht wiederverwendet werden.
- Die Daten müssen eine fachlich relevante Verteilung aufweisen, insbesondere, wenn sie den Ablauffluss des Systems steuern – schließlich wird die Aussage über das Systemverhalten in Bezug auf die gewählten Testdaten gemacht.

Eine der besten Testdatenquellen sind die echten Produktivdaten selber: Testdaten werden in dem Fall direkt aus Produktionsdatenbanken oder Protokolldateien extrahiert (Meier et al., 2007). Echtdaten berücksichtigen alle existierenden Testdatenabhängigkeiten und müssen auf ihre Glaubwürdigkeit nicht validiert werden. Eine solche Echtdatenextraktion ist jedoch aufgrund vieler Hindernissen nicht immer möglich: Produktivdaten stehen nicht immer zur Verfügung, personenbezogene Produktivdaten unterliegen Datenschutzgesetzen und dürfen erst nach einem aufwändigen Anonymisierungsvorgang verwendet werden, etc.

Für die Durchführung der L&P-Tests von ODS und CEPS werden folgende Datensätze benötigt (jeweils ein csv-Datei):

- Station IDs aus der ODS Datenquelle *PEGELONLINE*, um das Erkunden der Wasserstände der einzelnen Stationen nachzubilden.

- Zuordnung aller Stationen zu einem entsprechenden Gewässer, um die Abfrage der Wasserstände aller Stationen entlang eines Gewässers nachzubilden.
- Client-Registrierungsdaten, die Benutzer der App PegelAlarm repräsentieren und einen Wasserstands-Alarm registrieren.
- Fiktive Wasserstände entsprechend den registrierten Alarmen (um die Alarme kontrolliert auslösen zu können).

Die benötigten Daten sind unterschiedlicher Komplexität: Station IDs und Zuordnungen der Stationen zu entsprechenden Gewässern können über direkte Datenbank-Abfragen aus Produktivdaten extrahiert werden. Im Rahmen der eingesetzten dokumentorientierten Datenbank Couch DB geschieht dies über das Anlegen einer View mit entsprechender Map-Reduce-Funktionalität (z. B. für die Beschaffung aller Messstationen entlang eines Flusses siehe Abbildung 16). Diese Abfragen liefern immer aktuelle Echtdaten, die sofort für die Testdurchführung eingesetzt werden können.

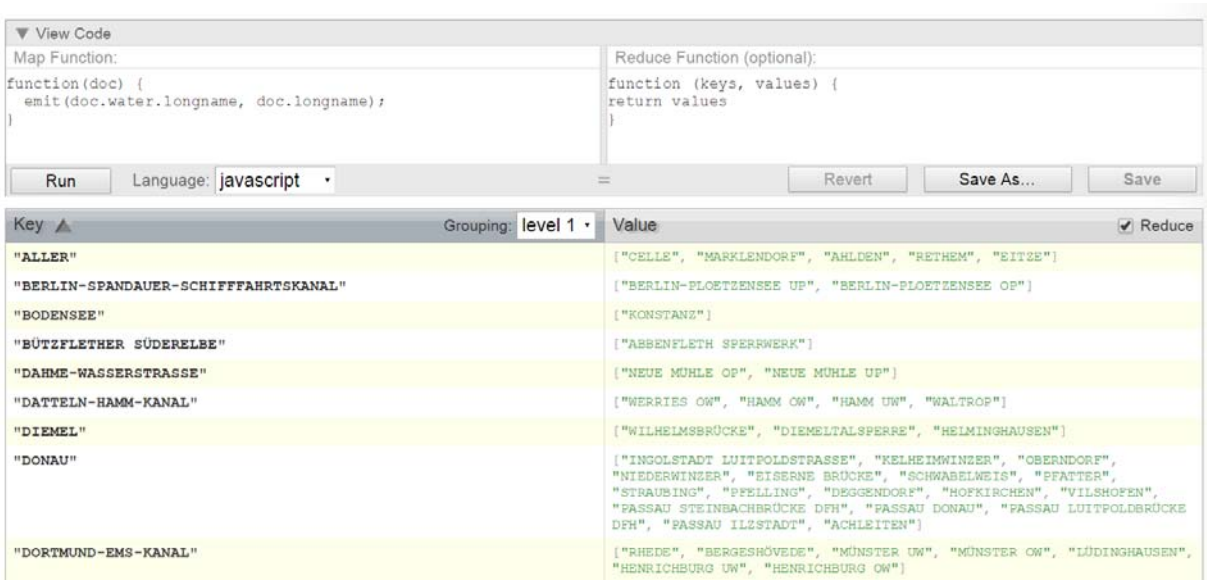

Abbildung 16 CouchDB-View für die Beschaffung aller Messstationen entlang eines Flusses (Quelle: Eigene Darstellung).

Client-Registrierungsdaten und fiktiven Wasserstände sind etwas komplexer und müssen in Relation zueinander erstellt werden. Aufgrund der benötigten Menge und existierenden Abhängigkeiten konnte ihre Erstellung nicht manuell durchgeführt werden. Client-Registrierungsdaten werden durch die Testdurchführung verbraucht: eine Client ID muss pro EPL-Adapter (siehe Abschnitt 3.7) eindeutig sein und verursacht einen Konflikt im Falle einer doppelter Belegung. Die Lösung dieser Herausforderungen lieferte die Entwicklung eines Java-Tools, das eine gewünschte Anzahl an Wasserstandsdaten und entsprechenden Client-Registrierungsdaten, die auch den definierten Lastszenarien genügen, aus einem Abzug der Produktivdaten generiert (siehe Abbildung 17) und somit der gewünschten Flexibilität bei der Erstellung der Daten, der benötigten Mengen und der Reproduzierbarkeitsanforderung genügt.

```
                                {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_27_0   {"STATION":"'STRAUBING'","RIVER":"'DONAU'","LEVEL":295.0}}
                                {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_27_1   {"STATION":"'STRAUBING'","RIVER":"'DONAU'","LEVEL":295.0}}
                                {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_27_2   {"STATION":"'STRAUBING'","RIVER":"'DONAU'","LEVEL":295.0}}
```

```
                        {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_27_3   {"STATION":"'STRAUBING'","RIVER":"'DONAU'","LEVEL":295.0}}

                        {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_27_4   {"STATION":"'STRAUBING'","RIVER":"'DONAU'","LEVEL":295.0} ()}

                        {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_28_0   {"STATION":"'PFATTER'","RIVER":"'DONAU'","LEVEL":364.0}}

                        {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_28_1   {"STATION":"'PFATTER'","RIVER":"'DONAU'","LEVEL":364.0}}

                        {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_28_2   {"STATION":"'PFATTER'","RIVER":"'DONAU'","LEVEL":364.0}}

                        {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_28_3   {"STATION":"'PFATTER'","RIVER":"'DONAU'","LEVEL":364.0}}

                        {"type":"HTTP","deviceId":"http://192.168.1.12:8080","eplArguments":
adapterPA   client_28_4   {"STATION":"'PFATTER'","RIVER":"'DONAU'","LEVEL":364.0}}
```

Abbildung 17 Generierte Client-Registrierungsdaten (Ausschnitt), die im JMeter-Skript genutzt werden, um die Registrierung der Alarme nachzubilden: 5 Registrierungsdatensätze pro Messstation, deren Messwerte in einer weiteren Testdatendatei entsprechend manipuliert sind um diese Alarme auszulösen. (Quelle: Eigene Darstellung).

Die Messwerte der einzelnen Stationen werden manipuliert und an CEPS weitergeleitet. Es werden zwei Datensätze vorbereitet: Der erste beinhaltet die unveränderten Wasserstände, die in dem Moment des Produktionsdatenabzuges registriert wurden. Alle Wasserstände des zweiten Datensatzes werden um einen festgelegten Wert erhöht. Entsprechend werden die Alarme, die im Laufe des Tests ausgelöst werden müssen, auf einen Wasserwert registriert, der in der Mitte zwischen den beiden Werten liegt (ursprüngliche Werte werden um die Hälfte des festgelegten Wertes erhöht).

4.5.7 Test-Vorbedingungen

Test-Vorbedingungen sind die „Bedingungen an den Zustand des Testobjekts und seiner Umgebung, die vor der Durchführung eines Testfalls oder Testablaufs erfüllt sein müssen“ (Hamburg & Löwer, 2014). Somit ist die Herstellung der definierten Vorbedingungen eine notwendige Voraussetzung für die Testdurchführung.

Um die Testdurchführung zu ermöglichen müssen ODS und CEPS in einen Zustand gebracht werden, in dem die folgenden Konfigurationen vorgenommen sind:

Für ODS:

- *PEGELONLINE* ist als Datenquelle mit entsprechenden Filtern (*JsonSourceAdapter*, *DbInsertionFilter*, *NotificationFilter*) angelegt.

Für CEPS:

- CEPS ist als ODS-Subscriber registriert.
- *PEGELONLINE* ist als Datenquelle, entsprechend ODS, angelegt.
- Der *PEGELONLINE*-EPL-Adapter (siehe Abbildung 18, rechts) ist angelegt, der die vom Benutzer übergebenen Parameter in eine gültige EPL-Anweisung (wie im Abschnitt 3.7 beschrieben) umwandelt.
- Die in den vorhergehenden Tests angelegten Alarme sind gelöscht.

Das Aktualisierungsintervall von ODS mit der Datenquelle *PEGELONLINE* wird auf 15 Minuten festgelegt: Die *PEGELONLINE*-Datenquelle veröffentlicht neue Daten im Durchschnitt jede Minute,

jedoch würde ein einminütiges Aktualisierungsintervall eine hohe Auslastung von ODS und CEPS bedeuten. Der Trade-off zwischen der Aktualität der Daten und der Ressourcenausnutzung basiert auf folgenden Annahmen: im Falle eines Hochwassers kann das Wasser mit einer bestimmten, nicht unendlich-hohen Geschwindigkeit steigen. Historische Spitzenwerte liegen bei ca. 50 Zentimeter pro Stunde[33]. Im Extremfall würde das einen Wasseranstieg von ca. 12,5 Zentimeter bedeuten, ohne dass die App-Benutzer aktuelle Daten zur Verfügung hätten. Das scheint ein akzeptabler Kompromiss zu sein.

Der geforderte Testobjektzustand wird mithilfe eines JMeter-Skript hergestellt (siehe Abbildung 18), das über die REST-API die oben beschriebenen Konfigurationen von ODS und CEPS vornimmt.

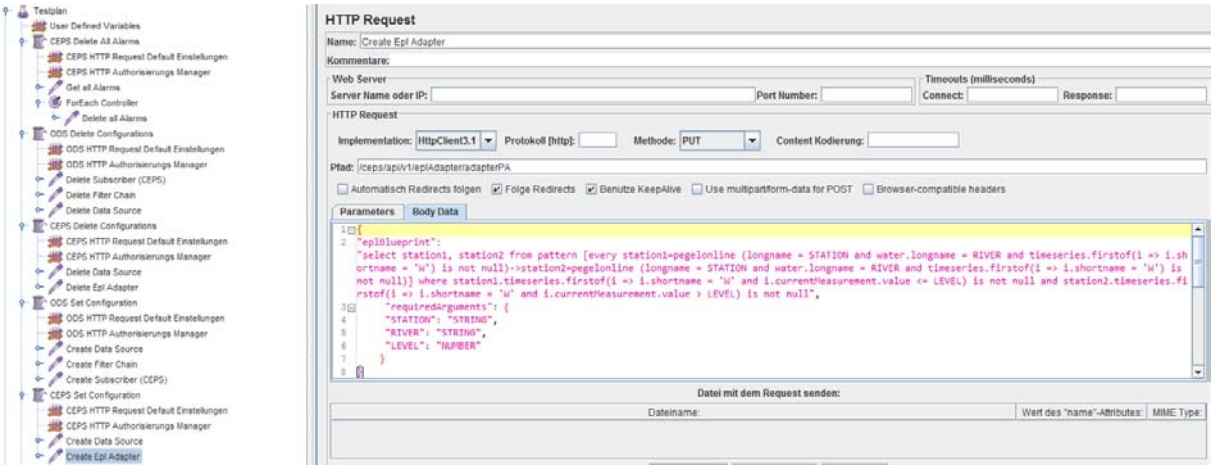

Abbildung 18 JMeter-Skript zur Erstellung der Test-Vorbedingungen.

4.5.8 Testskript-Parametrisierung

Das für den L&P-Test erstellte JMeter-Testskript bietet eine gewisse Flexibilität und kann über unterschiedliche Belegung der Parameter verschiedene Testszenarien (wie in Abschnitt 4.4 definiert) nachbilden. Folgend sind die Testskript-Parameter aufgeführt, die die Menge der generierten Last steuern:

- Anzahl an parallel laufenden Prozessen, die für die Ausführung des Testskripts genutzt werden (*Thread Count* in JMeter).
- Nutzungsintensität, die eine sich ändernde Anzahl an Benutzern spezifiziert (Session *Arrival Controller* in JMeter).
- Nutzerverhaltensprofile mit entsprechenden Gewichten (*Markov Session Controller* in JMeter). Dieser Parameter ist nur für die Thread-Gruppe *PegelAlarm Users* relevant.
- Anzahl an Wiederholungen. Dieser Wert definiert, wie oft jeder Thread alle ihm zugeordneten Elemente durchläuft und wirkt sich damit auf die Testdauer aus (*Loop Count* in JMeter).
- ThinkTimes zwischen einzelnen Benutzer-Aktionen.
- Rampenhochlaufzeit, die bestimmt, wie schnell die gewünschte maximale Anzahl an Benutzern im Test erreichen wird (*RumpUpTime* in JMeter).

In der Tabelle 10 sind alle Belegungen der Skript-Parameter pro Testszenario zusammengefasst.

[33] http://www.dassu.de/wissenswertes/unser-fluggelande/hochwasserproblematik/

Tabelle 10 Parameterbelegung des Testskripts für die Erzeugung verschiedenen Lastszenarien.

Thread-Gruppe	Parameter	Testszenario			
		Niedriglast	Normallast		Hochlast
Thread-Gruppe *PegelAlarm Users*	Thread Count	140	100	140	1.400
	Session Arrival Controller	Const.	Const.	Const.	Const.
	Nutzerverhaltensprofile mit Gewichten	NB_n (0,1) WB_n (0,9)	NB_h (0,2) WB_h (0,8)	NB_h (0,2) WB_h (0,8)	NB_h (0,3) WB_h (0,7)
	Loop Count	100	100	100	100
	ThinkTimes (in ms.)	THINK_TIME_REGISTER_ALARM 3000 THINK_TIME_SURF_WATERLEVEL 2000 THINK_TIME_START_REC 2000 THINK_TIME_DELETE_ALARM 3000 THINK_TIME_STOP_REC 2000			
	RumpUpTime	140 Sek.	100 Sek.	140 Sek.	140 Sek.
Thread-Gruppe *Monitoring Users*	Thread Count	14	75	105	1050
	Session Arrival Controller	Const.	Welle	Welle	Welle
	Loop Count	100	100	100	100
	ThinkTimes (Uniform Random in ms.)	1800000	1800000	1800000	1800000
	RumpUpTime	14 Sek.	75 Sek.	105 Sek.	300 Sek.
Thread-Gruppe *Artificial DataPush*	Thread Count	1	1	1	1
	Session Arrival Controller	Const.	Const.	Const.	Const.
	Loop Count	10	10	10	10
	ThinkTimes (Const. in ms.)	300000	300000	300000	300000
	RumpUpTime	1 Sek.	1 Sek.	1 Sek.	1 Sek.
Thread-Gruppe *Extra Users*	Thread Count	100	100	100	1.000
	Session Arrival Controller	Const.	Const.	Const.	Const.
	Loop Count	400	400	400	400
	ThinkTimes (Uniform Random in ms.)	3000	3000	3000	3000
	RumpUpTime	100 Sek.	100 Sek.	100 Sek.	100 Sek.
	Data Count	50	50	50	50

NB_n = *Neuer Benutzer* (kein Hochwasser)
WB_n =*Wiederkehrender Benutzer* (kein Hochwasser)
NB_h = *Neuer Benutzer* (Hochwasser-Bedingungen)
WB_h = *Wiederkehrender Benutzer* (Hochwasser-Bedingungen)

4.6 L&P-Testdurchführung

Dieser Abschnitt beschreibt in einer komprimierten Form die durchgeführten L&P-Tests mitsamt Randbedingungen und Ergebnissen. Die vorgestellten Ergebnisse bilden die Grundlage für die Analyse in Kapitel 5.

4.6.1 Rampen-Test #1

Im Zuge des Rampen-Tests #1 erfolgt eine erste Belastung des SUT auf Testumgebung #1 (siehe Abschnitt 4.5.2.1). Durch diesen ersten Test werden grundlegende Erfahrungen mit dem Ressourcen- und Zeitverhalten des SUT gewonnen und eine Abschätzung der Lastmenge, die auf der Testumgebung abgewickelt werden kann, getroffen.

Tabelle 11 stellt die Rahmenbedingungen des Tests dar.

Tabelle 11 Rampen-Test #1: Randbedingungen und Zusammenfassung

Testziel	Langsames Herantasten, um die ersten Informationen über den Ressourcenverbrauch und das Zeitverhalten des SUT zu sammeln.
Testszenario	Niedriglastbedingungen
Testendekriterium	1 Stunde Testlaufzeit
Testabbruchkriterium	5% Transaktionsabbrüche aller abgeschickten Anfragen
Testumgebung	Testumgebung #1
Gesamte Anzahl an parallelen Benutzern	PA: Rampe von 0 bis 800 Threads Monitoring: Rampe von 0 bis 80 Threads

In Abbildung 19 ist die stufenweise ansteigende Zahl an parallelen Nutzern auf dem System zu sehen. Die untere Linie bildet, wie auch in den folgenden Tests, die simulierten Monitoring-Benutzer ab.

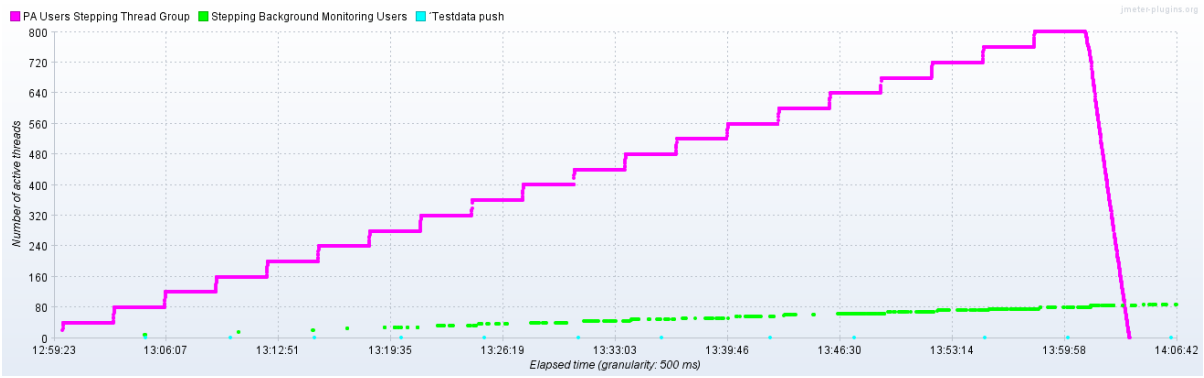

Abbildung 19 Rampen-Test #1: Rampe an parallel aktiven Benutzern. (Quelle: Eigene Darstellung).

Abbildung 20 zeigt die zunehmende Prozessorauslastung der Maschine A. Bei ca. 480 Usern ist die vorhandene CPU vollständig ausgelastet. Die anderen Hardwareressourcen sind nicht im Sättigungsbereich.

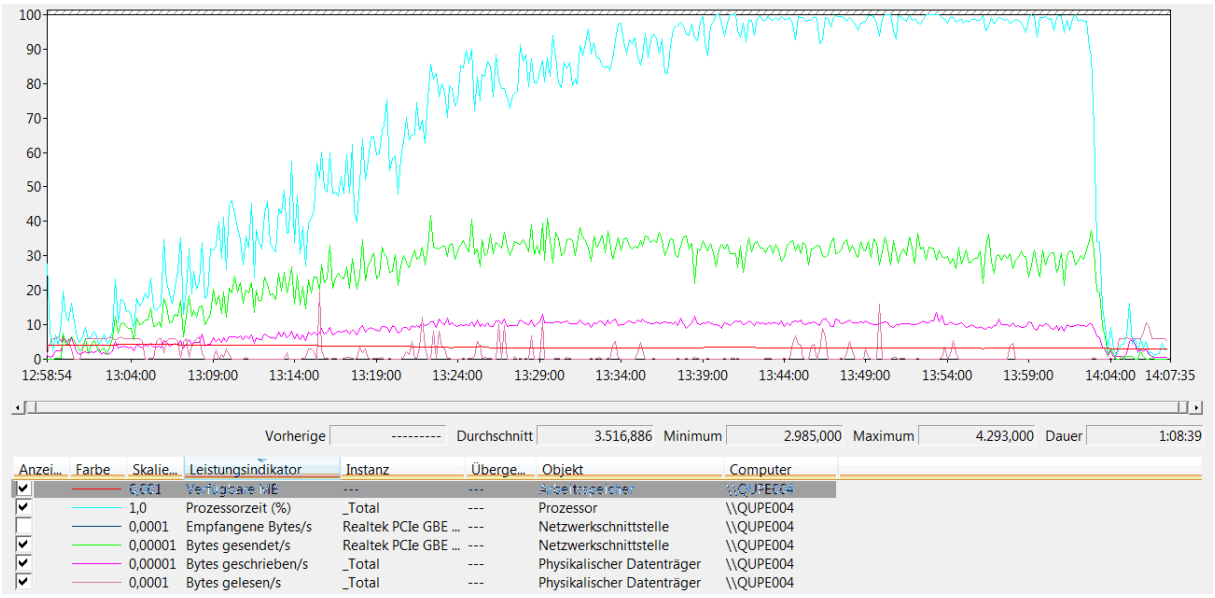

Abbildung 20 Rampen-Test #1: Gesamtressourcenverbrauch der Maschine A: Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

Betrachtet man die einzelnen Prozesse von ODS, CEPS und CouchDB, so fällt auf, dass die CouchDB beide Cores vollständig auslastet (siehe Abbildung 21, CPU-Auslastung auf Faktor 0,1 skaliert).

Bei den vereinzelten Spitzen der CPU-Auslastung durch CEPS handelt es sich um die zusätzliche Ressourcen, die die Esper Engine an sich zieht, wenn neue Daten ankommen und sie nach Events sucht. Mit dem Testverlauf nimmt die Anzahl an registrierten Alarmen zu, was für CEPS mehr Arbeit bedeutet und in den über die Testlaufzeit zunehmenden Spitzen der CPU-Auslastung resultiert (siehe Abbildung 21).

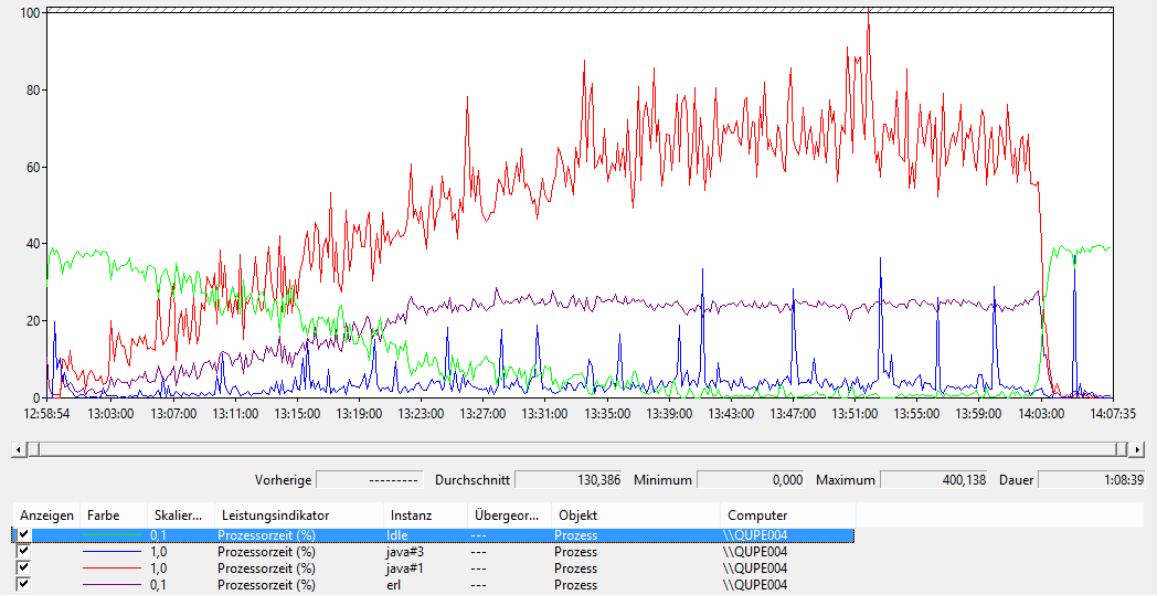

Abbildung 21 Rampen-Test #1: Prozessorauslastung durch ODS (java#1), CEPS (java#3) und CouchDB (erl). (Quelle: Eigene Darstellung).

Die Antwortzeiten aller Requests bleiben anfangs, bis zu 300 parallelen Benutzern, unterhalb der Akzeptanzkriterien und steigen ab 480 parallelen Benutzern deutlich an (vgl. Abbildung 22).

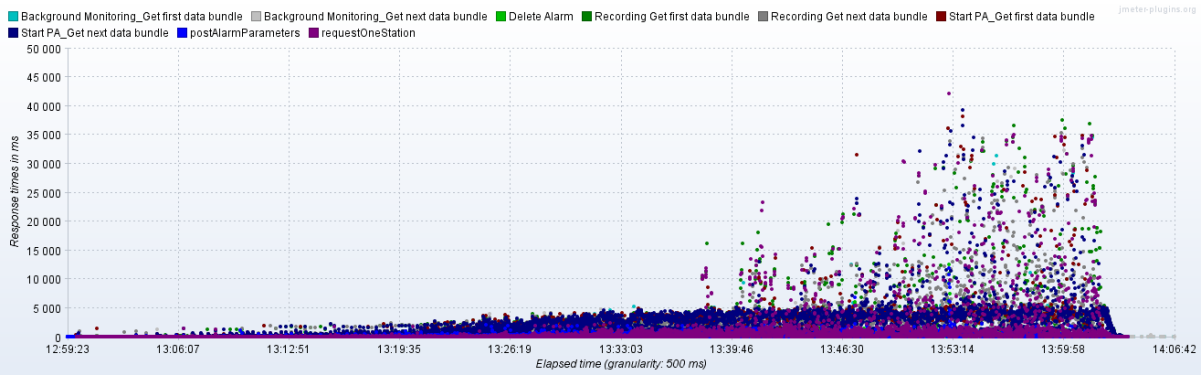

Abbildung 22 Rampen-Test #1: Antwortzeiten. (Quelle: Eigene Darstellung).

Bei der Betrachtung der einzelnen Anfragen fällt auf, dass diese sich bei den Antwortzeiten in zwei Gruppen aufteilen: die Anfragen, die mit einem Einzelwert beantwortet werden, antworten deutlich schneller als die Anfragen, die eine Menge an Ergebnissen zurück liefern (vgl. Abbildung 23).

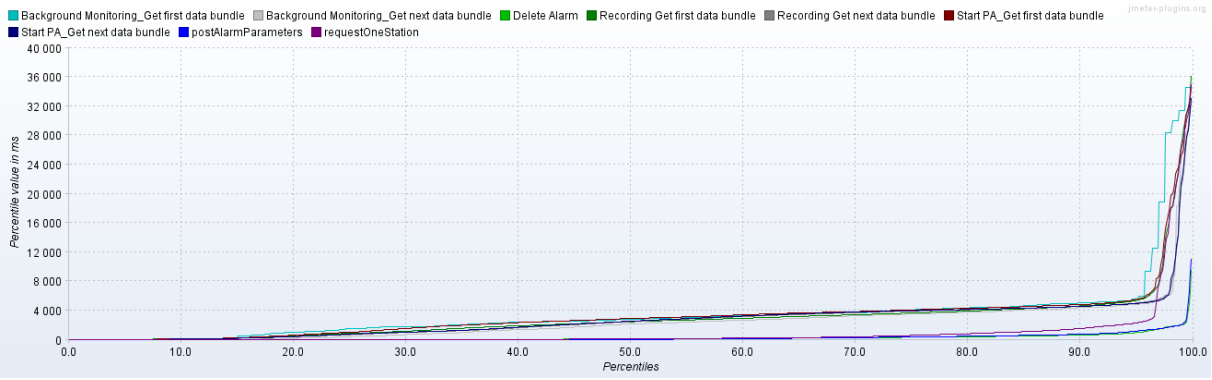

Abbildung 23 Rampen-Test #1: Streuung der Antwortzeiten. (Quelle: Eigene Darstellung).

Bis zu einer CPU-Auslastung von 80% werden die Anfragen gleichmäßig schnell beantwortet, wodurch in Abbildung 24 die Treppenfunktion der parallelen Anfragen wieder sichtbar wird. Mit zunehmender Last (über 300 parallelen Usern) steigt die Varianz der Antwortzeiten und damit des Durchsatzes deutlich an.

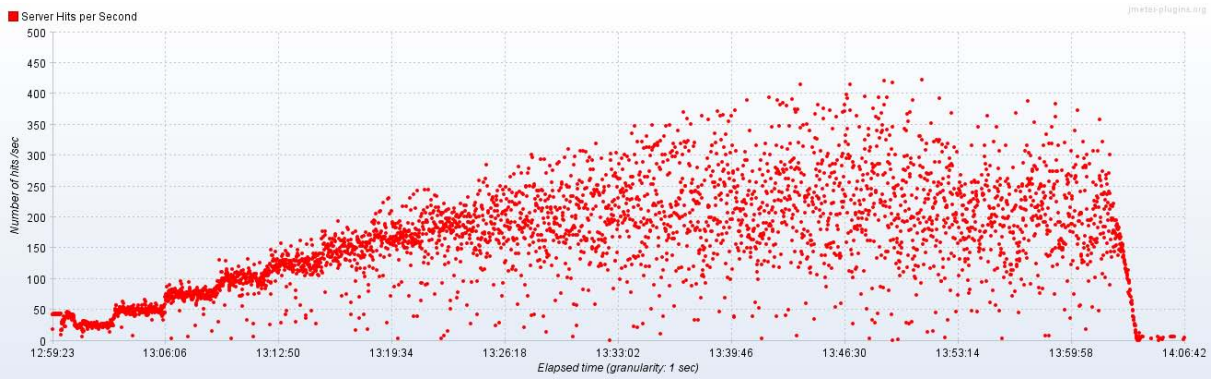

Abbildung 24 Rampen-Test #1: Durchsatz (Hits/Sec.). (Quelle: Eigene Darstellung).

4.6.2 Rampen-Test #2

Dieser Rampen-Test #2 entspricht dem vorab vorgestellten Test, jedoch auf Testumgebung #2 (siehe Abschnitt 4.5.2.2). Auch in diesem Test wird durch einen treppenartigen Anstieg der Anzahl paralleler Benutzer untersucht, wo der Maximallastbereich der Testumgebung #2 liegt. Tabelle 12 stellt die Rahmenbedingungen des Tests dar.

Tabelle 12 Rampen-Test #2: Randbedingungen und Zusammenfassung

Testziel	Langsames Herantasten, um die ersten Informationen über den Ressourcen-verbrauch und Zeitverhalten des SUT zu sammeln.
Testszenario	Niedriglastbedingungen
Testendekriterium	1 Stunde Testzeit
Testabbruchkriterium	5% Transaktionsabbrüche aller abgeschickten Anfragen
Testumgebung	Testumgebung #2 (Uninetz)
Gesamte Anzahl an parallelen Benutzern	PA: Rampe von 0 bis 900 Threads Monitoring: Rampe von 0 bis 90 Threads

Das Lastverhalten erfolgt ebenfalls stufenförmig, wird jedoch der größeren Hardware entsprechend angepasst (vgl. Abbildung 25).

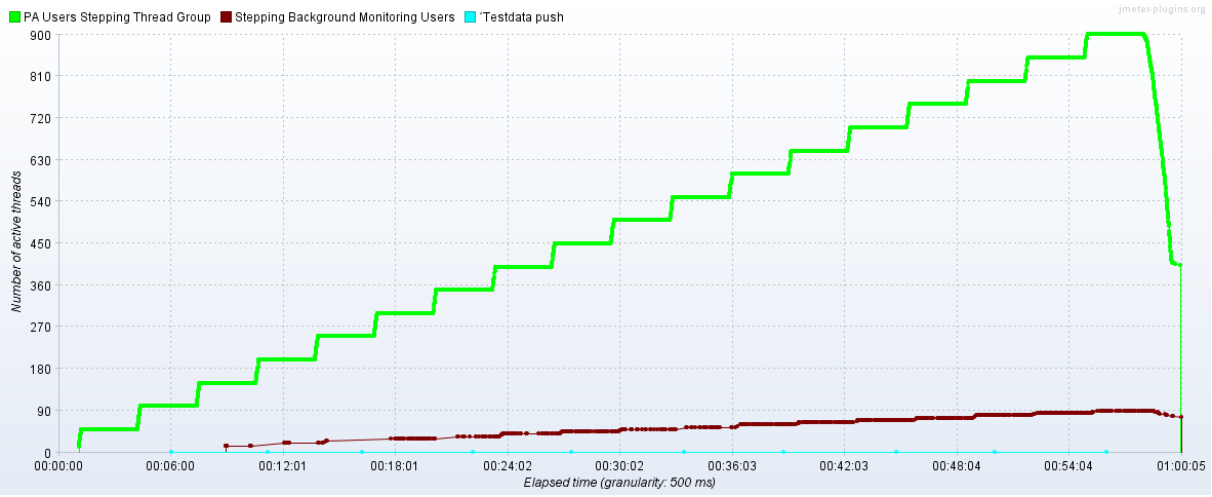

Abbildung 25 Rampen-Test #2: Rampe an parallel aktiven Benutzern. (Quelle: Eigene Darstellung).

Abbildung 26 zeigt die zunehmende Prozessorauslastung der Maschine A. Auffallend ist eine lineare Skalierung der Gesamtauslastung der CPU, die bei ca. 270 Usern stagniert. Eine wachsende Auslastung der Festplatte und des Netzwerks ist zu beobachten, was auf den Transfer großer Datenmengen hindeutet. Der Speicher ist nicht im Sättigungsbereich.

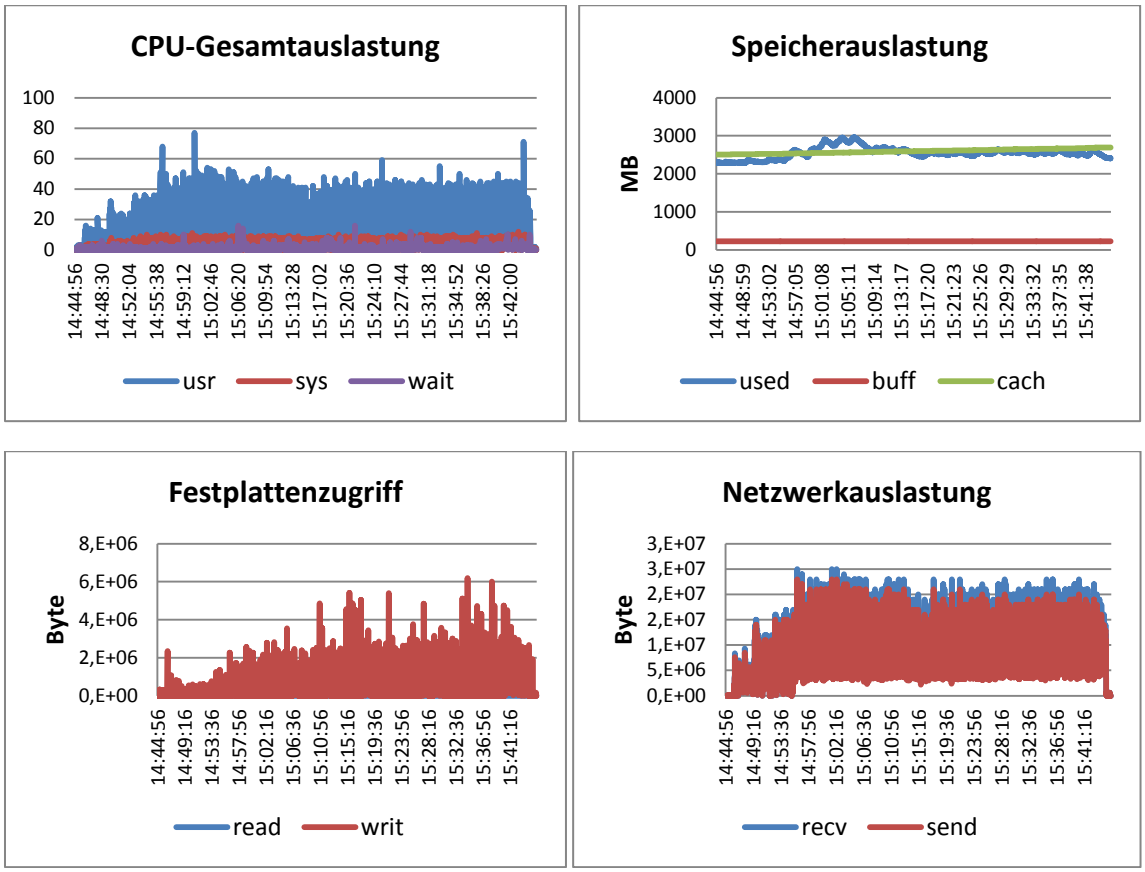

Abbildung 26 Rampen-Test #2: Gesamtressourcenverbrauch der Maschine A: Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

Abbildung 27 zeigt die zunehmende Prozessornutzung durch die einzelnen Prozesse von ODS, CEPS und CouchDB. Die ersten 5 Stufen der Treppenfunktion (ca. 270 User) sind auf dem Auslastungsmuster der CouchDB und ODS zu erkennen.

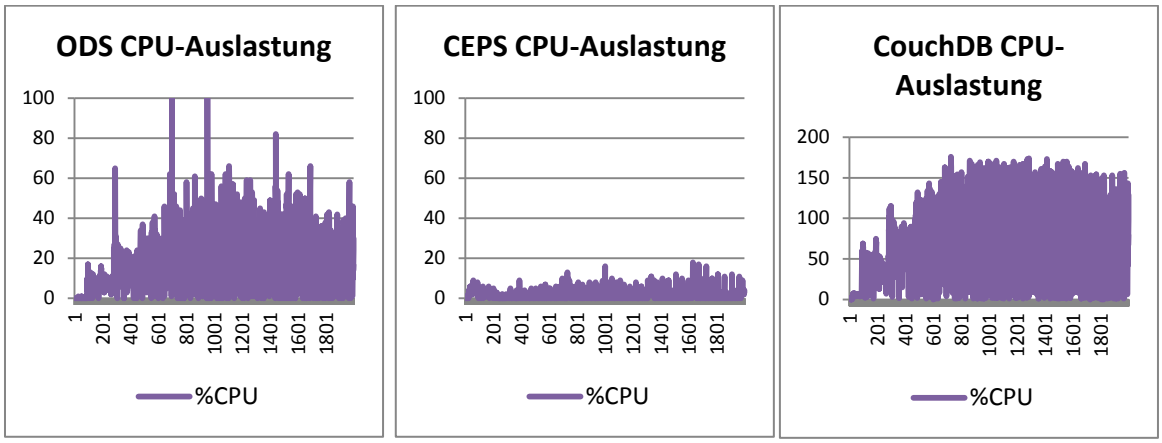

Abbildung 27 Rampen-Test #2: Prozessorauslastung durch ODS, CEPS und CouchDB. (Quelle: Eigene Darstellung).

Die beobachteten Antwortzeiten steigen deutlich an (vgl. Abbildung 28 und Abbildung 29). Im Lastbereich bis etwa 270 parallele Benutzer sind die Antwortzeiten innerhalb der Akzeptanzkriterien, anschließend überschreiten sie diese. Bereits ab 300 parallelen Benutzern steigen die Antwortzeiten über 10 Sek., ab 400 deutlich über 20 Sek. Ab 500 parallelen Benutzern stagniert das System und zeigte ein Überlastverhalten, Anfragen werden nicht mehr beantwortet und resultieren in einem Timeout. Der Test wird aus diesem Grund bei 900, und nicht wie ursprünglich geplant bei 1400 parallelen Benutzern beendet.

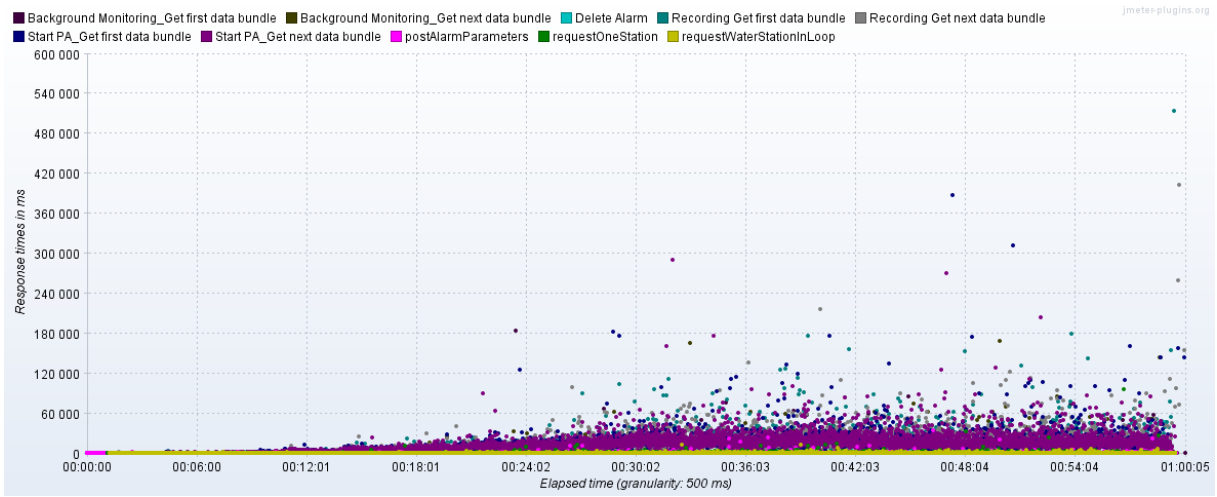

Abbildung 28 Rampen-Test #2: Antwortzeiten aller Anfragen. (Quelle: Eigene Darstellung).

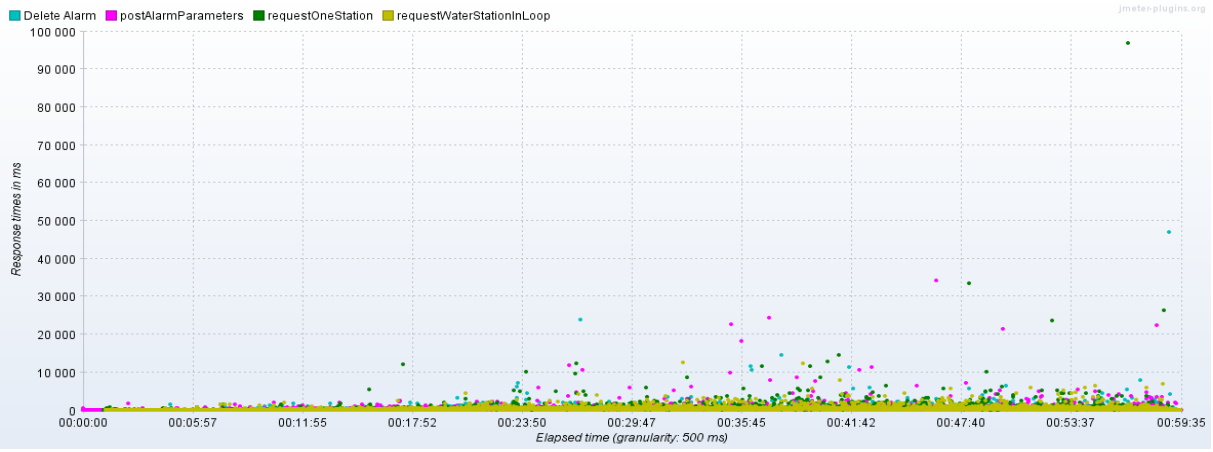

Abbildung 29 Rampen-Test #2: Antwortzeiten der leichtgewichtigen Anfragen (Einzelwert-Antwort). (Quelle: Eigene Darstellung).

Betrachtet man nur die leichtgewichtigen Anfragen, die ein einzelnes Ergebnis liefern (Abbildung 30), so bleiben die Antwortzeiten auch bis deutlich über 400 parallele Benutzer akzeptabel.

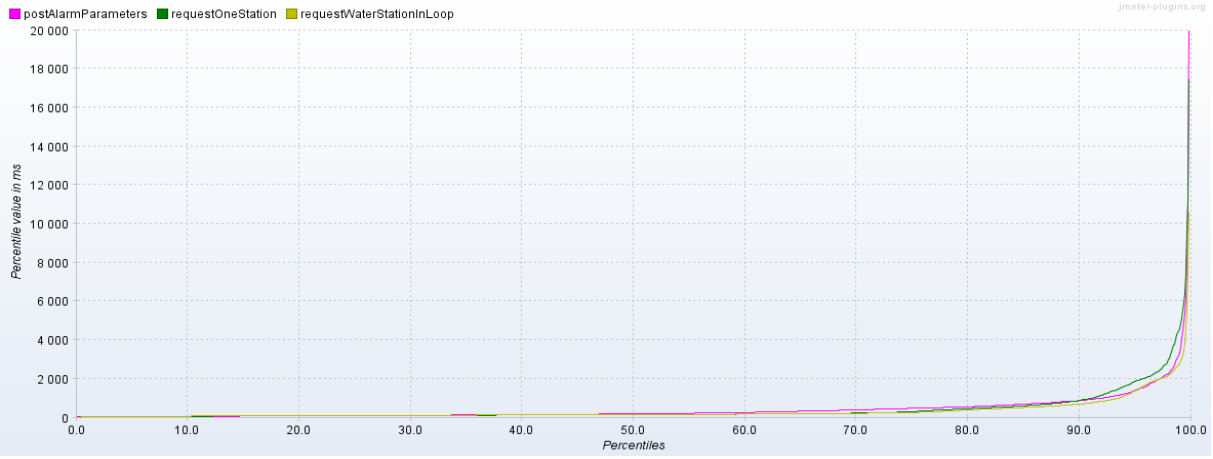

Abbildung 30 Rampen-Test #2: Streuung der Antwortzeiten der leichtgewichtigen Anfragen. (Quelle: Eigene Darstellung).

Die komplexen Anfragen hingegen übersteigen die Akzeptanzkriterien schnell (Abbildung 31).

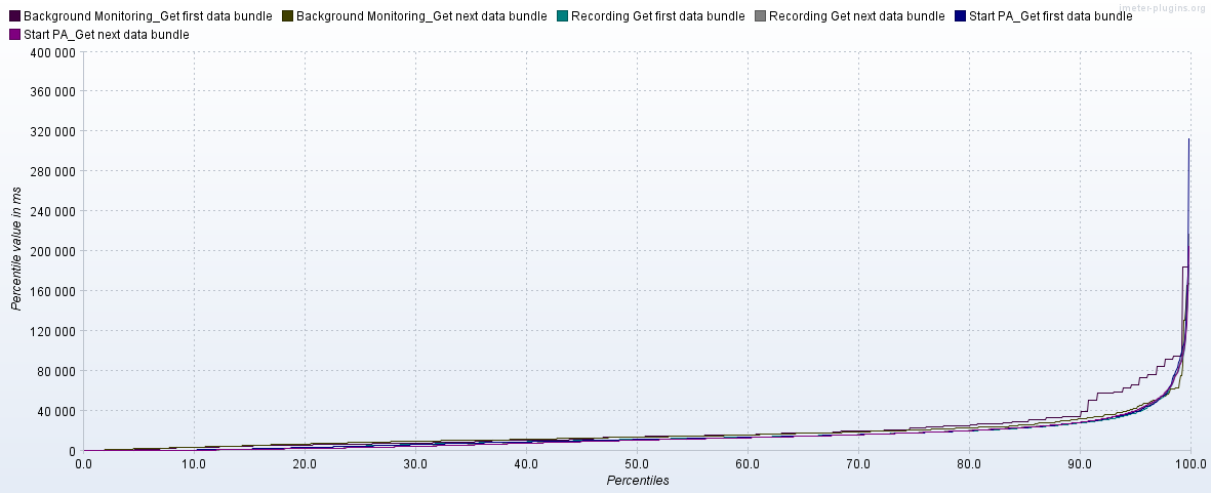

Abbildung 31 Rampen-Test #2: Streuung der Antwortzeiten der schwergewichtigen Anfragen. (Quelle: Eigene Darstellung).

Der Durchsatz stagniert bereits ab ca. 200 parallelen Benutzern und beginnt stark zu streuen (Abbildung 32).

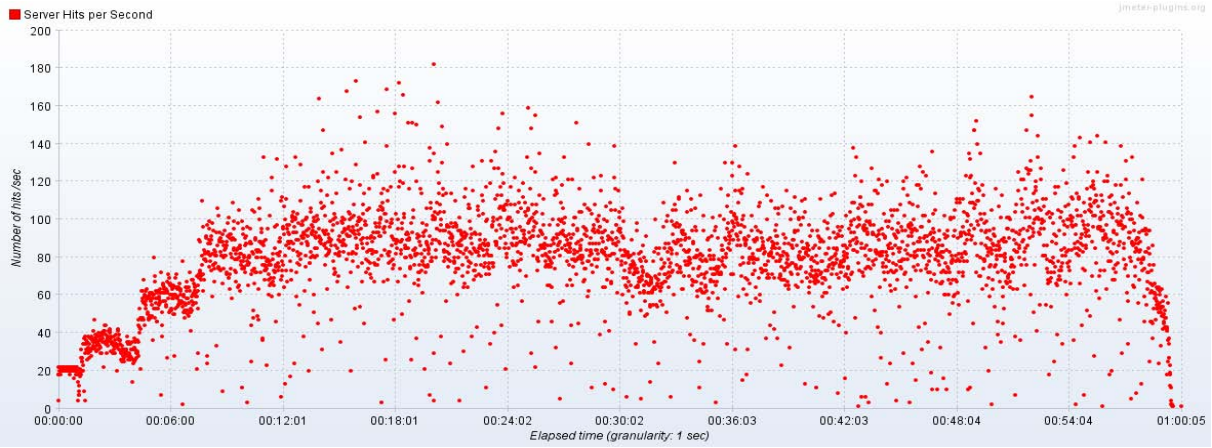

Abbildung 32 Rampen-Test #2: Durchsatz (Hits/Sec.). (Quelle: Eigene Darstellung).

4.6.3 Lasttest #1

Der Lasttest betrachtet das Verhalten der Anwendung unter einer konstanten Last über einen längeren Zeitraum hinweg. Tabelle 13 stellt die Rahmenbedingungen des Tests dar.

Tabelle 13 Lasttest #1: Randbedingungen und Zusammenfassung

Testziel	Bestimmung der Antwortzeiten unter einer konstanten Last nach einer kurzen Ramp-Up-Phase.
Testszenario	Niedriglastbedingungen
Testendekriterium	1 Stunde Testzeit
Testabbruchkriterium	5% Transaktionsabbrüche aller abgeschickten Anfragen
Testumgebung	Testumgebung #1
Gesamte Anzahl an parallelen Benutzern	PA: 140 Threads Monitoring: 14 Threads

In diesem Lastszenario wird die Lastobergrenze von 140+14 parallelen Benutzern schnell erreicht (Ramp-Up über 4 Minuten) und dann konstant über eine halbe Stunde gehalten (siehe Abbildung 33).

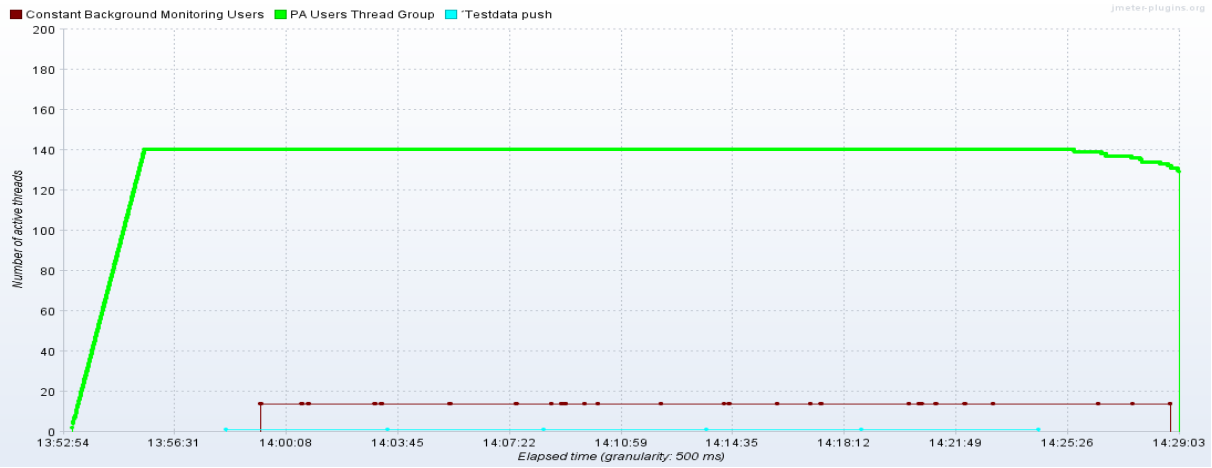

Abbildung 33 Lasttest #1: Parallel aktive Benutzern. (Quelle: Eigene Darstellung).

Die CPU-Auslastung beträgt recht konstant ca. 25% über die gesamte Testzeit (siehe Abbildung 34).

Abbildung 34 Lasttest #1: Gesamtressourcenverbrauch der Maschine A: Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

Die CPU-Auslastung durch die CouchDB unterliegt einer starken Schwankung, liegt jedoch im Durchschnitt bei ca. 70% (vgl. Abbildung 35). ODS lastet die CPU zu ca. 20% aus. CEPS benötigt bis auf einzelne Peaks von ca. 10% kaum Rechenkapazität.

Abbildung 35 Lasttest #1: Prozessorauslastung durch ODS (java#1), CEPS (java#3) und CouchDB (erl). (Quelle: Eigene Darstellung).

Der Durchsatz bleibt über die gesamte Testlaufzeit konstant (siehe Abbildung 36) und deutet auf eine gleichschnelle Bearbeitung der Anfragen hin.

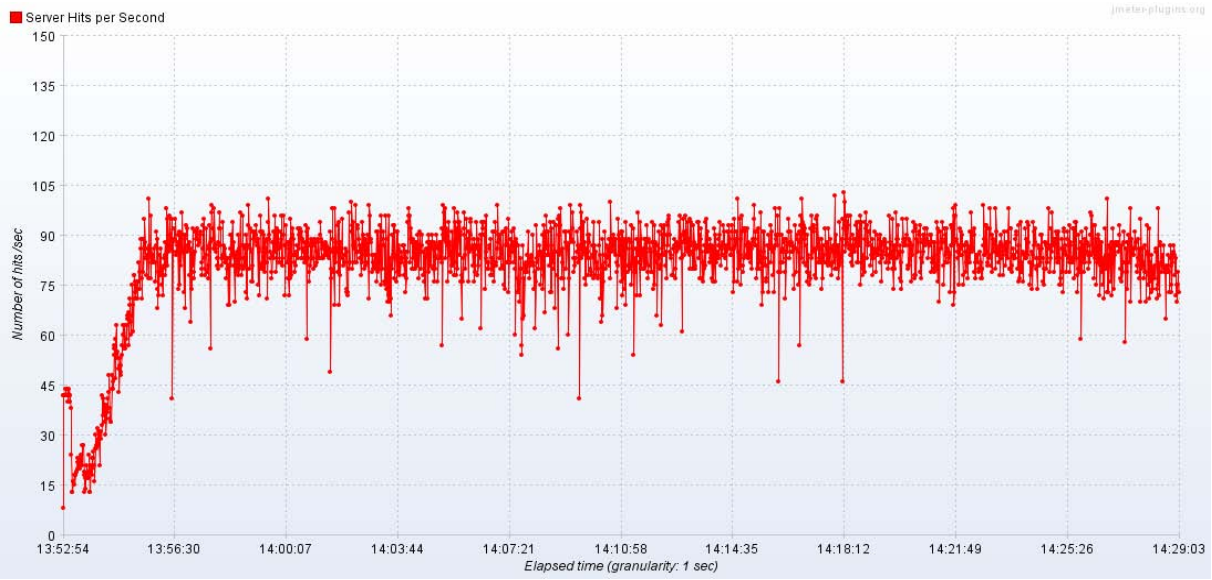

Abbildung 36 Lasttest #1: Durchsatz (Hits/Sec.). (Quelle: Eigene Darstellung).

Die beobachteten Antwortzeiten liegen im akzeptablen Bereich (siehe Abbildung 37). Die einzelnen Ausreißer können toleriert werden.

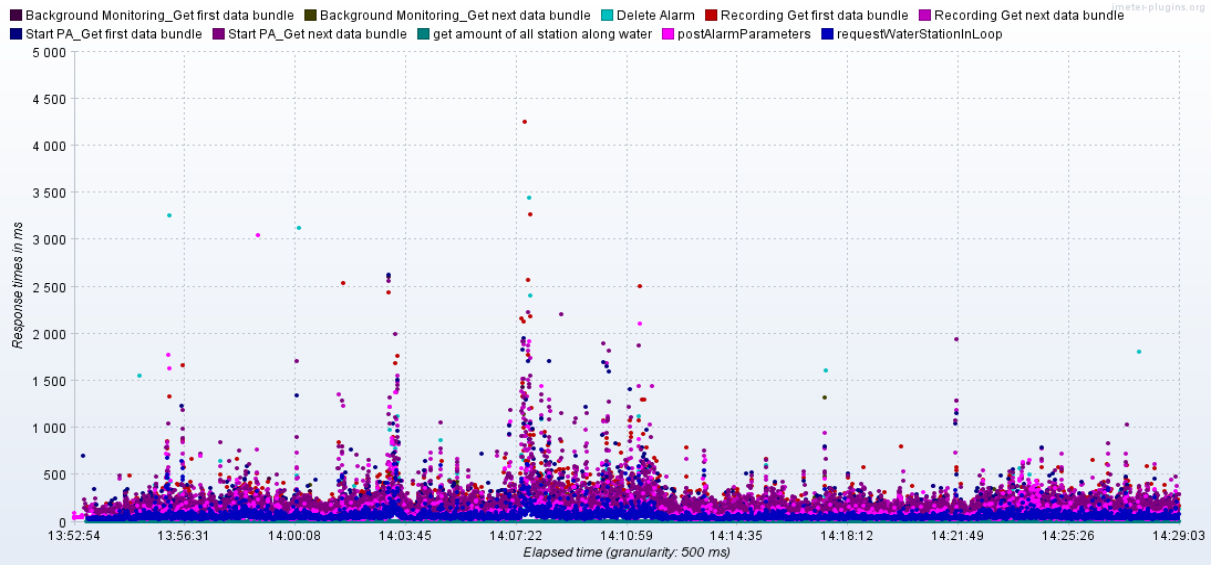

Abbildung 37 Lasttest #1: Antwortzeiten. (Quelle: Eigene Darstellung).

4.6.4 Lasttest #2

Der Lasttest betrachtet das Verhalten der Anwendung unter einer konstanten Last über einen längeren Zeitraum hinweg. In Lasttest #2 erfolgt nun eine Untersuchung der Auswirkung der Monitoring-Benutzer auf den Ressourcenverbrauch und das Zeitverhalten.

Tabelle 14 stellt die Rahmenbedingungen des Tests dar.

Tabelle 14 Lasttest #2: Randbedingungen und Zusammenfassung

Testziel	Bestimmung der Antwortzeiten unter einer konstanten Last nach einer kurzen Ramp-Up-Phase.
Testszenario	Normallastbedingungen
Testendekriterium	1 Stunde Testlaufzeit
Testabbruchkriterium	5% Transaktionsabbrüche aller abgeschickten Anfragen
Testumgebung	Testumgebung #1
Gesamte Anzahl an parallelen Benutzern	PA: 140 Threads Monitoring: 105 Threads

In diesem Lastszenario wird die Lastobergrenze von 140+105 parallelen Benutzern schnell erreicht (Ramp-Up über 3 Minuten) und dann konstant über eine ¾ Stunde gehalten (siehe Abbildung 38).

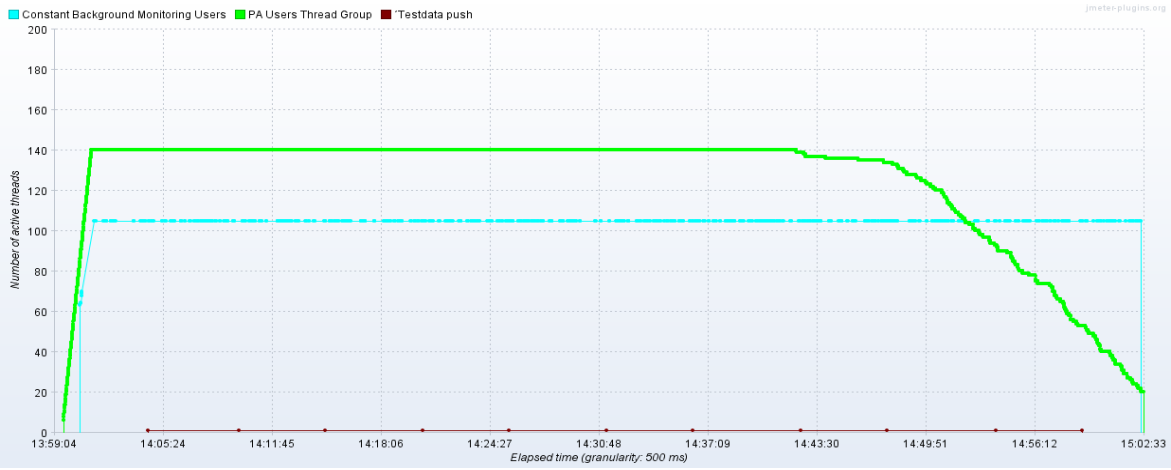

Abbildung 38 Lasttest #2: Parallel aktive Benutzern. (Quelle: Eigene Darstellung).

Die CPU-Auslastung beträgt ca. 35% über die gesamte Volllastzeit (siehe Abbildung 39). Insgesamt ist eine höhere Ressourcenauslastung zu beobachten als in Lasttest #1 (vgl. Abbildung 34).

Abbildung 39 Lasttest #2: Gesamtressourcenverbrauch der Maschine A: Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

CouchDB beansprucht die ihr zur Verfügung stehende CPU vollständig, ihre CPU-Auslastung liegt im Durchschnitt bei ca. 95% (vgl. Abbildung 40). ODS lastet die CPU zu ca. 25% aus. CEPS benötigt kaum

Rechenkapazität. Einzelne Spitzen der CPU-Nutzung sind über die Esper Engine zu erklären, die in diesen Momenten neu ankommende Daten auf Events durchsucht.

Abbildung 40 Lasttest #2: Prozessorauslastung durch ODS (java#1), CEPS (java#3) und CouchDB (erl). (Quelle: Eigene Darstellung).

Der Durchsatz bleibt über die gesamte Testlaufzeit konstant (siehe Abbildung 41), jedoch streut dieser stärker als in Lasttest #1 (vgl. Abbildung 36).

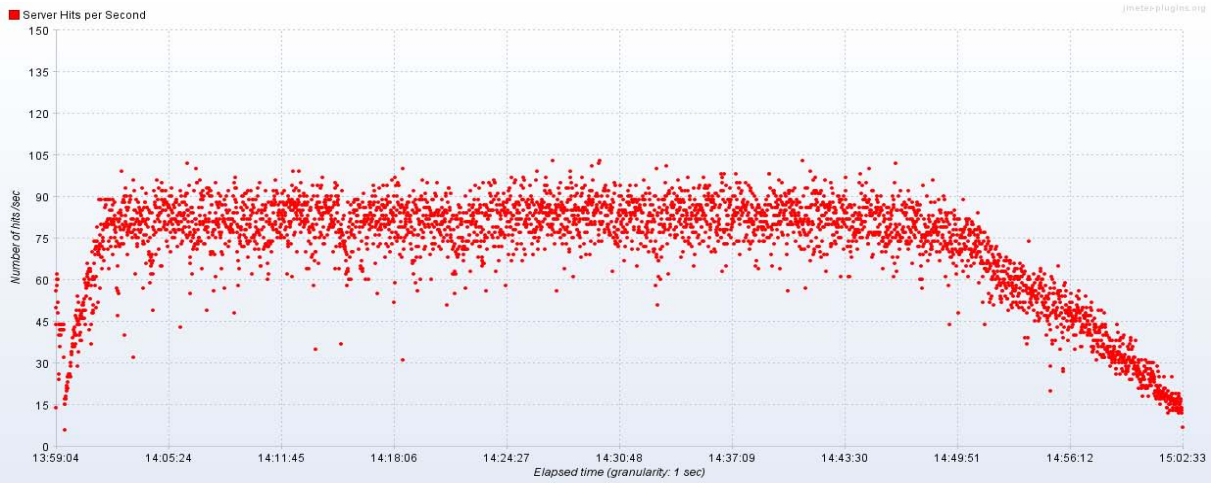

Abbildung 41 Lasttest #2: Durchsatz (Hits/Sec.). (Quelle: Eigene Darstellung).

Die beobachteten Antwortzeiten liegen noch im akzeptablen Bereich (siehe Abbildung 42), streuen jedoch ebenfalls stärker als in Lasttest #1 (vgl. Abbildung 37). Einzelne Ausreißer können toleriert werden.

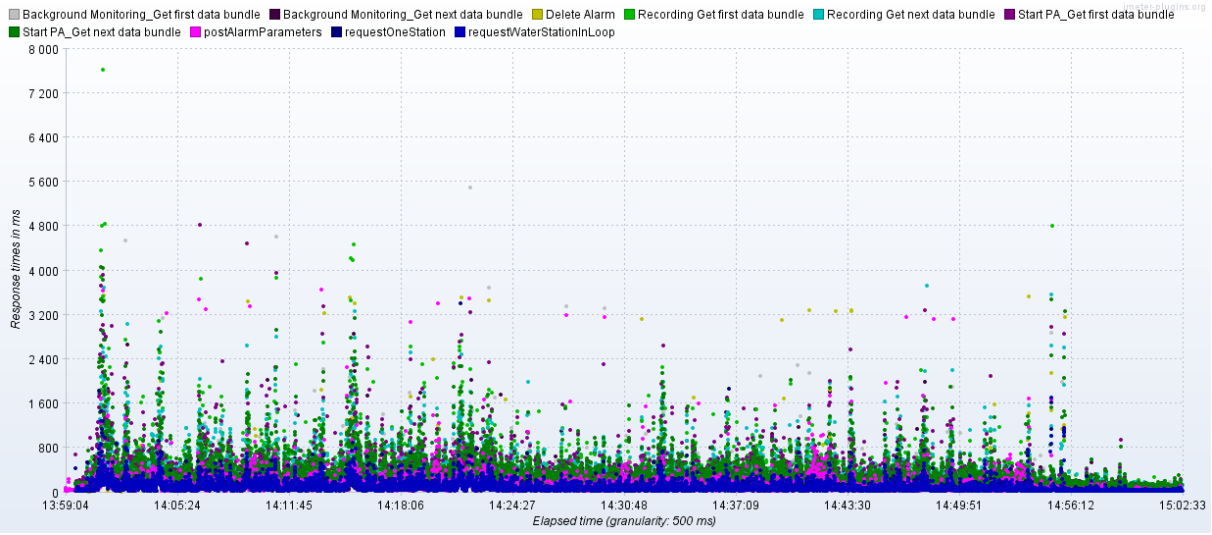

Abbildung 42 Lasttest #2: Antwortzeiten. (Quelle: Eigene Darstellung).

4.6.5 Lasttest #3

Der Lasttest betrachtet das Verhalten der Anwendung unter einer konstanten Last über einen längeren Zeitraum hinweg. In diesem dritten Lasttest liegt der Fokus auf dem Verhalten unter Hochlastbedingungen. Tabelle 15 stellt die Rahmenbedingungen des Tests dar.

Tabelle 15 Lasttest #3: Randbedingungen und Zusammenfassung.

Testziel	Bestimmung der Antwortzeiten unter einer konstanten Last nach einer kurzen Ramp-Up-Phase.
Testszenario	Hochlastbedingungen
Testendekriterium	1 Stunde Testlaufzeit
Testabbruchkriterium	5% Transaktionsabbrüche aller abgeschickten Anfragen
Testumgebung	Testumgebung #2
Gesamte Anzahl an parallelen Benutzer	PA: 200 Threads Monitoring: 150 Threads

In diesem Lastszenario wird die Lastobergrenze von 200+150 parallelen Benutzern schnell erreicht (Ramp-Up über 5 Minuten) und dann konstant über eine Stunde gehalten (siehe Abbildung 43).

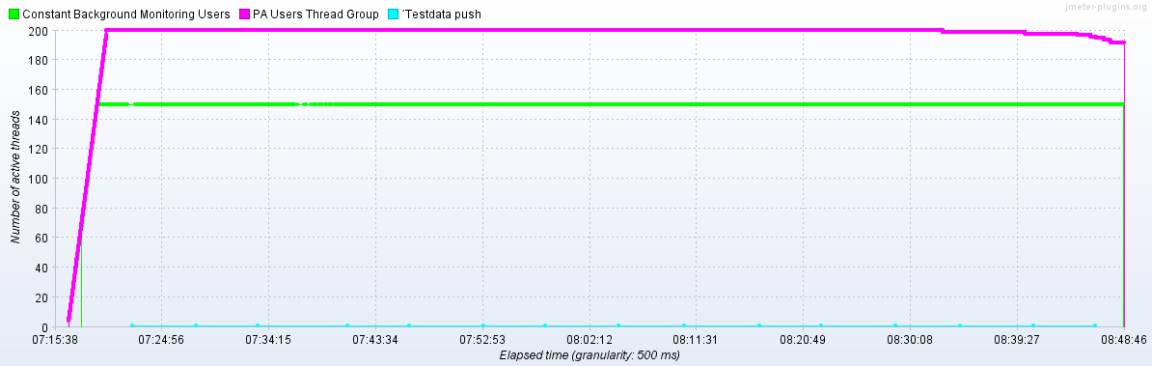

Abbildung 43 Lasttest #3: Parallel aktive Benutzern. (Quelle: Eigene Darstellung).

Abbildung 44 zeigt die zunehmende Prozessorauslastung der Maschine A. Die Gesamtauslastung der CPU beträgt ca. 35% über die gesamte Testlaufzeit. Andere Ressourcen sind auch konstant ausgelastet über die gesamte Laufzeit (siehe Abbildung 44).

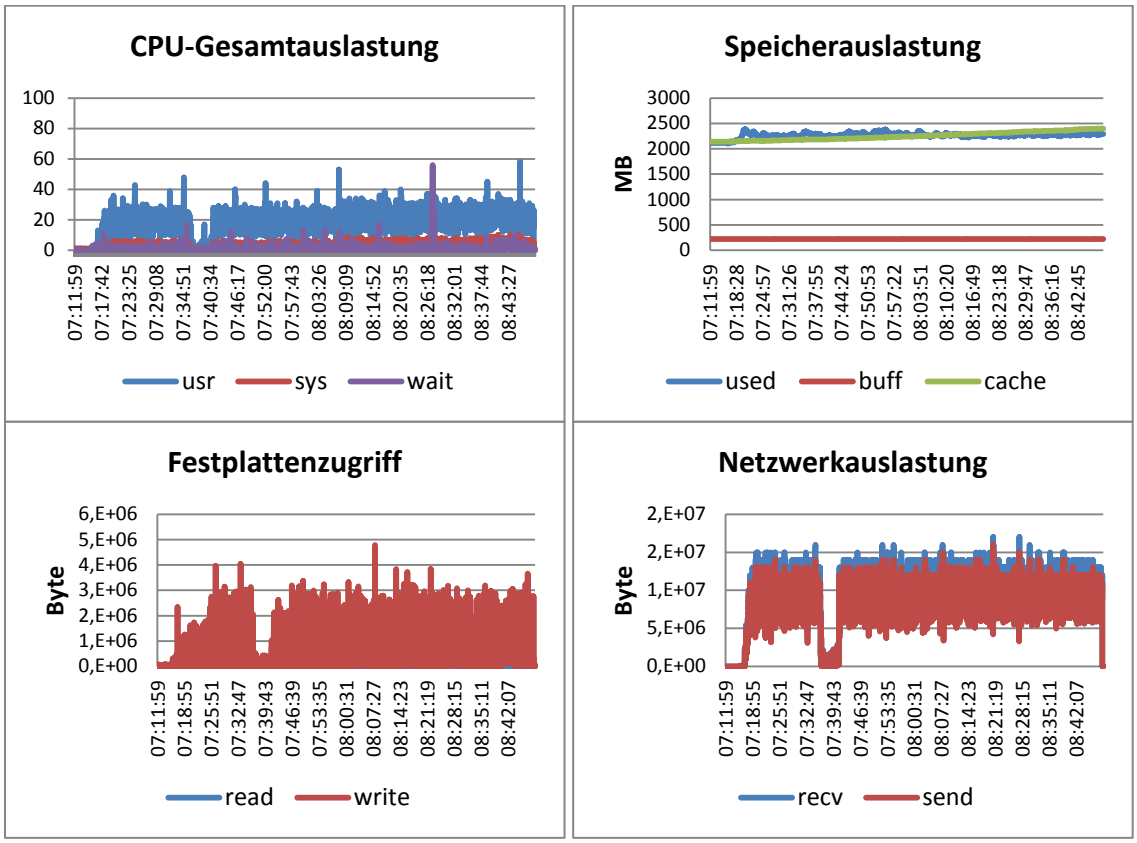

Abbildung 44 Lasttest #3: Gesamtressourcenverbrauch der Maschine A: Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

Betrachtet man die einzelnen Prozesse von ODS, CEPS und CouchDB, so fällt auf, dass die CouchDB eine CPU vollständig auslastet (siehe Abbildung 45).

Eine hohe Spitze der CPU-Auslastung durch CEPS ist durch den Lauf des expliziten Garbage Collectors zu erklären. Wie viele zusätzliche Ressourcen sein Vorgang benötigt hängt stark von den Randbedingungen ab – wurden in der Stunde seit seinem letzten Lauf viele Alarme ausgelöst, hat er viele Daten zu entfernen und benötigt viele Ressourcen. Dies ist aufgrund des laufenden Lasttests hier der Fall. Kurze Spitzen sind durch die Esper Engine zu erklären (siehe Abbildung 45).

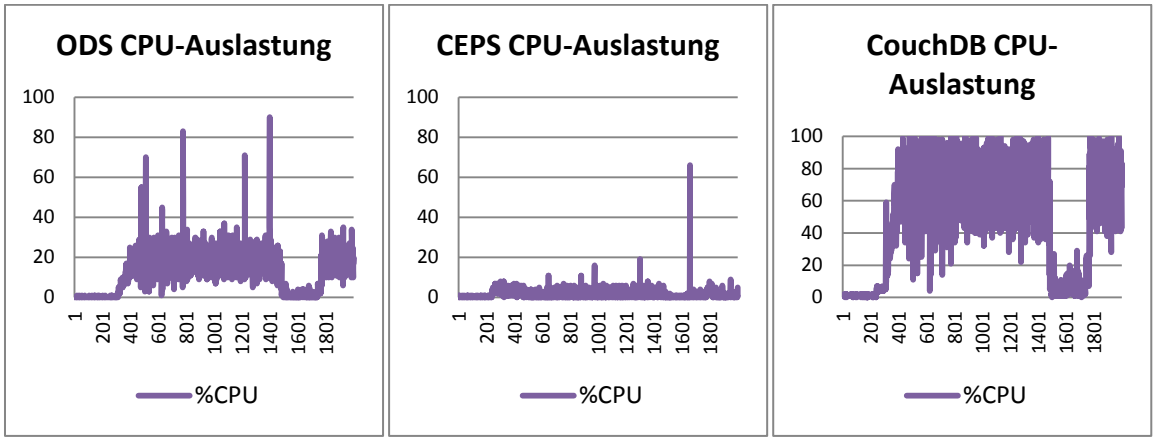

Abbildung 45 Lasttest #3: Prozessorauslastung durch ODS, CEPS und CouchDB. (Quelle: Eigene Darstellung).

Der Durchsatz bleibt über die gesamte Testlaufzeit konstant (siehe Abbildung 46). Der Tiefpunkt ist auf ein kurzes Verbindungsunterbrechen zurückzuführen.

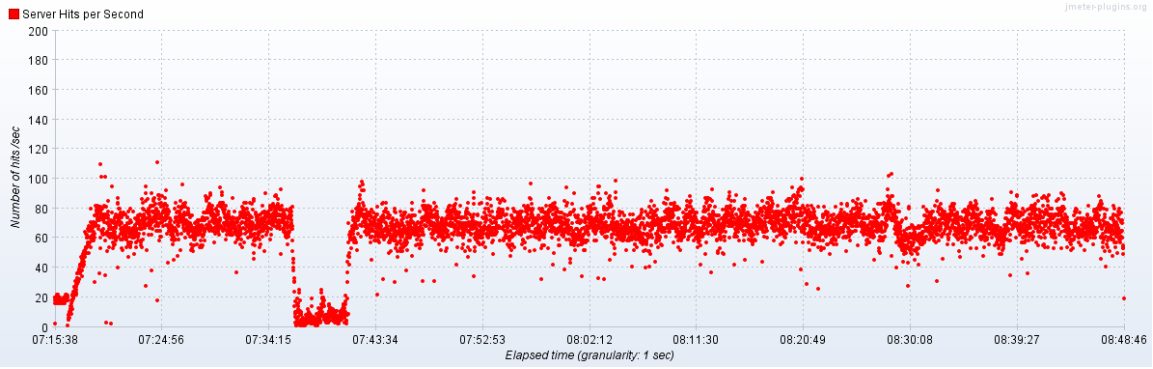

Abbildung 46 Lasttest #3: Durchsatz (Hits/Sec.). (Quelle: Eigene Darstellung).

Die beobachteten Antwortzeiten (Abbildung 47, Abbildung 48) bilden wieder zwei Klassen, die ein konstantes Verhalten über die gesamte Testlaufzeit zeigen: die leichtgewichtigen Anfragen (z. B. *requestOneStation*) liegen gut in akzeptablen Bereichen (siehe Abbildung 48), ihre einzelne Ausreißer können toleriert werden. Die schwergewichtigen Anfragen hingegen liegen am Rande der definierten Akzeptanzkriterien. Während für die Aufzeichnungsfunktionalität diese Antwortzeiten akzeptabel sind, kann es beim ersten Starten der App zu Verzögerungen führen, die nicht gewünscht sind.

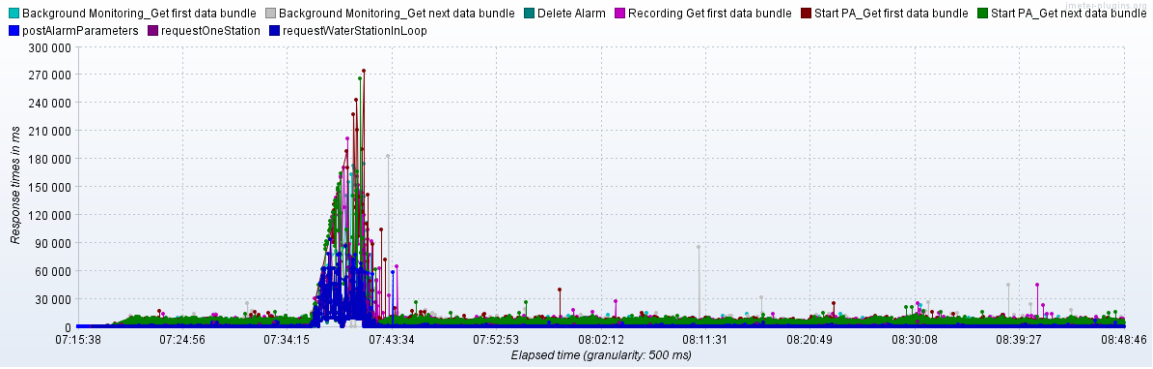

Abbildung 47 Lasttest #3: Antwortzeiten. (Quelle: Eigene Darstellung).

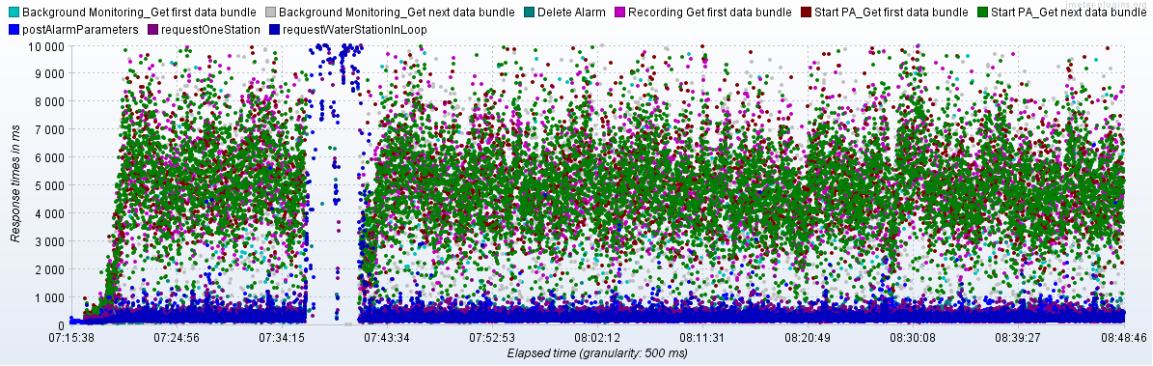

Abbildung 48 Lasttest #3: Antwortzeiten (Zoom). (Quelle: Eigene Darstellung).

4.6.6 OSM-Anbindung im laufenden Betrieb

Im Rahmen dieses Tests findet die Anbindung einer OSM-Datenquelle (*Unterfranken*) statt. Der Test betrachtet das Verhalten der Anwendung unter einer konstanten Last, während derer die neue Datenquelle

angebunden wird. Tabelle 16 stellt die Rahmenbedingungen des Tests dar.

Tabelle 16 Lasttest mit OSM-Anbindung: Randbedingungen und Zusammenfassung.

Testziel	Reaktion des Zeit- und Ressourcenverbrauchsverhalten des Systems auf die Anbindung einer neuen Datenquelle am Bespiel der Anbindung der OSM.
Testszenario	Normallastbedingungen
Testendekriterium	1 Stunde Testlaufzeit
Testabbruchkriterium	5% Transaktionsabbrüche aller abgeschickten Anfragen
Testumgebung	Testumgebung #1
Gesamte Anzahl an parallelen Benutzern	PA: 100 Threads Monitoring: 75 Threads

In diesem Lastszenario wird die Lastobergrenze von 100+75 parallelen Benutzern schnell erreicht (Ramp-Up über 3 Minuten) und dann konstant über eine Stunde gehalten (siehe Abbildung 49). Gleichzeitig findet die Anbindung der OSM-Datenquelle statt.

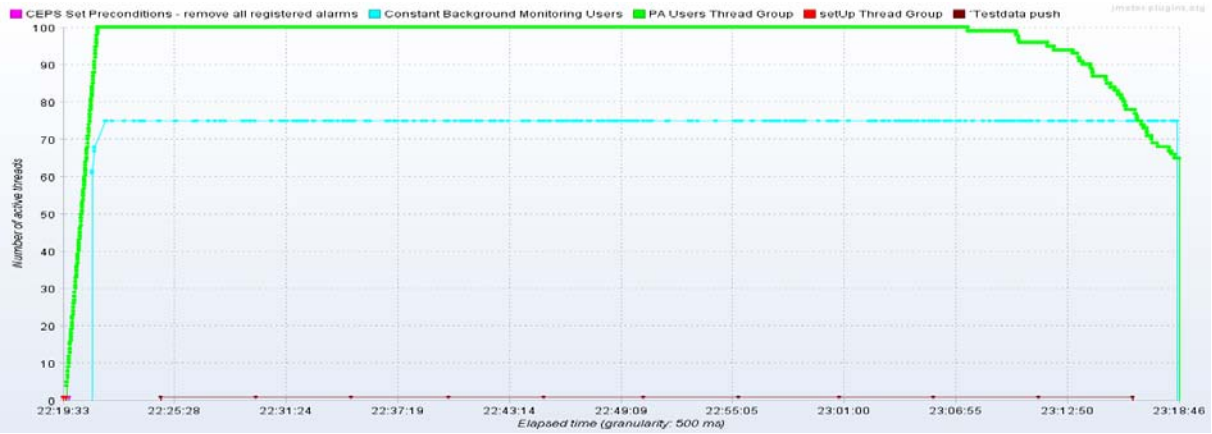

Abbildung 49 Lasttest mit OSM-Anbindung: parallel aktive Benutzern. (Quelle: Eigene Darstellung).

Die Anbindung der OSM-Daten (Ausschnitt „*Unterfranken*“) dauert ca. 25 Min (von 22:20 bis 22:35) und beansprucht zusätzliche Ressourcen (die CPU-Auslastung steigt, ebenso die Anzahl der gelesenen und geschriebenen Bytes, siehe Abbildung 50).

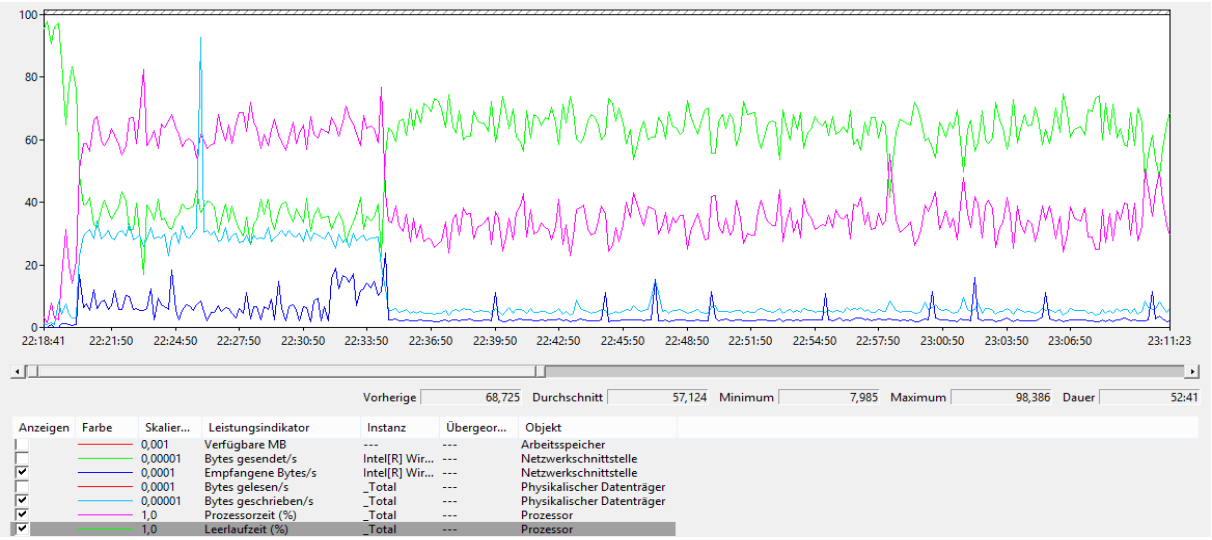

Abbildung 50 Lasttest mit OSM-Anbindung: Gesamtressourcenverbrauch der Maschine A: Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

Die CPU-Auslastung durch die CouchDB steigt in der Zeit der Anbindung auf 200% an (siehe Abbildung 51, Couch DB CPU-Auslastung auf Faktor 0,1 skaliert).

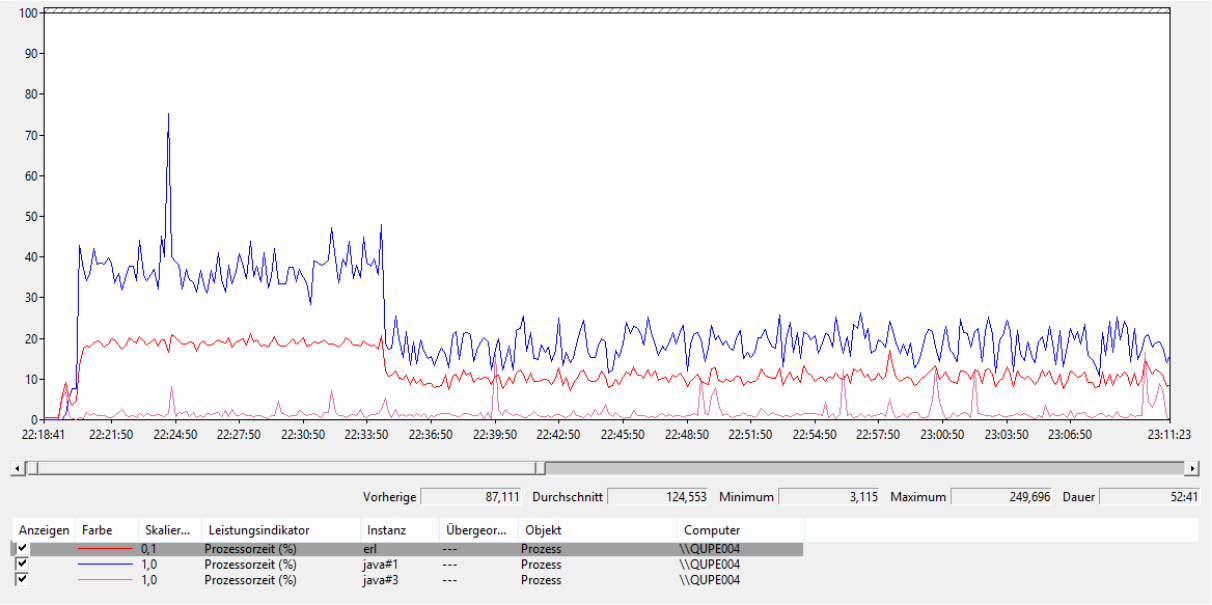

Abbildung 51 Lasttest mit OSM-Anbindung: Prozessorauslastung durch ODS (java#1), CEPS (java#3) und CouchDB (erl). (Quelle: Eigene Darstellung).

Über die gesamte Zeit des Anbindungsvorgangs werden die Daten auf die Platte geschrieben, was auch in Abbildung 52 zu beobachten ist.

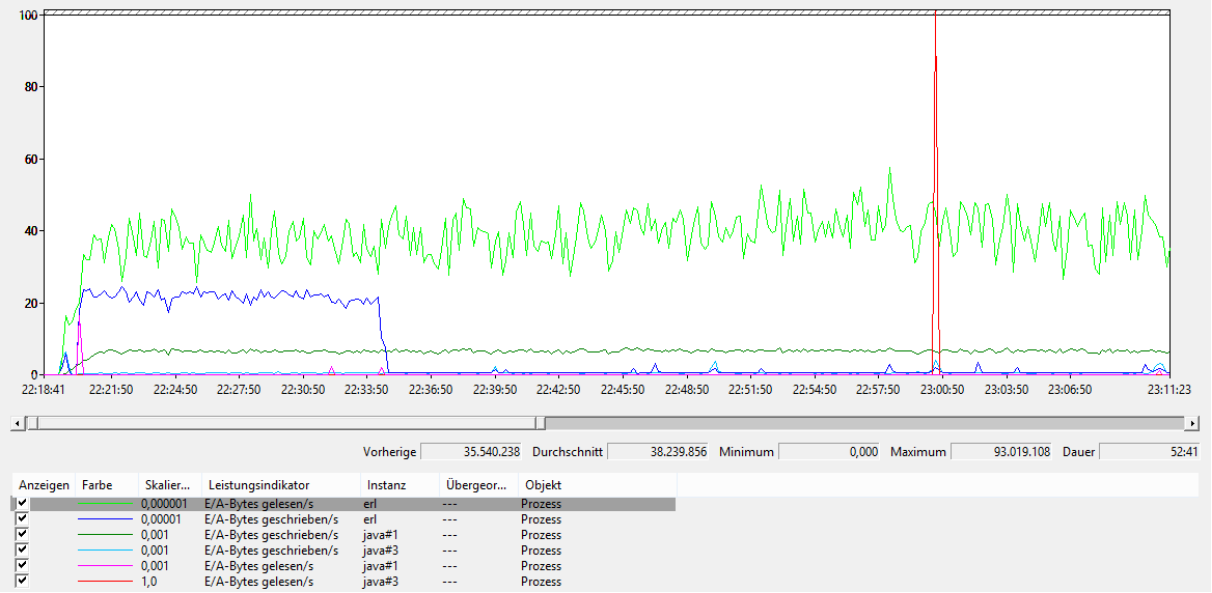

Abbildung 52 Lasttest mit OSM-Anbindung: E/A-Vorgänge auf der Festplatte durch ODS (java#1), CEPS (java#3) und CouchDB (erl). (Quelle: Eigene Darstellung).

Der Durchsatz bleibt jedoch über die gesamte Testlaufzeit konstant (siehe Abbildung 53) und deutet auf eine gleichschnelle Bearbeitung der Anfragen hin. Dies zeigt sich auch konsistent in den beobachteten Antwortzeiten (siehe Abbildung 54).

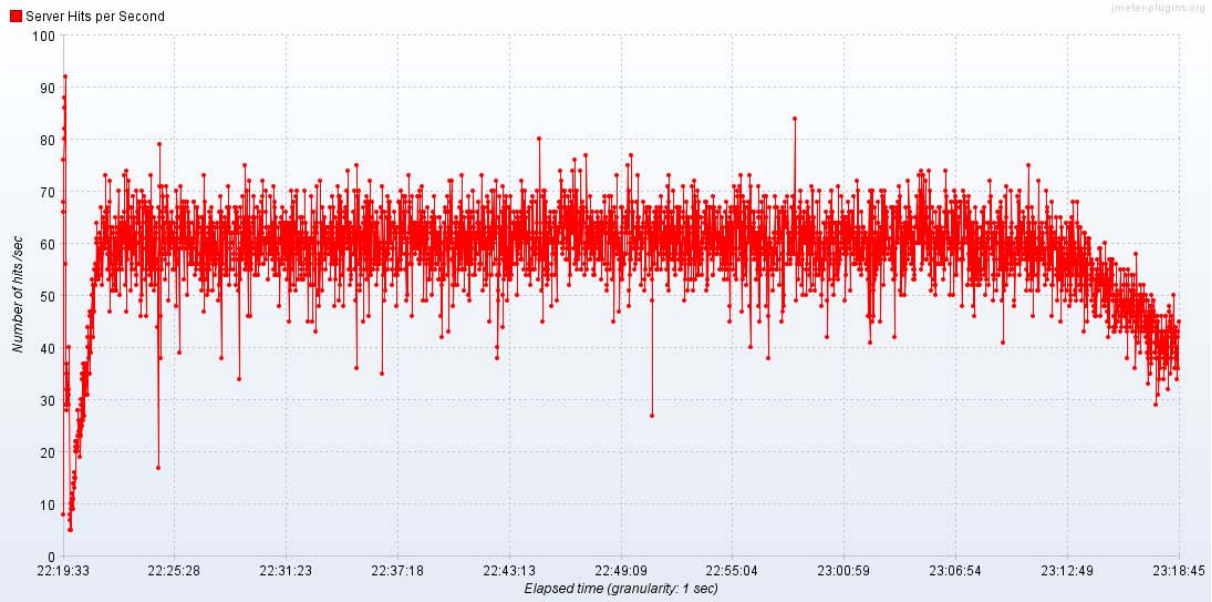

Abbildung 53 Lasttest mit OSM-Anbindung: Durchsatz (Hits/Sec.). (Quelle: Eigene Darstellung).

Die Antwortzeiten liegen in den gewünschten Bereichen, eine leicht angestiegene Anzahl an Ausreißern ist beobachtbar, kann jedoch toleriert werden (siehe Abbildung 54).

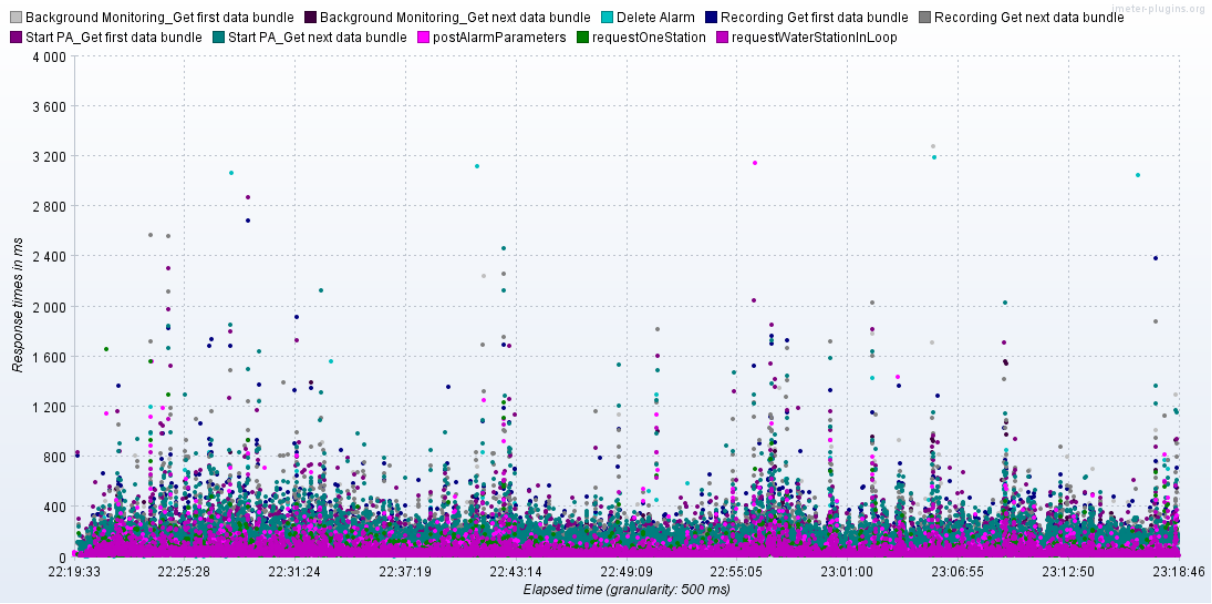

Abbildung 54 Lasttest mit OSM-Anbindung: Antwortzeiten. (Quelle: Eigene Darstellung).

4.6.7 Lasttest mit OSM-Zusatzlast

Im Rahmen dieses Tests wird die definierte Lastmenge um weitere zusätzliche Anfragen an ODS auf die OSM-Datenbestände ergänzt. Der Test betrachtet das Verhalten der Anwendung unter einer solchen Lastzusammensetzung.

Tabelle 17 stellt die Rahmenbedingungen des Tests dar.

Tabelle 17 Lasttest mit OSM-Zusatzlast: Randbedingungen und Zusammenfassung.

Testziel	Reaktion des Zeit- und Ressourcenverbrauchsverhaltens des Systems auf zusätzliche Last auf ODS (am Bespiel der OSM-Datenabfrage).
Testszenario	Normallastbedingungen
Testendekriterium	1 Stunde Testlaufzeit
Testabbruchkriterium	5% Transaktionsabbrüche aller abgeschickten Anfragen
Testumgebung	Testumgebung #2
Gesamte Anzahl an parallelen Benutzern	PA: 100 Threads Monitoring: 75 Threads OSM: 100 Threads

In diesem Lastszenario wird die Lastobergrenze von insgesamt 375 parallelen Benutzern schnell erreicht (Ramp-Up über 3 Minuten). Die OSM-Last (100 Threads) wird ca. 30 Min. konstant gehalten und danach schnell abgebaut (siehe Abbildung 55).

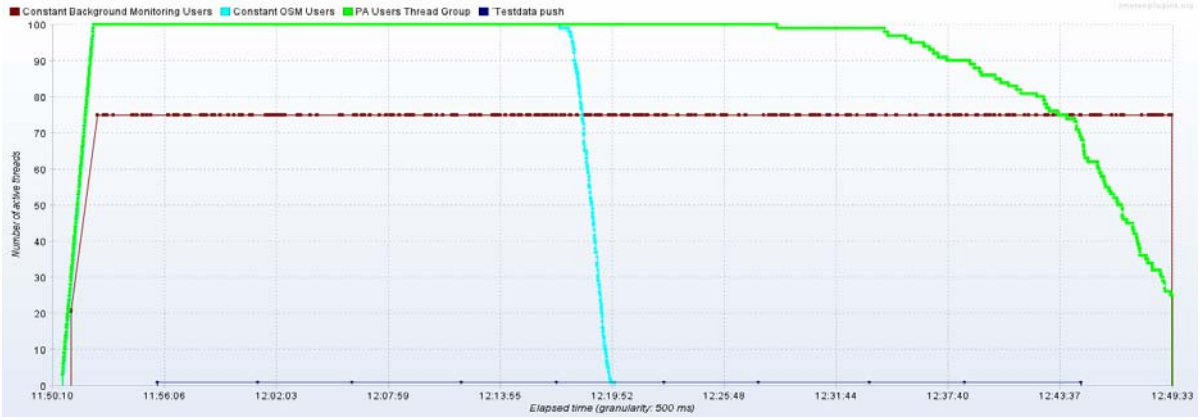

Abbildung 55 Lasttest mit OSM-Zusatzlast: Rampe an parallel aktiven Benutzern. (Quelle: Eigene Darstellung).

Abbildung 56 zeigt die Prozessorauslastung der Maschine A. Die zusätzlichen 100 OSM-Benutzer wirken sich auf die Gesamtauslastung des Systems stark aus: die CPU- und Netzwerk-Auslastung nehmen zu. Nachdem die OSM-Benutzer abgebaut sind sinkt die Auslastung der Ressourcen deutlich.

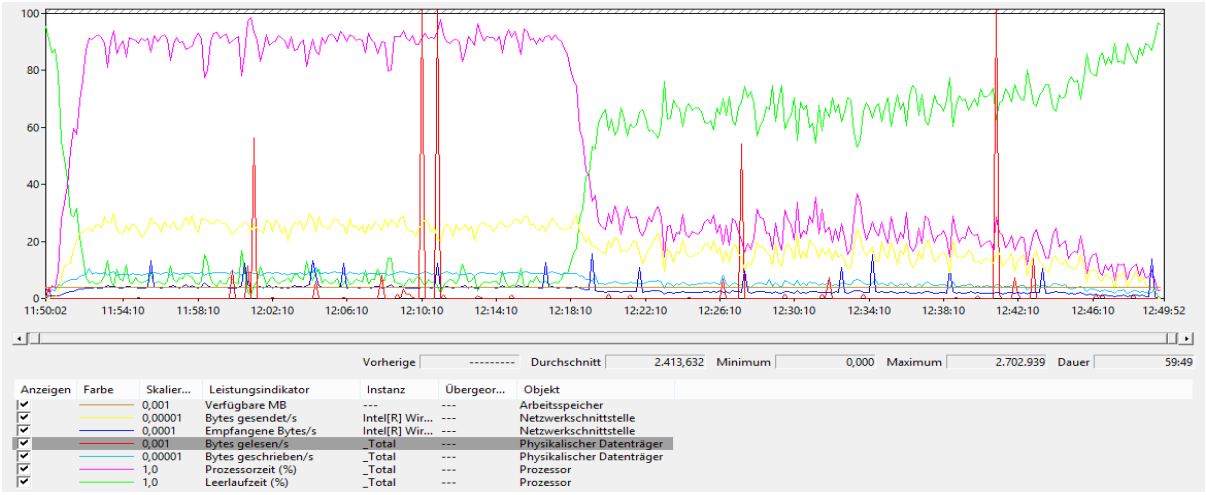

Abbildung 56 Lasttest mit OSM-Zusatzlast: Gesamtressourcenverbrauch der Maschine A: Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

Betrachtet man die einzelnen Prozesse von ODS, CEPS und CouchDB, so fällt auf, dass die CouchDB in

den ersten 30 Minuten stark ausgelastet ist (siehe Abbildung 57). ODS wird ebenfalls durch die zusätzlichen Requests deutlich stärker ausgelastet, als in dem Fall, in denen er nur die Benutzer der PegelAlarm-App bedienen muss. Entfällt diese zusätzliche Last, reduziert sich die Ressourcennutzung auf die schon beobachteten Werte (vgl. Abbildung 35). Zusätzliche ODS-Last wirkt, wie erwartet, nicht auf die Ressourcennutzung seitens CEPS, jedoch auf die Antwortzeiten der beiden Services (siehe Abbildung 58).

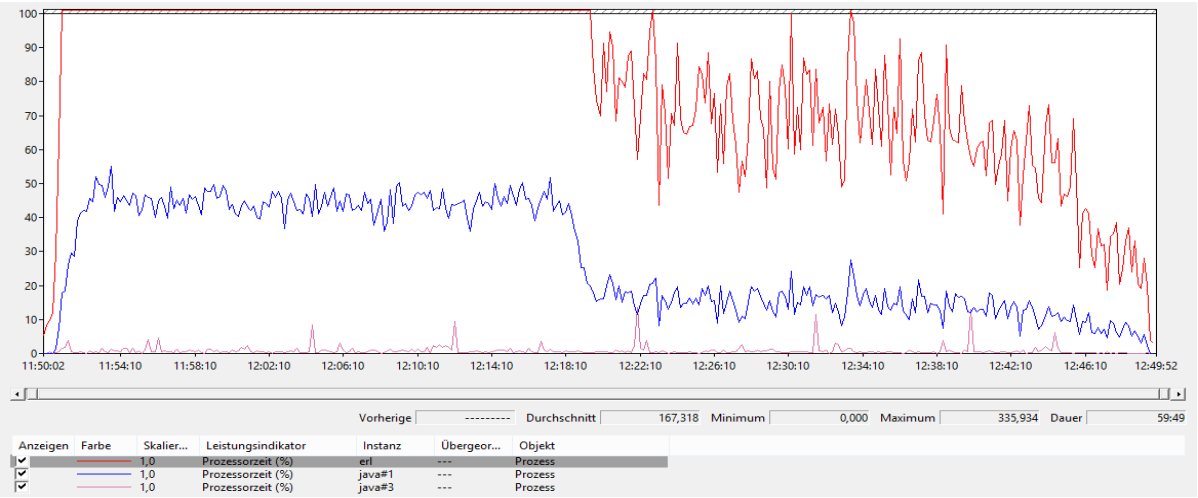

Abbildung 57 Lasttest mit OSM-Zusatzlast: Prozessorauslastung durch ODS (java#1), CEPS (java#3) und CouchDB (erl). (Quelle: Eigene Darstellung).

Die Antwortzeiten aller Requests (inkl. OSM-Anfragen, siehe Abbildung 58) bleiben innerhalb der Akzeptanzkriterien, jedoch sind sie unter Zusatzlastbedingungen deutlich höher (siehe zum Vergleich Abbildung 54). Mit dem Abbau der Zusatzlast sinken sie auf die erwarteten Werte (wie bereits in vorherigen Tests beobachtet, vgl. Abbildung 37).

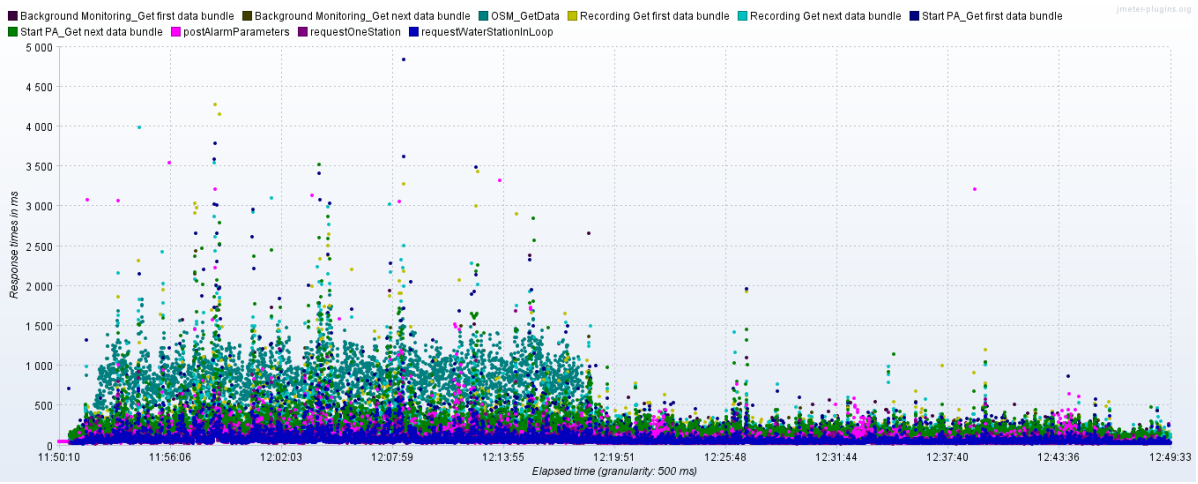

Abbildung 58 Lasttest mit OSM-Zusatzlast: Antwortzeiten (inkl. OSM-Anfragen). (Quelle: Eigene Darstellung).

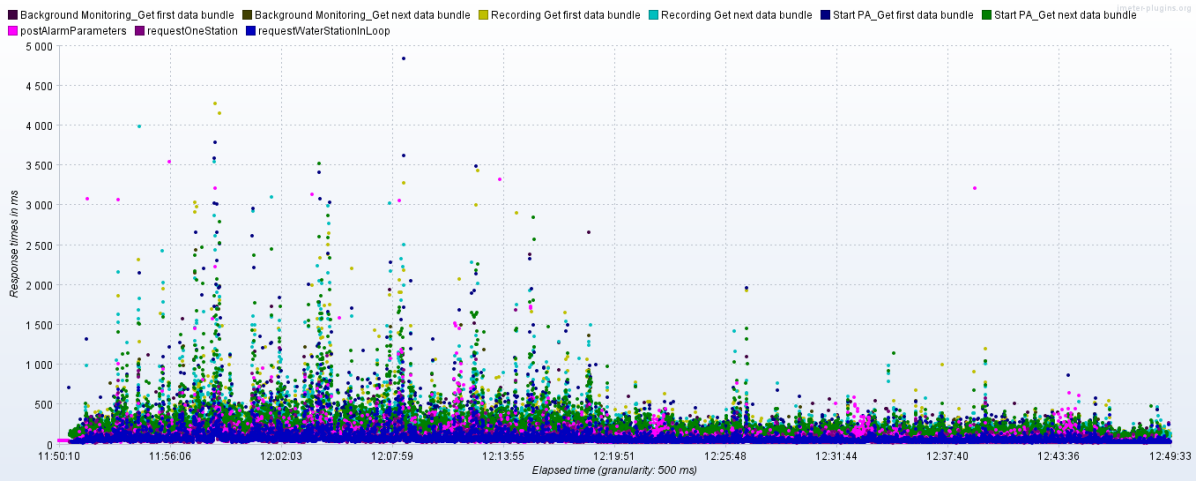

Abbildung 59 Lasttest mit OSM-Zusatzlast: Antwortzeiten (ohne OSM-Anfragen). (Quelle: Eigene Darstellung).

Der Durchsatz streut viel stärker unter Zusatzlastbedingungen (siehe Abbildung 60).

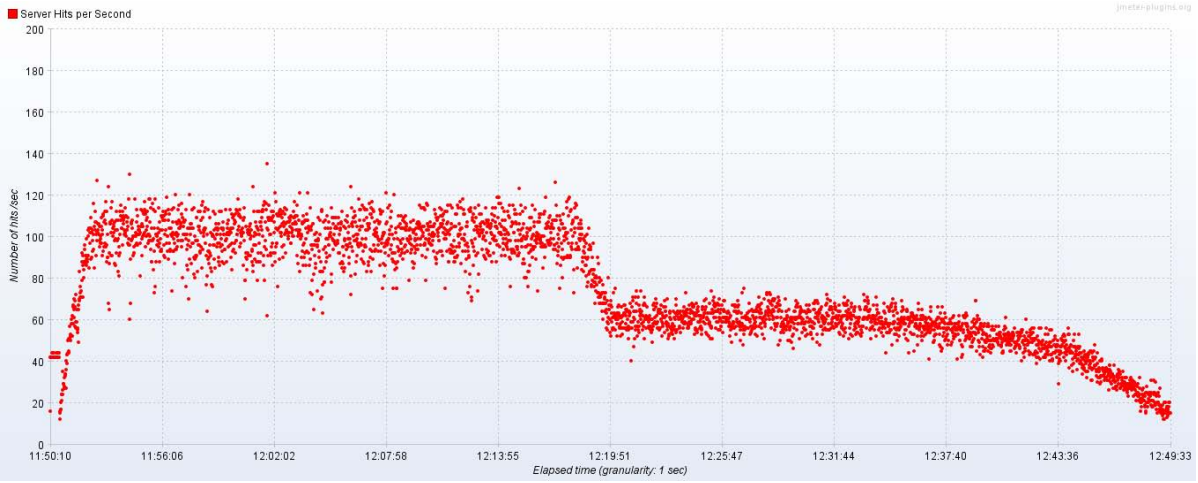

Abbildung 60 Lasttest mit OSM-Zusatzlast: Durchsatz (Hits/Sec.). (Quelle: Eigene Darstellung).

4.6.8 Separierung von ODS und CEPS auf 2 Maschinen

Im Rahmen dieses Tests werden ODS und CEPS separat auf 2 verschiedenen Maschinen deployed. Der Test betrachtet, wie sich das Verhalten der Anwendung unter solchen Deployment-Bedingungen ändert. Tabelle 18 stellt die Rahmenbedingungen des Tests dar.

Tabelle 18 Lasttest mit Separierung ODS und CEPS auf 2 Maschinen: Randbedingungen und Zusammenfassung.

Testziel	Beobachtung des Verhaltensänderung der SUT unter einem verteilten Deployment
Testszenario	Normallastbedingungen
Testendekriterium	1 Stunde Testlaufzeit
Testabbruchkriterium	5% Transaktionsabbrüche aller abgeschickten Anfragen
Testumgebung	Testumgebung #1, CEPS und ODS auf separaten Maschinen (A und A*)
Gesamte Anzahl an parallelen Benutzern	PA: 100 Threads Monitoring: 75 Threads

In diesem Lastszenario wird die Lastobergrenze von 175 parallelen Benutzern schnell erreicht (Ramp-Up über 3 Minuten, siehe Abbildung 61) und dann konstant über eine Stunde gehalten.

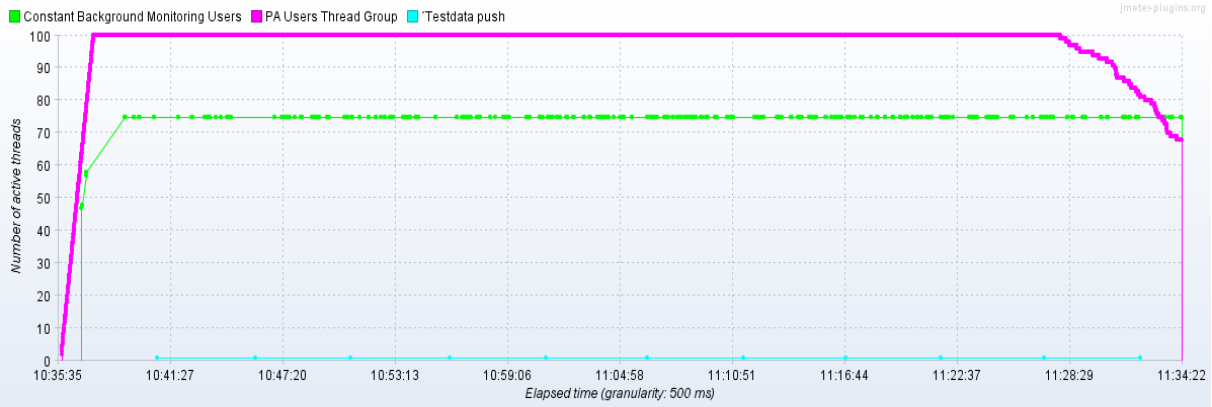

Abbildung 61 Lasttest mit Separierung ODS und CEPS auf 2 Maschinen: parallel aktive Benutzer. (Quelle: Eigene Darstellung).

Abbildung 62 zeigt die Ressourcennutzung der Maschine, die ODS hostet. Die Gesamtprozessorauslastung auf dieser Maschine beträgt ca. 25%.

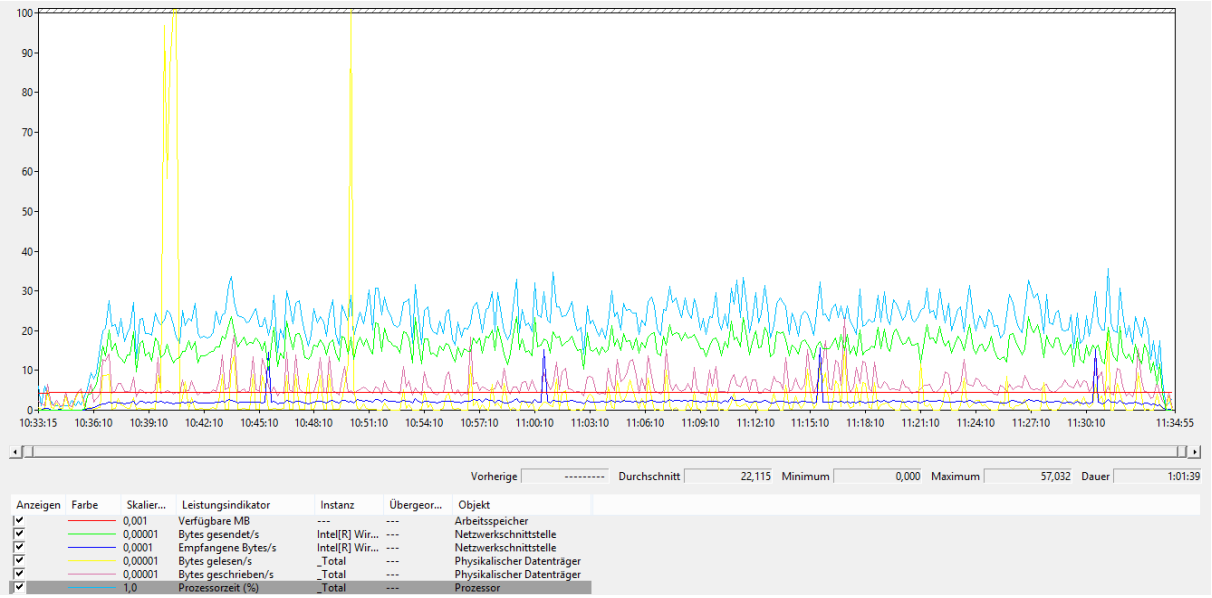

Abbildung 62 Lasttest mit Separierung ODS und CEPS auf 2 Maschinen: Gesamtressourcenverbrauch der Maschine A (ODS): Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

Abbildung 63 zeigt die Ressourcennutzung der Maschine, die CEPS hostet. Die CEPS-Maschine ist nicht ausgelastet, die Gesamtprozessorauslastung beträgt ca. 7%. Die Spitzen der CPU-Auslastung stimmen mit den Ankunftszeiten neuer Daten überein, die mit der Esper Engine verarbeitet werden.

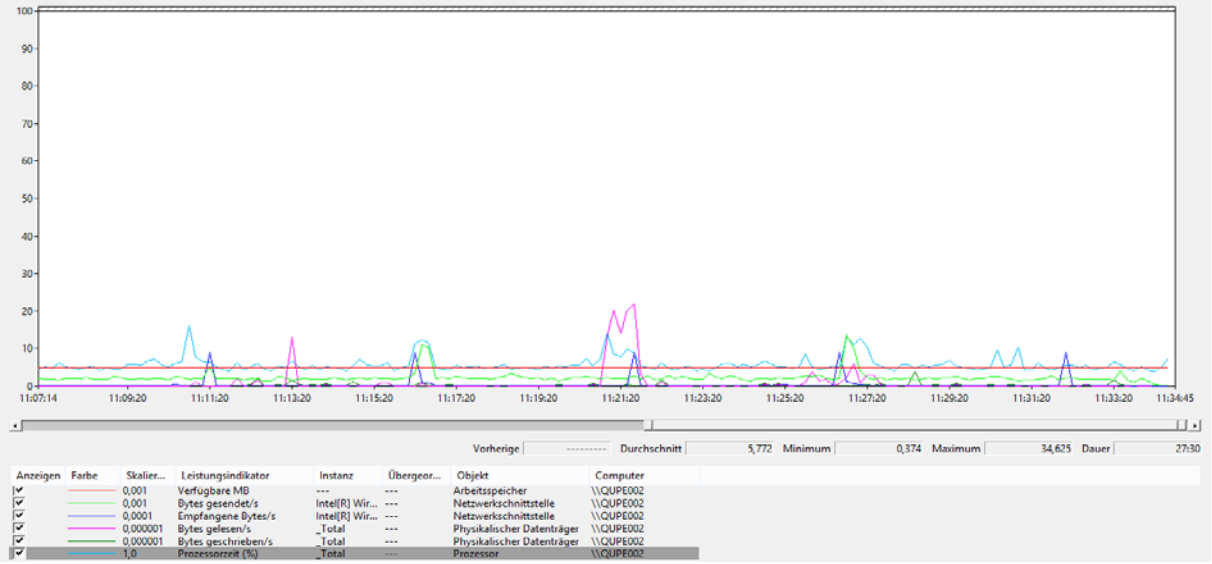

Abbildung 63 Lasttest mit Separierung ODS und CEPS auf 2 Maschinen: Gesamtressourcenverbrauch der Maschine A* (CEPS): Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

Betrachtet man die einzelnen Prozesse von ODS und CouchDB auf Maschine A, so fällt auf, dass die CouchDB deutlich mehr die CPU auslastet als ODS (siehe Abbildung 64).

Abbildung 64 Lasttest mit Separierung ODS und CEPS auf 2 Maschinen: Prozessorauslastung durch ODS (java#1) und CouchDB (erl). (Quelle: Eigene Darstellung).

Die Auslastung der CPU von Maschine A* durch die einzelnen Prozesse (CEPS und CouchDB) ist in der Abbildung 65 zu sehen. Sie spiegelt die CPU-Auslastung durch das Ankommen neuer Daten (sowohl von ODS als auch die Testdaten, die das Feuern der Alarme provozieren) wider, liegt jedoch bei weitem nicht im Sättigungsbereich.

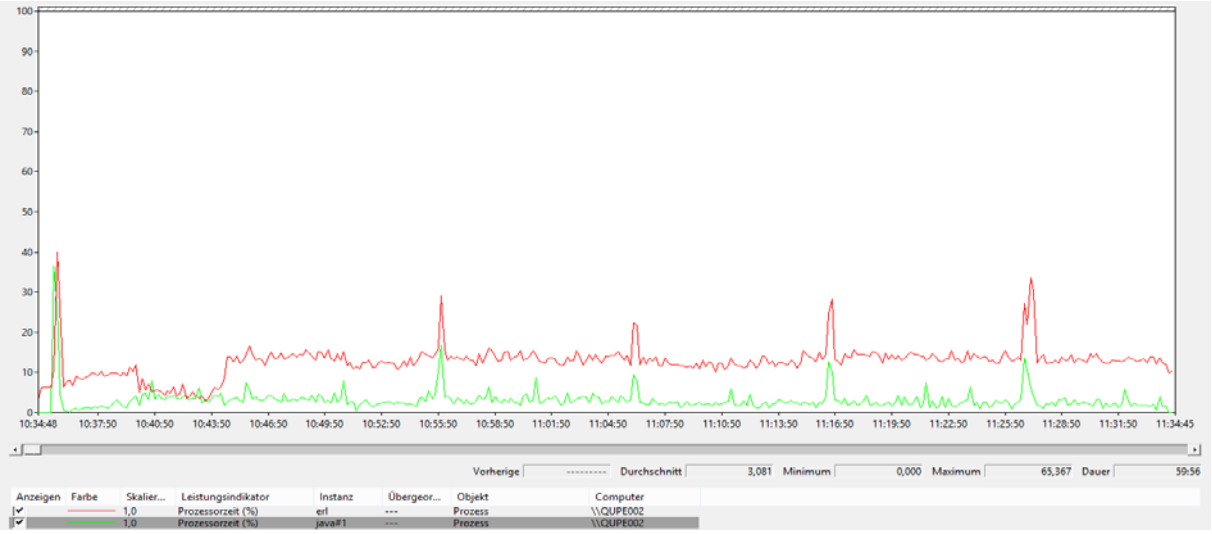

Abbildung 65 Lasttest mit Separierung ODS und CEPS auf 2 Maschinen: Prozessorauslastung durch CEPS (java#1) und CouchDB (erl). (Quelle: Eigene Darstellung).

Beide Services liefern in dieser Konstellation deutlich bessere Antwortzeiten (siehe Abbildung 66 und Abbildung 67).

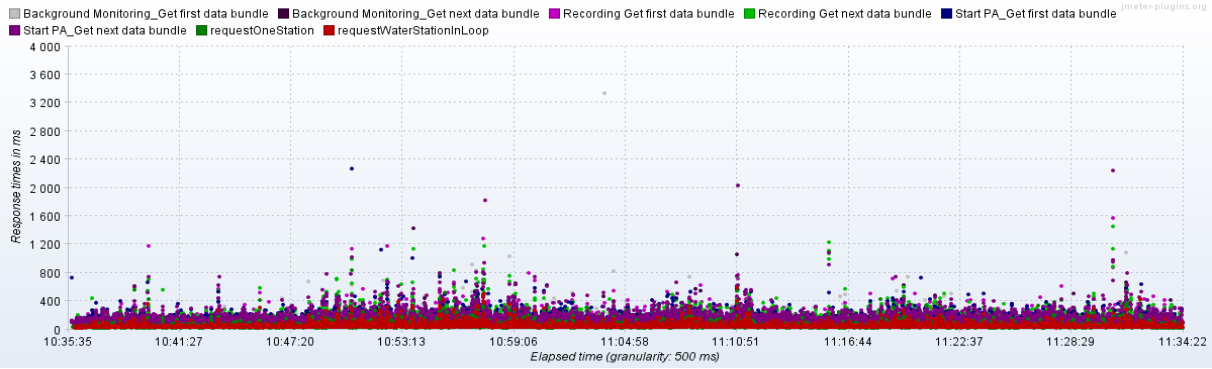

Abbildung 66 Lasttest mit Separierung ODS und CEPS auf 2 Maschinen: ODS-Antwortzeiten. (Quelle: Eigene Darstellung).

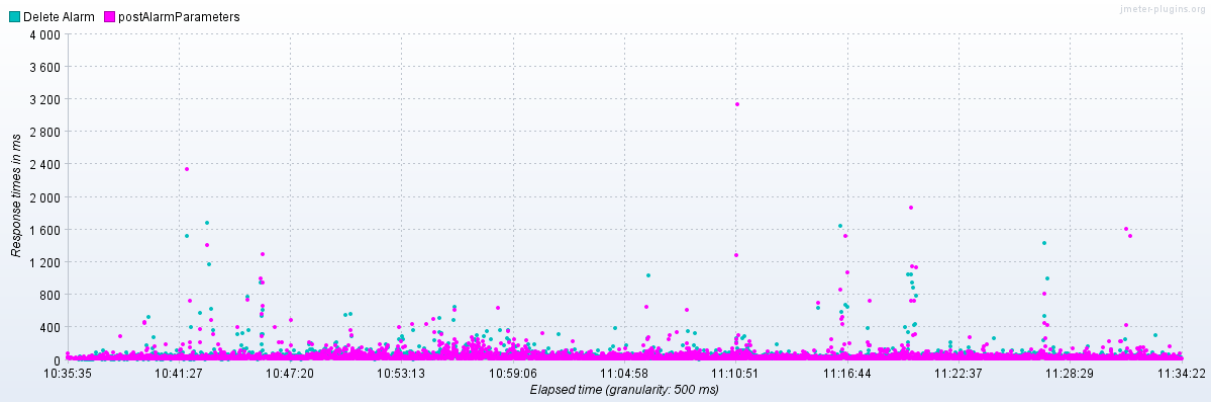

Abbildung 67 Lasttest mit Separierung ODS und CEPS auf 2 Maschinen: CEPS-Antwortzeiten. (Quelle: Eigene Darstellung).

4.6.9 Langlauftest

Im Zuge des Langlauftests erfolgt die Überprüfung der Langzeitstabilität des SUT auf Testumgebung #1 (siehe Abschnitt 4.5.2.1). Performance-Schwachstellen, die sich erst nach längerem Betrieb zeigen, werden mit diesem Test identifiziert. Tabelle 19 stellt die Rahmenbedingungen des Tests dar.

Tabelle 19 Langlauftest: Randbedingungen und Zusammenfassung.

Testziel	Überprüfung der Langzeitstabilität des SUT: Untersuchung, ob sich Performance und Ressourcenverbrauch über die Zeit konsistent verhalten und die Anwendung die erwarteten Performance-Merkmale auch unter Dauerlast erbringen kann.
Testszenario	Normallastbedingungen
Testendekriterium	3 Stunden Testlaufzeit
Testabbruchkriterium	5% Transaktionsabbrüche aller abgeschickten Anfragen
Testumgebung	Testumgebung #1
Gesamte Anzahl an parallelen Benutzern	PA: 140 Threads Monitoring: 105 Threads

In diesem Lastszenario wird die Lastobergrenze von 245 parallelen Benutzern schnell erreicht (Ramp-Up über 3 Minuten) und dann konstant über drei Stunden gehalten (siehe Abbildung 68).

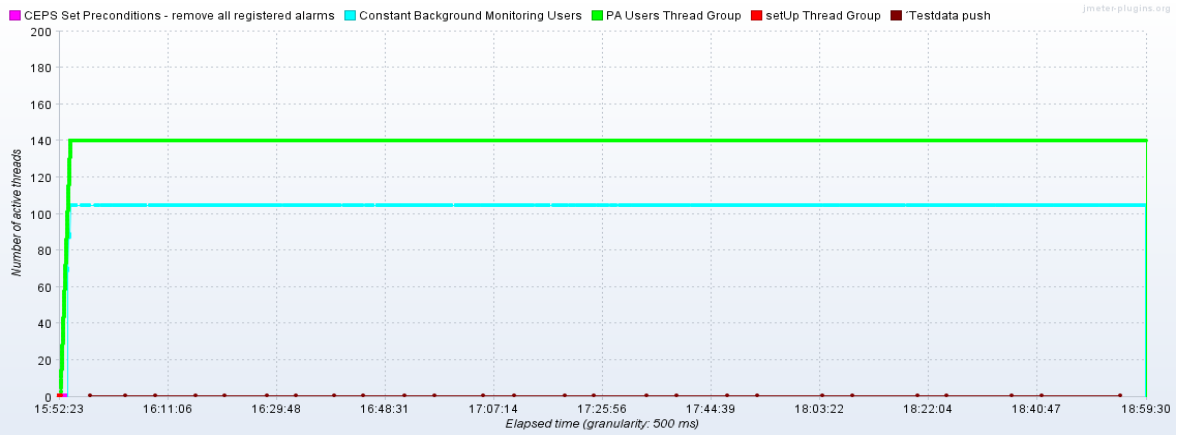

Abbildung 68 Langlauftest: Rampe an parallel aktiven Benutzern. (Quelle: Eigene Darstellung).

Bei der Analyse der Testergebnisse konnte kein abweichendes Verhalten über die Zeit festgestellt werden (vgl. Abbildung 69, Abbildung 70, Abbildung 71, Abbildung 72). Das SUT wird damit als Langlauf-stabil angenommen.

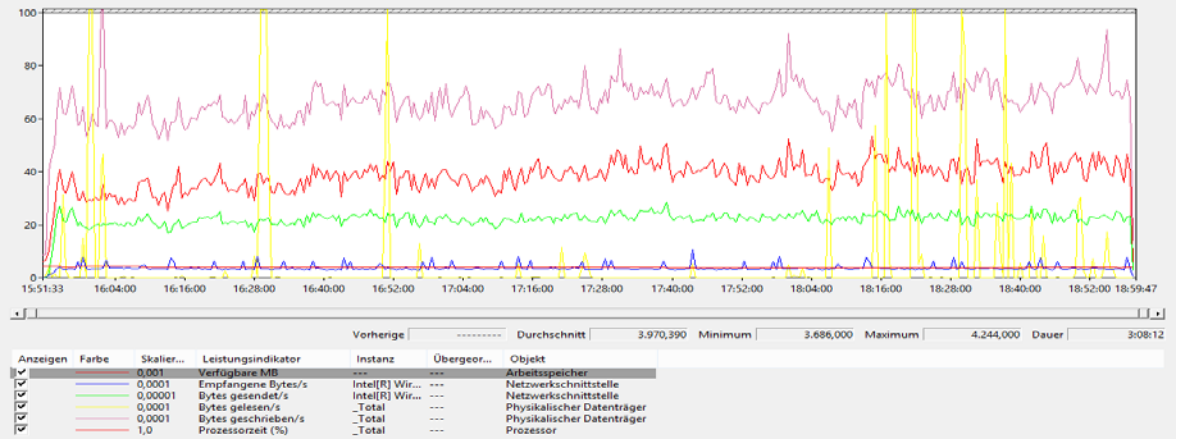

Abbildung 69 Langlauftest: Gesamtressourcenverbrauch der Maschine A: Prozessorauslastung, Speicherverbrauch, Netzwerkdurchsatz und Festplattendurchsatz. (Quelle: Eigene Darstellung).

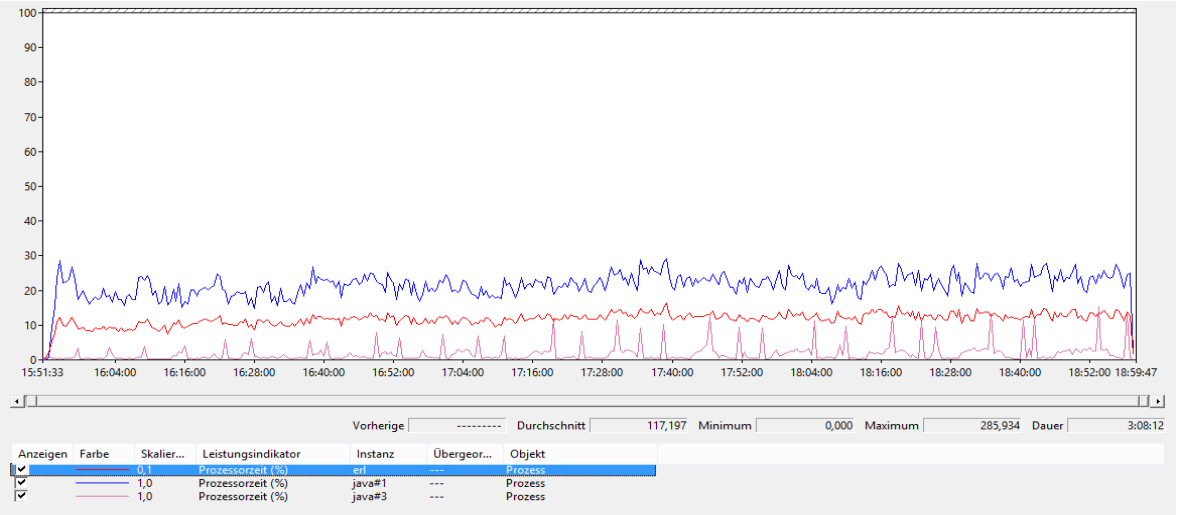

Abbildung 70 Langlauftest: Prozessorauslastung durch ODS (java#1), CEPS (java#3) und CouchDB (erl). (Quelle: Eigene Darstellung).

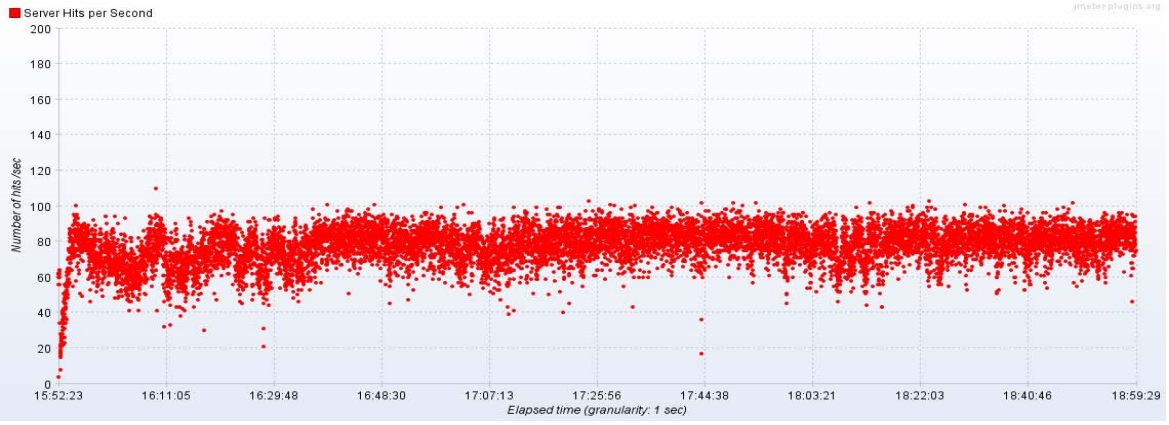

Abbildung 71 Langlauftest: Durchsatz (Hits/Sec.). (Quelle: Eigene Darstellung).

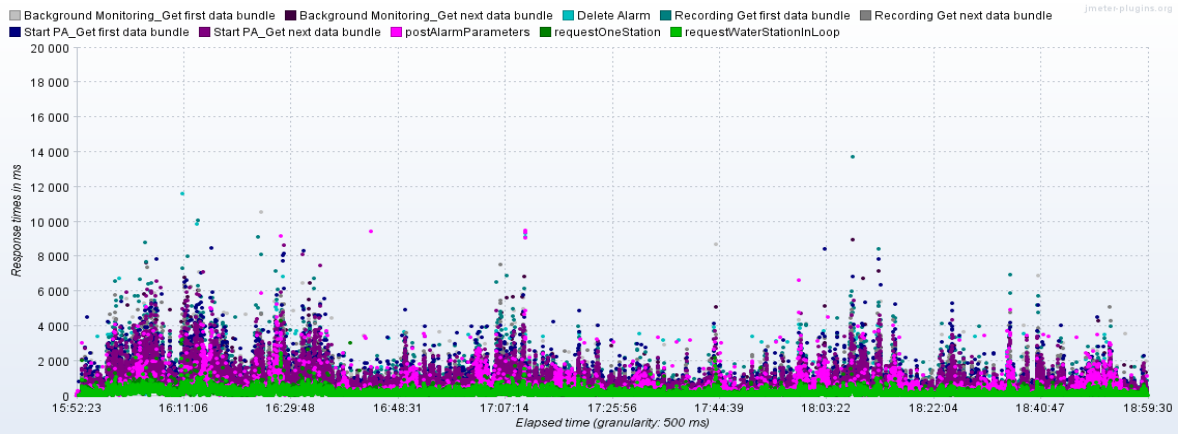

Abbildung 72 Langlauftest: Antwortzeiten. (Quelle: Eigene Darstellung).

5 Testergebnisse

Basierend auf der Analyse der Testergebnisse werden in den nächsten Abschnitten die identifizierten Performance-Probleme beschrieben und die möglichen Ursachen aufgezeigt.

5.1 ODS und CEPS Datenbank Connection Pool

Bereits in der frühen Phase der in dieser Arbeit durchgeführten L&P-Tests wurden die ersten Befunde identifiziert. Das erste Limit der Tests wurde bereits bei ca. 25 parallelen Benutzern erreicht – ab diesem Punkt überstieg die Anzahl der Transaktionsabbrüche 30% aller Anfragen. Ein massives Fehleraufkommen seitens ODS verhinderte eine weitere Testdurchführung. Alle Fehlermeldungen wiesen jedoch dasselbe, in Abbildung 73 exemplarisch dargestellte Muster auf.

```
ERROR [2015-03-29 18:45:59,397] io.dropwizard.jersey.errors.LoggingExceptionMapper: Error handling a
request: a41dcf5ad313a0b2
! org.apache.http.conn.ConnectionPoolTimeoutException: Timeout waiting for connection from pool
! at
org.apache.http.impl.conn.PoolingClientConnectionManager.leaseConnection(PoolingClientConnectionManager.jav
a:226) ~[httpclient-4.3.jar:4.3]
! at
org.apache.http.impl.conn.PoolingClientConnectionManager$1.getConnection(PoolingClientConnectionManager.jav
a:195) ~[httpclient-4.3.jar:4.3]
! at org.apache.http.impl.client.DefaultRequestDirector.execute(DefaultRequestDirector.java:422)
~[httpclient-4.3.jar:4.3]
! at org.apache.http.impl.client.AbstractHttpClient.doExecute(AbstractHttpClient.java:863) ~[httpclient-
4.3.jar:4.3]
! at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:82) ~[httpclient-
4.3.jar:4.3]
! at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:106) ~[httpclient-
4.3.jar:4.3]
! at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:57) ~[httpclient-
4.3.jar:4.3]
! at org.ektorp.http.StdHttpClient.executeRequest(StdHttpClient.java:175) ~[org.ektorp-1.4.2.jar:na]
! ... 67 common frames omitted
! Causing: org.ektorp.DbAccessException: org.apache.http.conn.ConnectionPoolTimeoutException: Timeout
waiting for connection from pool
```

Abbildung 73 ODS Fehlermeldung – die Anzahl an parallelen Datenbank-Zugriffen überschreitet den festgelegten Wert. (Quelle: Eigene Darstellung).

Eine Untersuchung der möglichen Fehlerursachen ergab, dass die Anzahl an maximal möglichen Datenbank-Verbindungen (*MaxConnections* in *DbModule.java*) in ODS und CEPS nicht explizit definiert wird und damit durch den Default-Wert festgelegt wird. Nach einer entsprechenden Anpassung der Implementierung im Rahmen des Bugfixings ist dieser Wert nun in der Konfigurationsdatei einstellbar, so dass die maximale Anzahl an Datenbank-Verbindungen auf einen sinnvollen Wert, der sich aus dem Verhältnis zwischen ODS- und CEPS-Server-Kapazität und -Anzahl sowie der Kapazität des Datenbank-Servers ergibt.

Dieser Befund kann als Performance-Bug eingestuft werden (wie im Abschnitt 2.4 definiert). Die oben genannte Behebung des Bugs optimiert die Auslastung eines einzelnen ODS- oder CEPS-Servers. Überschreitet die Nutzlast die Kapazität eines ODS- oder CEPS-Servers, so ist ein weiteres Anheben der maximalen Verbindungsanzahl keine Lösung. In diesem Fall muss auf eine Erhöhung der Server-Anzahl, verbunden mit entsprechendem Loadbalancing, zurückgegriffen werden.

5.2 CouchDB als limitierende Komponente

Betrachtet man die Skalierungseigenschaften der beiden Services (siehe Abbildung 21), so stellt sich die

Datenbank-Komponente, auf die beide Services zugreifen, als die limitierende Komponente dar. Sie erreicht als erste die Grenzen der ihr zur Verfügung stehenden Rechenkapazität (CPU) und verhindert damit eine lineare Skalierung der Services. Je mehr Benutzer über die Services gleichzeitig auf die Datenbank zugreifen möchten, desto stärker ist sie belegt. Besonders negativ sind dabei die Anfragen nach großen Datenmengen (wie etwa bei der Pegel-Aufzeichnungsfunktion) aufgefallen.

Außerdem erwies sich die CouchDB sehr sensibel auf Umgebungsschwankungen. In einem Testdurchlauf wurde die Maschine absichtlich durch andere Anwendungen stark belastet. Hierbei traten wiederholt Timeout-Fehler im laufenden Service-Betrieb auf, da die Kommunikation mit CouchDB zu lange dauerte (Abbildung 74).

```
INFO  [2015-04-15 12:24:57,420]
org.jvalue.ods.processor.ProcessorChainManager$ProcessorRunnable: starting processor chain "mainFilter" for
source "pegelonline"
ERROR [2015-04-15 12:26:36,339]
org.jvalue.ods.processor.ProcessorChainManager$ProcessorRunnable: error
while running processor chain
! java.net.SocketTimeoutException: Read timed out
! at java.net.SocketInputStream.socketRead0(Native Method) ~[na:1.7.0_75]
! at java.net.SocketInputStream.read(SocketInputStream.java:152)~[na:1.7.0_75]
! at java.net.SocketInputStream.read(SocketInputStream.java:122)~[na:1.7.0_75]
! at org.apache.http.impl.io.AbstractSessionInputBuffer.fillBuffer(AbstractSessionInputBuffer.java:160)
! at org.apache.http.impl.io.SocketInputBuffer.fillBuffer(SocketInputBuffer.java:84)~[httpcore-4.3.jar:4.3]
! at org.apache.http.impl.io.AbstractSessionInputBuffer.readLine(AbstractSessionInputBuffer.java:273)
! at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:140)
! at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:57)
! at org.apache.http.impl.io.AbstractMessageParser.parse(AbstractMessageParser.java:260)
! at org.apache.http.impl.AbstractHttpClientConnection.receiveResponseHeader
     (AbstractHttpClientConnection.java:283)
! at org.apache.http.impl.conn.DefaultClientConnection.receiveResponseHeader
     (DefaultClientConnection.java:251)~[httpclient-4.3.jar:4.3]
! at org.apache.http.impl.conn.ManagedClientConnectionImpl.receiveResponseHeader
     (ManagedClientConnectionImpl.java:197)~[httpclient-4.3.jar:4.3]
! at org.apache.http.protocol.HttpRequestExecutor.doReceiveResponse(HttpRequestExecutor.java:271)
! at org.apache.http.protocol.HttpRequestExecutor.execute(HttpRequestExecutor.java:123)
! at org.apache.http.impl.client.DefaultRequestDirector.tryExecute(DefaultRequestDirector.java:682)
! at org.apache.http.impl.client.DefaultRequestDirector.execute(DefaultRequestDirector.java:486)
! at org.apache.http.impl.client.AbstractHttpClient.doExecute(AbstractHttpClient.java:863)
! at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:82)
! at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:106)
! at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:57)
! at org.ektorp.http.StdHttpClient.executeRequest(StdHttpClient.java:175)
! ... 20 common frames omitted
! Causing: org.ektorp.DbAccessException: java.net.SocketTimeoutException: Read timed out
INFO  [2015-04-15 12:26:36,340]
org.jvalue.ods.processor.ProcessorChainManager$ProcessorRunnable: stopping processor chain
…
```

Abbildung 74 Timeout-Fehler im laufenden Betrieb, da die Kommunikation mit CouchDB aufgrund limitierter Ressourcen starken Verzögerungen unterworfen war. (Quelle: Eigene Darstellung).

5.3 CEPS Bottleneck

Bei der Durchführung der Tests fiel ein Verhalten von CEPS negativ auf: Beim Einspielen der Testdaten wurden die Bedingungen zum Versenden von Benachrichtigungen über gefeuerte Alarme an die betroffenen Clients geschaffen. Im Fall, dass Clients nicht erreichbar sind, schreibt CEPS entsprechende Fehlermeldungen auf die Console bzw. in das Logfile. Diese Fehlermeldungen kamen im Test jedoch nicht, wie erwartet, mehr oder wenigen gleichzeitig, sondern in einem ca. 15-Sekunden-Takt. In Abbildung 75 ist eine Reihe solcher Fehlermeldungen zu sehen.

```
ERROR [2015-04-14 09:45:30,582] org.jvalue.ceps.notifications.NotificationManager: Failed to send
notification to client  client_3_4
…
ERROR [2015-04-14 09:45:45,614] org.jvalue.ceps.notifications.NotificationManager: Failed to send
notification to client  client_4_0
…
ERROR [2015-04-14 09:46:00,645] org.jvalue.ceps.notifications.NotificationManager: Failed to send
notification to client  client_4_4
```

```
…
ERROR [2015-04-14 09:46:15,677] org.jvalue.ceps.notifications.NotificationManager: Failed to send
notification to client  client_5_2
…
ERROR [2015-04-14 09:46:30,708] org.jvalue.ceps.notifications.NotificationManager: Failed to send
notification to client  client_5_0
…
```

Abbildung 75 Eine Reihe zeitlich versetzter Fehlermeldungen seitens CEPS. Grund: die Clients für Benachrichtigungen sind nicht erreichbar. (Quelle: Eigene Darstellung).

Dieses unerwartete Verhalten wurde als ein Zeichen dafür interpretiert, dass das Verschicken von Benachrichtigungen sequentiell und nicht parallel abläuft. Waren alle Clients erreichbar, so erfolgte das Versenden der Benachrichtigungen nach dem gleichen Muster, nur in deutlich kürzerem Abstand (Abbildung 76).

```
INFO  [2015-04-20 12:26:31,495] org.jvalue.ceps.notifications.NotificationManager: Sending event to
client client_7_61 (deviceId: http://81.169.251.206:8080)
INFO  [2015-04-20 12:26:31,610] org.jvalue.ceps.notifications.NotificationManager: Sending event to
client client_7_63 (deviceId: http://81.169.251.206:8080)
INFO  [2015-04-20 12:26:31,709] org.jvalue.ceps.notifications.NotificationManager: Sending event to
client client_7_10 (deviceId: http://81.169.251.206:8080)
INFO  [2015-04-20 12:26:31,809] org.jvalue.ceps.notifications.NotificationManager: Sending event to
client client_7_33 (deviceId: http://81.169.251.206:8080)
INFO  [2015-04-20 12:26:31,909] org.jvalue.ceps.notifications.NotificationManager: Sending event to
client client_7_54 (deviceId: http://81.169.251.206:8080)
INFO  [2015-04-20 12:26:32,004] org.jvalue.ceps.notifications.NotificationManager: Sending event to
client client_7_47 (deviceId: http://81.169.251.206:8080)
```

Abbildung 76 Eine Reihe zeitlich versetzter Meldungen seitens CEPS beim Versenden von Benachrichtigungen. (Quelle: Eigene Darstellung).

Die Vermutung, dass die Benachrichtigungen an die Clients nicht parallel versandt werden, konnte durch Quellcode-Untersuchungen von CEPS (*NotificationManager.java*, *PatternListenerDispath.java*, *EsperManager.java*, *EventUpdateListener.java*) bestätigt werden: Die Esper Engine verwaltet die Events parallel und die Implementierung des Verhaltens verläuft bis zu den Grenzen der Esper Engine im Multithreading (*PatternListenerDispath*.java). Genau im Übergang zwischen der Esper Engine und CEPS erfolgt der Wechsel zwischen Multi-Threading und Single-Threading.

Das Problem ist in der Implementierung des CEPS NotificationManagers (*NotificationManager.java*) verortet: die Methode *onNewEvents* des *NotificationManagers* wurde vom Entwickler als laufzeitkritischer Abschnitt eingestuft und mit „*synchronized*“ gegen parallele Zugriffe geschützt. Das Multithreading serialisiert sich auf dieser Methode. Die Dauer der Blockierung für nachfolgende Threads hängt mit der Dauer des Methodenaufrufs zusammen, die im Falle eines HTTP-Clients, der nicht erreichbar ist, aus einem Timeout (hier ca. 15 Sek.) resultiert.

Aus diesem Befund wird geschlossen, dass die App PegelAlarm beim Versenden der Alarm-Benachrichtigungen im Betrieb stark limitiert ist. Gerade in den kritischen Zeiten des Hochwassers, wenn die erwartete Anzahl an gesendeten Alarmen stark ansteigt, kann diese Implementierung zu massiven Verzögerungen führen, selbst unter der Annahme, dass alle betroffenen Clients (in dem Fall über den GCM-Server) erreichbar sind. Der Methoden-Aufruf dauert ca. 122 ms (durchschnitt über 500 Methoden-Aufrufe), Benachrichtigungen werden von CEPS in einem 15-minütigen Intervall verschickt (entsprechend dem Aktualisierungsintervall von ODS mit der Datenquelle *PEGELONLINE*). Mit der aktuellen Implementierung kann CEPS unter Vollauslastung in dieser Zeit nur ca. 7.300 Benachrichtigungen verschicken (1 Alarm dauert 0,122 Sekunden, in 900 Sekunden können damit höchstens 7.300 Alarme bearbeitet werden). Diese Zahl – 7.300 Benachrichtigungen pro 15 Minuten – ist somit die maximale

Benachrichtigungskapazität des CEPS – sie kann durch Skalierung jeglicher Art nicht erhöht werden (außer, die Laufzeit pro Alarm wird verringert, dann kann eine quantitative Anpassung dieser Zahl stattfinden, jedoch keine qualitative). In dem Moment, in dem die Anzahl an gesendeten Alarmen diese Kapazität übersteigt, wird sich unvermeidliche eine Schlange aus Benachrichtigungen bilden. Wird einer der Client unerreichbar, sodass auf einen Timeout gewartet wird, werden alle weiteren Benachrichtigungen entsprechend verzögert.

Dieser Befund wird als Performance-Bottleneck eingestuft (wie im Abschnitt 2.4 definiert).

5.4 Dauerverhalten des SUT

ODS und CEPS produzieren während ihrer Laufzeit einen Datenüberschuss, es werden also mehr Daten erzeugt, als entfernt werden. Nach ca. 10 Tagen Normalnutzung des Systems (Synchronisierung mit der Datenquelle *PEGELONLINE* und Weiterleitung der Daten an CEPS, kein L&P-Test) ist die Festplatte der Host-Maschine (ca. 22 GB) komplett voll. Dies führt zu einer Nicht-Verfügbarkeit der Host-Maschine (keine Befehlsausführung mehr möglich). Ein manuelles Einschreiten ist erforderlich (die Datenbank komplett zurücksetzen). Beide Services sehen einen kontinuierlichen Betrieb vor und werden in der aktuellen Implementierung früher oder später im Produktiveinsatz durch dieses Problem ausfallen (je nach Intensität der Nutzung, wie in Abbildung 77 dargestellt).

Name	Size	Number of Documents	Update Seq
_replicator	4.1 KB	1	1
_users	8.1 KB	2	2
ceps-clients	3.2 GB	1	609809
ceps-epladapter	356.1 KB	1	89
ceps-events	338.6 MB	1	73019
ceps-sources	356.1 KB	1	89
ceps-users	12.1 KB	4	4
ods-datasources	380.1 KB	3	95
ods-osm	1.7 GB	4465274	4465276
ods-pegelonline	14.7 MB	637	11997

Showing 1-10 of 11 databases ← Previous Page | Rows per page: 10 | Next Page →

Abbildung 77 Ein Beispiel der volllaufenden CouchDB-Instanz: *ceps-clients* besteht aus einem Dokument, benötigt jedoch 3,2 GB aufgrund der hohen Anzahl an Aktualisierungen (*Update Seq*). (Quelle: Eigene Darstellung).

Das beobachtete Hosting der SUT unter root-Privilegien führte zur Nicht-Verfügbarkeit der Maschine und stellt ein Risiko für einen Produktiveinsatz dar.

5.5 Weitere Datenquellen in ODS

Die Anbindung weiterer Datenquellen in ODS ist ein Kandidat für eine mögliche Performance-Schwachstelle, da die Verwaltung zusätzlicher Daten den Service sowie die darunter liegende CouchDB belastet. Das beobachtete ODS-Verhalten bei der Anbindung weiteren Datenquellen stellt sich im Test jedoch als sehr positiv heraus (vgl. Abschnitt 4.6.6) - obwohl eine solche Anbindung die Systemressourcen beansprucht, wirkt sie sich kaum auf die Antwortzeiten aus. Nach dem Abschluss der Persistierung der über die neu angebundene Datenquelle gelesenen Daten in die Datenbank zeigt das System das gleiche

Ressourcennutzungsverhalten wie vor der Anbindung.

Eine zusätzliche Last, die auf ODS durch weitere Datenquellen produziert wird, wirkt sich negativ auf das Gesamtverhalten des Systems aus – sowohl die Antwortzeiten als auch Ressourcenausnutzung leiden darunter (vgl. Abschnitt 4.6.7).

5.6 CEPS-Sicherheitsrisiko

Eine interessante Beobachtung wurde im Rahmen der Testskript-Implementierung gemacht: Daten, die CEPS regelmäßig von ODS bekommt und auf denen er bei der Ereigniserkennung operiert, können theoretisch von einem beliebigen Absender stammen. Zwar verhindert ein in ODS und CEPS implementierter Authentifizierungsvorgang einen unerwünschten Zugriff von außen, dieser bietet jedoch kein Schutz vor Insider-Angriffen. Werden gefälschte Daten im korrekten Aufbau von einem böswilligen Insider an CEPS geschickt, werden sie genauso behandelt, wie echte Daten von ODS. Gerade für eine Anwendung wie PegelAlarm, die ein hohes Benutzervertrauen verlangt, stellt dies ein großes Risiko dar – falsche Alarmbenachrichtigungen können in den Katastrophenzeiten zu Panikausbrüchen führen. Dieses Sicherheitsrisiko sollte vor dem flächendeckenden Einsatz von CEPS bewertet werden.

6 Ergebnisse der Arbeit

Basierend auf der Analyse der Testergebnisse und der gemachten Erfahrungen mit dem SUT werden die Ergebnisse der Arbeit zusammengefasst und als eine Reihe an Verbesserungsvorschlägen zu Implementierung von ODS, CEPS und PegelAlarm vorgestellt. Weiterhin weist das Kapitel auf die Einschränkungen des ausgewählten Ansatzes hin und zeigt auf ein mögliches weiteres Vorgehen.

6.1 Verbesserungsvorschläge

Im folgenden Abschnitt werden die möglichen Maßnahmen dargelegt, mit denen die in Kapitel 5 aufgeführten Probleme gelöst werden können und die zur Steigerung der Performance-Eigenschaften führen.

6.1.1 Reduzierung der zu übertragenen Datenmengen

Die Testergebnisse haben gezeigt, dass die aktuelle Art und Weise des ODS-Einsatzes in der App PegelAlarm zum Transfer von massiven Datenmengen zwischen beiden führt. Dies wirkt sich nicht nur auf den Zeit- und Ressourcenverbrauch der in ODS eingesetzten CouchDB aus, sondern auch auf das Netzwerk. Wird eine Reduzierung der übertragenen Datenmengen erreicht, kann ODS unter gleichen Bedingungen viel mehr Anwender bedienen.

Eine Minimierung der übertragenen Datenmengen kann sowohl über eine Reduzierung der Anzahl an Anfragen (1), als auch über eine Reduzierung der übertragenen Datenmengen in einer Anfrage (2), erreicht werden. Folgend werden diese Vorschläge vorgestellt.

(1) PegelAlarm bekommt seine Daten von ODS und hält sie lokal in einer Datenbank, jedoch hat die App den Anspruch, den Anwendern die aktuellsten Daten zur Verfügung zu stellen: So werden die Daten, die älter als 30 Sekunden sind (aktuelle Einstellung für den *maxAge*-Wert aus Android-Ressourcen-Datei *R.integer*) als „alt“ interpretiert. Dies initiiert eine Anfrage nach neuen Daten aus ODS. Selbst unter der Annahme relativ kurzer ThinkTimes seitens der Anwender führt dies dazu, dass ein Wasserstand für den gleichen Anwender innerhalb seiner Sitzung mehrfach angefordert wird. ODS selber synchronisiert sich mit der Datenquelle *PEGELONLINE* einmal alle 15 Minuten (aktuelle Einstellung für das Synchronisierungsintervall, wie in Abschnitt 4.5.7 beschrieben) und die Datenquelle *PEGELONLINE* bekommt neue Daten in einem unregelmäßigen Intervall. Dies führt dazu, dass es, je nach zeitlichem Verlauf der Anfragen, zu Situationen kommen kann, in denen mehrfache Anfragen seitens PegelAlarm an ODS gestellt werden und die aktuellsten Daten geladen werden, die Daten an sich jedoch nicht „neu“ sind (im Sinne von: unterschiedlich von den schon vorhandenen Daten). In Abbildung 78 ist eine solche Situation dargestellt:

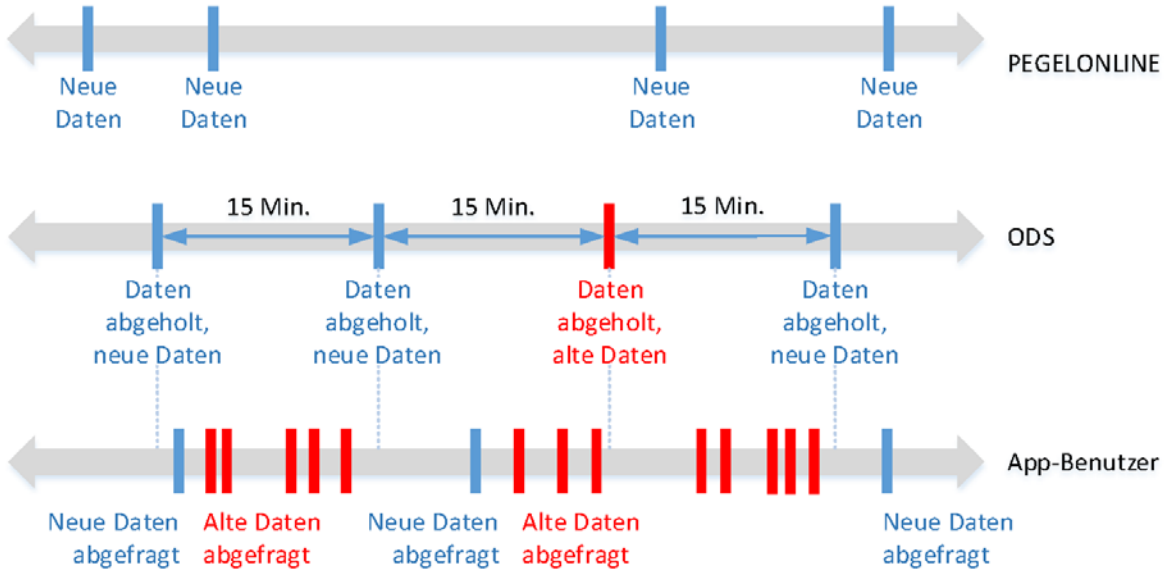

Abbildung 78 Synchronisierungsintervalle der Datenquelle *PEGELONLINE*, ODS und PegelAlarm. (Quelle: Eigene Darstellung).

Aus diesem Grund scheint die aktuelle Einstellung für den *MaxAge*-Wert viel zu streng zu sein. Um diese Timing-Effekte zu vermeiden wird vorgeschlagen, statt eines fixen *MaxAge*-Parameters eine etwas komplexere Zeitüberprüfung seitens PegelAlarm zu implementieren und dabei vor allem gegen das Synchronisierungsintervall des ODS zu prüfen. Dies kann eventuell für einzelne Anwender eine vernachlässigbare Performance-Einschränkung darstellen, wird aber den ODS massiv von unnötigen Abfragen entlasten.

Die aktuelle Implementierung von ODS setzt eine Grenze für die Anzahl an Datensätzen, die in einer Abfrage abgeholt werden können, diese ist aktuell auf einen fixen Wert – 100 – gesetzt. Die Intention dahinter ist durchaus nachvollziehbar: eine unbeschränkte Abfrage von allen Daten einer Datenquelle kann zu einem massiven Datentransfer führen. Jedoch stellt die Datenquelle *PEGELONLINE* aktuell Datensätze von über 600 Messstationen zur Verfügung. Dies führt bei einem Synchronisierungsvorgang (im Rahmen des ersten Starts der App oder bei Verwendung der Aufzeichnungsfunktion) zu einer Anfragen-Schleife seitens PegelAlarm. Eine Minimierung der Anzahl an HTTP-Anfrage kann über eine entsprechende Anpassung dieses Parameters erreicht werden. Dazu muss ODS dem Benutzer bzw. Administrator eine Möglichkeit der Kontrolle über diesen Parameter bieten. Eine Möglichkeit, diesen Wert pro Datenquelle einzustellen, wird den unnötigen Overhead der HTTP-Anfragen-Schleife beheben.

Die aktuelle Implementierung der Funktionalität „*Zeige alle Stationen entlang eines Flusses*“ der App PegelAlarm ist realisiert wie folgt: eine Liste der dem Fluss zugehörigen Stationen wird zusammengestellt. In einer Schleife wird anschließend jede einzelne Station separat bei ODS abgefragt. Diese Implementierung provoziert einen großen Overhead, da pro Station ein HTTP-Request initiiert wird. Durch das Anlegen einer extra View in CouchDB (Map- und Reduce-Funktionen, wie in der Abbildung 79 dargestellt) können alle notwendigen Daten in einer einzigen Abfrage angefordert werden.

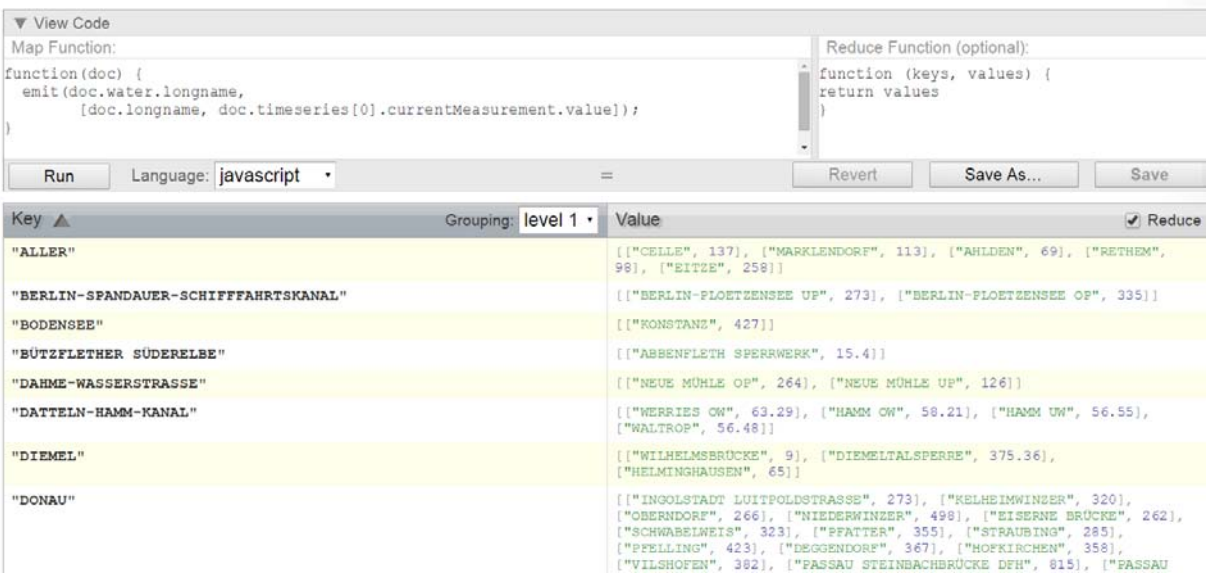

Abbildung 79 CouchDB-View, die aktuelle Wasserstands-Messungen für alle Stationen entlang eines Flusses liefern. (Quelle: Eigene Darstellung).

(2) Der Optimierungsvorschlag *Reduzierung der übertragenen Datenmengen für eine Anfrage* bezieht sich auf die aktuelle Implementierung der Abfrage der Daten von einzelnen Messstationen in PegelAlarm, die einen kompletten Datensatz (Dokument) aus der CouchDB anfragt. Ein Dokument beinhaltet verschiedene Messstation-Attribute, wie z. B. *Länge*, *Breite*, *Wassername*, *Agenturname*, etc. sowie eine Reihe der charakteristischen Messungen, wie z. B. *aktueller Wasserstand*, *Mittelwert der Wasserstände in einer Zeitspanne*, *mittlerer höchster Wert der Wasserstände in einer Zeitspanne*, *höchster bekannter Wasserstand* etc. App-Benutzer interessiert sich vor allem für die Aktualität eines einzigen Wertes – des *aktuellen Wasserstands*. Damit ist die Anfrage nach dem kompletten Dokument bei ODS seitens PegelAlarm überflüssig. Eine Anpassung der Implementierung kann die Menge an übertragenen Daten um 70% reduzieren: ein kompletter Datensatz zu einer Station beträgt ca. 1,4 KB, dabei ist der *aktuelle Wasserstand* (*timeseries.currentMeasurement*-Attribut des Dokuments) nur ca. 0,4 KB groß.

6.1.2 Beseitigung der CEPS-Bottlenecks

Eine Beseitigung des in Abschnitt 5.3 beschriebenen Bottlenecks in der Benachrichtigung über aufgetretene Alarme durch CEPS wird dringend empfohlen. Das Bottleneck stellt ein großes Risiko für den produktiven App-Einsatz dar und kann, gerade in kritischen Zeiten großer Hochwassergefahr, zu einem Versagen des Gesamtsystems führen.

Eine mögliche Lösung für dieses Problem ist, die *onNewEvents()*-Methode des *NotificationManagers* soweit zustandslos und Thread-sicher zu machen, dass keine *synchronized*-Implementierung notwendig ist – die EsperEngine läuft bis zum Serialisierungspunkt der bislang als *synchronized* markierten onNewEvents()-Methode parallel im multithreading. Eine weitere Lösungsmöglichkeit, die Ausführung des Abschickens der Benachrichtigungen zu parallelisieren, ist die Verwendung unabhängigen *Listener*-Instanzen statt einer zentralen *Listener*-Instanz (*NotificationManager*). Diese Lösung kann jedoch bei vielen App-Installationen (und damit vielen potentiellen Empfängern von Benachrichtigungen) zu Speicher-Problemen führen, da dafür eine große Anzahl an zusätzlichen Java-Objekten erzeugt werden

muss. Das dadurch entstehende Problem ließe sich gegebenenfalls durch einen dynamischen, begrenzten Pool an Listener-Instanzen reduzieren. Die Ausarbeitung einer solchen Lösung geht jedoch über den Rahmen dieser Arbeit hinaus und ist in einer weiterführenden Arbeit umzusetzen und zu validieren.

6.1.3 Explizite CEPS GarbageCollection

Bei der Analyse der Testergebnisse wurde ein Zusammenhang zwischen einer extremen Zunahme des Ressourcenverbrauchs seitens CEPS (punktuell von 10% auf 70%) mit dem Lauf der expliziten CEPS Garbage Collection (vgl. Abschnitt 3.6.3) festgestellt (siehe z. B. Abbildung 45) – insbesondere dann, wenn es zwischen den einzelnen GC-Läufen (aktuell 1 Stunde Abstand) eine große Anzahl an getriggerten Alarmen gegeben hat.

Besonders in den Hochwasserzeiten ist eine hohe Anzahl an ausgelösten Alarmen zu erwarten und eine entsprechend hohe Anzahl an gespeicherten Daten, die diese Alarme ausgelöst haben. Die App PegelAlarm benötigt jedoch diese Daten nicht: eine Benachrichtigung durch GCM mit der entsprechenden Ereignis-ID ist für das Verständnis, welcher Alarm getriggert wurde, ausreichend. Die gespeicherten Ereignisdaten werden von vorne hinein dem expliziten Garbage Collection überlassen. Daher ist zu empfehlen, diese Daten nicht zu speichern – hierdurch wird zum einen der Ressourcenverbrauch durch das Speichern vermieden, zum anderen reduziert sich der Aufwand für die explizite Garbage Collection enorm.

6.1.4 Feingranulare PegelAlarm-Einstellungen

Die Möglichkeit eines wellenartigen Lastaufkommens auf ODS (wie in Abschnitt 4.4.6 vorgestellt) kann vermindert werden, indem die App die Standard-Einstellung der Aufzeichnungsfunktion, vor allem des Aufzeichnungsintervalls, mit einem zufälligen Wert innerhalb eines definierten Zeitraums vorbelegt. Außerdem wird empfohlen, die dem Benutzer zur Verfügung stehenden Intervalle vom minimal möglichen Wert aufwärts viel feingranularer (z. B. in 5 Minuten-Schritten) anzubieten, damit sich die Synchronisierung des Monitorings über die Benutzer hinweg besser verteilt.

6.1.5 Performantere Gestaltung der HW-Ressourcen für CouchDB

Die Datenbank-Komponente der beiden Services – CouchDB – hat sich als limitierende Komponente herausgestellt (vgl. Abschnitt 5.2). Sie benötigt viel Rechenkapazität (CPU) und verhindert als erste Komponente eine lineare Skalierung der Services.

Generell unterscheidet man zwei Ansätze, Performance-Eigenschaften eines Systems zu beeinflussen: die erste Möglichkeit ist, den Ressourcenbedarf zu reduzieren, die andere liegt darin, die Qualität oder Quantität der verfügbaren Ressourcen zu erhöhen. Die erste Möglichkeit wurde bereits in Abschnitt 6.1.1 besprochen (Reduzierung der Anfragen und der zu übertragenen Datenmengen). Die andere Möglichkeit ist, den hohen Ressourcenbedarf seitens CouchDB über zusätzliche Ressourcen abzuwickeln und ihr ausreichend viele Ressourcen zur Verfügung zu stellen, die für die Erreichung der Akzeptanzkriterien notwendig sind. Dies bedeutet, CouchDB auf performanterer Hardware zu hosten (mehr CPU) oder über den Einsatz eines Load Balancings die eingehenden Anfragen auf mehrere Server zu verteilen.

6.1.6 Separate Deployments von ODS und CEPS

Eine Separierung von ODS und CEPS auf unterschiedliche Maschinen hat sich, entgegen den Erwartungen,

positiv auf das Zeitverhalten und den Ressourcenverbrauch ausgewirkt: trotz der durch die Netzwerkverbindung entstehenden Latenzen zwischen beiden Services wurden deutlich bessere Antwortzeiten beobachtet. Die vorhandene Architektur der beiden Services lässt sie gut auf verschiedene HW-Komponenten verteilen, die limitierende Komponente CouchDB muss in diesem Fall nur einen Service bedienen, und ihr stehen damit doppelt so viel Ressourcen für diese Aufgabe zur Verfügung. Somit beweist sich die Design-Entscheidung, CEPS als einen separaten Service zu implementieren, aus Sicht der CPU-Auslastung als positiv. Hierdurch wird jedoch das Netz mehr belastet, so dass eine Abwägung zu erfolgen hat.

6.1.7 Exklusive ODS und CEPS-Instanzen für PegelAlarm

CouchDB hat sich als extrem sensitiv für Umgebungsschwankungen erwiesen und wird als kritische Komponente des SUT eingestuft. Zur Erfüllung der NFA wird empfohlen, die ODS- und CEPS- Instanzen exklusiv für die App PegelAlarm auf exklusiver Hardware zu betreiben. Dadurch kann eine Beeinträchtigung durch andere Applikationen sowie andere Service-Consumer ausgeschlossen werden.

Ein weiterer Vorschlag basiert auf der Ausnutzung der verteilten Architektur der Services: die Wasserstand-Aufzeichnungsfunktionalität der App läuft im Hintergrund und hat deswegen weniger kritische NFA (wie im Abschnitt 4.2 definiert). Eine Auslagerung dieser Funktionalität auf eine weitere ODS-Instanz kann die ODS-Instanz, die interaktive Anfragen der App-Benutzer abwickelt, entlasten und damit bessere Antwortzeiten liefern.

6.2 Limitierungen und weiterführende Arbeiten

Grundsätzlich erwarten die Benutzer von einer App eine schnellere Reaktion als von einer Desktop-Anwendung (Gawande & Rudagi, 2012). Jedoch werden gerade die Endbenutzererfahrungen mit einer mobilen Anwendung von einer Vielzahl an Umgebungsbedingungen beeinträchtigt: Netzwerkverbindung (Wi-Fi, 3G, 4G, etc.), Qualität des GPS-Signals und die Geräte-Hardware sind von Fall zu Fall sehr unterschiedlich. Einige Plattform-APIs passen sogar ihr Verhalten an den Akkuladestand des Geräts an (Ravindranath et al., 2012). Die Betrachtung der Performance-Eigenschaften von mobilen Anwendungen aus dem Sichtpunkt der Endbenutzer ist daher von großer Bedeutung.

Für die Durchführung einer vollständigen Performance-Untersuchung einer App wird von (Yang, 2012) eine systematische Dekomposition der mobilen Anwendung in drei Hauptbestandteile vorgeschlagen, die sich auf die Erfahrung der Benutzer mit der Anwendung auswirken:

- Gerätebezogene Performance-Eigenschaften,
- Netzwerkbezogene Performance-Eigenschaften,
- Serverseitige Performance-Eigenschaften.

Die Gesamtperformance der Anwendung wird somit als die Kombination dieser Performance-Eigenschaften verstanden. Die durchgeführte Betrachtung der Performance-Eigenschaften der Services ist damit der erste Schritt von drei in einer vollständigen Performance-Untersuchung der App PegelAlarm.

Eine weitergehende Analyse der Client-Anwendung (die Software, die unmittelbar auf dem mobilen Gerät läuft) sowie des Netzverhaltens hat mit etwas anderen Herausforderungen zu tun: Geräte-bezogene

Performance-Eigenschaften einer App sind im Wesentlichen von den dem Gerät zur Verfügung stehenden technischen Möglichkeiten bedingt. Die zahlreichen Betriebssystemarten und -versionen tragen ebenfalls zu einer hohen Verhaltensvarianz bei. Die dadurch erzeugte große Anzahl an möglichen Permutationen aus Hard- und Software stellt eine der größten Herausforderungen für die Durchführung der L&P-Tests dar.

Netzwerkbezogene Performance-Eigenschaften sind schwer in einem Labortest reproduzierbar. Eine Reihe von aktuellen Studien konzentriert sich auf die Vorhersage von Mobilnetzwerkleistungen, z. B. (Xu, Mehrotra, Mao, & Li, 2013). Die Autoren prognostizieren, basierend auf gesammelten Daten zur aktuellen Netzwerk-Performance (Durchsatz, Paket-Verlust und Verzögerungen), mithilfe von Regressionsbäumen die zukünftige Netzwerkleistung.

Vorstellbar ist auch, das Verhalten von mobilen Anwendung im Produktiveinsatz zu überwachen und basierend auf Live-Daten eine Analyse der Performance-Engpässe und Ausfälle der App nach dem Ansatz von (Ravindranath et al., 2012), (Ravindranath, 2014) vorzunehmen.

Eine entscheidende Rolle für die Plausibilität der durch den L&P-Test erzielten Ergebnisse spielt der definierte Workload. Die durchgeführten L&P-Tests basieren auf Abschätzungen des erwarteten Nutzerverhaltens in Form von Nutzerverhaltensmodellen, den Think Times und der Lastintensität. Nach dem besten Wissensstand durchgeführt bleiben diese Abschätzungen jedoch Erwartungen. Wenn diese Informationen das erste Mal messbar zur Verfügung stehen, etwa nach dem ersten Produktiv-Einsatz der App PegelAlarm, ist nachzuprüfen, inwieweit sie mit den durchgeführten Schätzungen übereinstimmen. Die vorgenommenen Abschätzungen der erwarteten parallelen Anzahl an Benutzern sind auch als Richtwert zu sehen – wie Naturkatastrophen sich entwickeln und welche Ausmaße sie annehmen, kann man nur begrenzt voraussagen.

Die durchgeführten L&P-Tests basieren auf dem aktuellen Entwicklungsstand der App: sollten weiterführende Arbeiten an der Anwendung erweiterte Funktionalitäten zur Verfügung stellen oder die angestrebte Konsolidierung aller deutschen Wasserstands-Portale in den ODS durchgeführt werden, wird dies voraussichtlich zu einem veränderten Performance-Verhalten führen und muss neu betrachtet werden.

Auswirkungen der Interaktion zwischen dem SUT und anderen möglichen Consumern, die die Services zukünftig benutzen werden, konnten nur ansatzweise beobachtet werden (vgl. Abschnitt 4.4.7) – die generischen Funktionalitäten, die ODS und CEPS zur Verfügung stellen, lassen eine große Variation an möglichen Einsatzszenarien zu. Solche Untersuchungen können jedoch zusätzliche Informationen zu dem Performance-Verhalten der Services liefern und zeigen, ob die generische Natur der Services auch unter anderen Randbedingungen den gestellten nicht-funktionalen Anforderungen genügen kann. Die Betrachtung solcher weiteren Einsatzszenarien ist nicht Teil dieser Arbeit und muss in weiterführenden Arbeiten untersucht werden. Der zur Verfügung stehende Quellcode der beiden Services ist zudem eine gute Basis für eine Performance-Untersuchung aus der White-Box-Sicht, z.B. durch eine Code-Instrumentierung.

Die zur Verfügung gestellte Testumgebung entspricht nicht der geplanten Produktivumgebung. Im Rahmen der Ergebnisanalyse wurde dies berücksichtigt. Dennoch beeinflusst diese Differenz die Zuverlässigkeit der Analyseergebnisse, da Annahmen getroffen. Die getroffenen Annahmen sind in den jeweiligen Abschnitten

explizit herausgestellt. Dennoch wird eine Validierung der in dieser Arbeit getroffenen Optimierungsvorschläge auf der späteren Produktivumgebung dringend empfohlen.

Die in dieser Arbeit erstellten Markov-Ketten sind eine gute Grundlage für die Herleitung weiterer Informationen und können anderweitig eingesetzt werden: so können sie auch für ein statistisches Testen verwendet werden um eine Aussage über die Zuverlässigkeit des Systems zu gewinnen.

Zusammenfassung

Das Zusammenspiel zwischen dem JValue Open Data Service (ODS), dem JValue Complex Event Processing Service (CEPS) und der App PegelAlarm bietet den Menschen in Deutschland eine Unterstützung in Zeiten potenzieller Hochwassergefahr, indem es die Funktionalität eines Hochwasser-Alarms bereitstellt. Aufgrund dieser spezifischen Funktionalität ist das, für den L&P-Test benötigte, erwartete Nutzerverhalten recht speziell und richtet sich nach der Hochwassersaison, in der mit einem massiven Anstieg der Nutzerzahl zu rechnen ist.

Das Ziel der vorliegenden Arbeit ist die qualitative und quantitative Bewertung der Performance-Eigenschaften von ODS und CEPS am Anwendungsfall der App PegelAlarm mittels einer Reihe an L&P-Tests. Dazu wurde das erwarteten App-Nutzerverhalten definiert und in drei verschiedene Lastszenarien eingesetzt, die sich in der Anzahl an aktiven Benutzern und der Nutzungsintensität unterscheiden. Die auf das System gerichtete Last wurde mithilfe eines Lastgenerators erzeugt. Die Wahl fiel auf das von der Apache Software Foundation entwickelte JMeter sowie dessen Erweiterung um das von (Van Hoorn et al., 2008) vorgestellte Plug-In Markov4JMeter. Die Implementierung des JMeter-Skripts bildete damit die Grundlage für eine automatisierte Testdurchführung. Mit der Generierung der Testdaten und der Skript-Parametrisierung (je nach Lastszenario) wurde die Testvorbereitung abgeschlossen und in die Testdurchführung übergegangen.

Anhand der L&P-Tests wurde das System in einer kontrollierten Testumgebung den definierten Lastszenarien ausgesetzt, beobachtet und gegen die definierten Akzeptanzkriterien bewertet. Auf Basis der Analyse der Testergebnisse konnte die Frage nach der Skalierung und den Performance-Eigenschaften des Systems beantwortet werden. In Abhängigkeit der Lastszenarien verändert sich das beobachtete Ressourcenverbrauch- und Zeitverhalten. Ein CEPS-Bottleneck beim Versenden der Alarm-Benachrichtigungen stellt eine massive Limitierung eines erfolgreichen Einsatzes der App im Betrieb dar und muss behoben werden. Die CouchDB als Persistenzkomponente der beiden Services ist die limitierende Komponente des Gesamtsystems und verhindert eine lineare Skalierung. Außerdem hat sie sich in der aktuellen Konfiguration für einen Dauerbetriebseinsatz als nicht geeignet erwiesen, da während der Laufzeit von ODS und CEPS ein Datenüberschuss entsteht, der die Datendateien der Datenbank volllaufen lässt. Zudem sollte die Menge der Daten, die zwischen der App und dem ODS transferiert werden, verringert werden, um Netzwerk und Datenbank zu entlasten. Die Untersuchungen haben zahlreiche Datenstrukturen aufgezeigt, bei denen sich die Übertragung optimieren lässt. Eine Reihe an weiteren Optimierungspunkten und Verbesserungsvorschlägen für die Weiterentwicklung und den Betrieb wurde abgeleitet, welche zu besseren Performance-Eigenschaften des Systems führen.

Die Arbeit schließt mit einer Bewertung des Vorgehens und der Ergebnisse und zeigt die Limitierungen der Arbeit auf. Ein Ausblick auf das mögliche weitere Vorgehen für die Bewertung der Performance-Eigenschaften aus Sicht der Endbenutzer der App rundet die Gesamtbetrachtung ab.

Literaturverzeichnis

Abbors, F., Ahmad, T., Truscan, D., & Porres, I. (2012). *MBPeT: A Model-Based Performance Testing Tool.* Paper presented at the VALID 2012, The Fourth International Conference on Advances in System Testing and Validation Lifecycle.

Barber, S. (2004a). Beyond performance testing, parts 1-14. *IBM developer works, rational technical library*.

Barber, S. (2004b). *Creating effective load models for performance testing with incomplete empirical data.* Paper presented at the 26th Annual International Telecommunications Energy Conference, INTELEC 2004.

Baumgarten, C., Christiansen, E., Naumann, S., Penn-Bressel, G., Rechenberg, J., & Walter, A.-B. (2012). Hochwasser - Verstehen, erkennen, handeln. *Umweltbundesamt*.

Bautista, L., Abran, A., & April, A. (2012). Design of a performance measurement framework for cloud computing. *Journal of Software Engineering and Applications, 5*, 69-75.

Bear, T. (2006). Retrieved 13.01.2015, from http://blackanvil.blogspot.de/2006/06/shootout-load-runner-vs-grinder-vs.html

Bernardo, M., & Hillston, J. (2007). *Formal methods for performance evaluation*: Springer.

Card, S. K., Robertson, G. G., & Mackinlay, J. D. (1991). *The information visualizer, an information workspace.* Paper presented at the SIGCHI Conference on Human factors in computing systems.

Castillo, I., Losavio, F., Matteo, A., & Bøegh, J. (2010). REquirements, Aspects and Software Quality: the REASQ model. *Journal of Object Technology, 9*(4), 69-91.

Chow, T., & Cao, D.-B. (2008). A survey study of critical success factors in agile software projects. *Journal of Systems and Software, 81*(6), 961-971.

Chung, L., & do Prado Leite, J. C. S. (2009). On non-functional requirements in software engineering *Conceptual modeling: Foundations and applications* (pp. 363-379): Springer Berlin Heidelberg.

Consulting, F. (2009). eCommerce Web Site Performance Today. A commissioned study conducted by Forrester Consulting on behalf of Akamai Technologies, Inc.

Denaro, G., Polini, A., & Emmerich, W. (2004). *Early performance testing of distributed software applications.* Paper presented at the ACM SIGSOFT Software Engineering Notes.

Eichhorn, P. (2014). *A Notification Service for an Open Data Service.* (Bachelor's Theses), Friedrich-Alexander-Universität Erlangen-Nürnberg.

Elfar, I. K., & Whittaker, J. A. (2001). Model-Based Software Testing. *Encyclopedia of Software Engineering (edited by J.J. Marciniak), Wiley*.

Fowler, M. (2002). *Patterns of enterprise application architecture*: Addison-Wesley Longman Publishing Co. Inc.

Garvin, D. A. (1984). What does product quality really mean. *Sloan management review, 26*(1).

Gawande, A., & Rudagi, B. (2012). Mobile Application Performance Testing: An End-to-End Approach. Retrieved 30.03.2015, from http://www.devx.com/wireless/Article/48170

Geologie, H. L. f. U. u. (2013). Hochwasser Mai–Juni 2013 in Hessen. *Hydrologie in Hessen, Heft 10*.

Glinz, M. (2007). *On non-functional requirements.* Paper presented at the 15th IEEE International Requirements Engineering Conference, 2007. RE'07.

Hamburg, M., & Löwer, A. (2014). ISTQB/GTB Standard Glossar der Testbegriffe. Deutsch - Englisch. (Version 2.3 ed.): German Testing Board e.V.

Hruschka, P., & Starke, G. (2012). Template für Architekturdokumentation

Jain, R. (1991). *The art of computer systems performance analysis*: John Wiley & Sons.

Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012). Understanding and detecting real-

world performance bugs. *ACM SIGPLAN Notices, 47*(6), 77-88.
LfU. (2006). August-Hochwasser 2005 in Südbayern - Endbericht: Bayerisches Landesamt für Umwelt.
LfU. (2014). Junihochwasser 2013 - Wasserwirtschaftlicher Bericht (2. überarbeitete ed.): Bayerisches Landesamt für Umwelt.
Liggesmeyer, P. (2002). *Software-Qualität: Testen, Analysieren und Verifizieren von Software*: Spektrum Akademischer Verlag.
Lochmann, K., & Goeb, A. (2011). *A unifying model for software quality.* Paper presented at the 8th international workshop on Software quality.
Maier, A., Schmitt, H., & Rost, D. (2014). *PQ4Agile-Qualitätsmodell.*
Mannion, M., & Keepence, B. (1995). SMART requirements. *ACM SIGSOFT Software Engineering Notes, 20*(2), 42-47.
Martin, G. L., & Corl, K. G. (1986). System response time effects on user productivity. *Behaviour & Information Technology, 5*(1), 3-13.
Meier, J., Farre, C., Bansode, P., Barber, S., & Rea, D. (2007). *Performance testing guidance for web applications: patterns & practices*: Microsoft press.
Menascé, D. (2002). *Load testing, benchmarking, and application performance management for the web.* Paper presented at the Int. CMG Conference.
Molyneaux, I. (2014). *The Art of Application Performance Testing: From Strategy to Tools*: O'Reilly Media, Inc.
Prowell, S. J. (2005). *Using markov chain usage models to test complex systems.* Paper presented at the 38th Annual Hawaii International Conference on System Sciences, HICSS'05.
Ravindranath, L. (2014). *Improving the performance and reliability of mobile applications.* Massachusetts Institute of Technology.
Ravindranath, L., Padhye, J., Agarwal, S., Mahajan, R., Obermiller, I., & Shayandeh, S. (2012). *AppInsight: Mobile App Performance Monitoring in the Wild.* Paper presented at the OSDI.
Reel, J. S. (1999). Critical success factors in software projects. *Software, IEEE, 16*(3), 18-23.
Reischl, P. (2014). *Improving Data Quality using Domain-Specific Data Types.* (Master's Theses), Friedrich-Alexander-Universität Erlangen-Nürnberg.
Savoia, A. (2001). Web Load Test Planning: Predicting how your Web site will respond to stress. *STQE Magazine*.
Schroeder, B., Wierman, A., & Harchol-Balter, M. (2006). *Open Versus Closed: A Cautionary Tale.* Paper presented at the NSDI.
Schwartz, J. A. (2006). *Utilizing performance monitor counters to effectively guide windows and SQL server tuning efforts.* Paper presented at the Int. CMG Conference.
Spool, J. (2011). Do users change their settings. *Online article. Retrieved from* http://www.uie.com/brainsparks/2011/09/14/do-users-change-their-settings/.
Suffian, M. D. M., & Fahrurazi, F. R. (2012). *Performance testing: Analyzing differences of response time between performance testing tools.* Paper presented at the International Conference on Computer & Information Science (ICCIS).
Sung, S. J. (2011). *How can we use mobile apps for disaster communications in Taiwan: Problems and possible practice*. Paper presented at the 8th International Telecommunications Society (ITS) Asia-Pacific Regional Conference.
Tsysin, K. (2015). *Design of a Reflective REST-based Query API.* (Master's Theses), Friedrich-Alexander-Universität Erlangen-Nürnberg.
Van Hoorn, A., Rohr, M., & Hasselbring, W. (2008). Generating probabilistic and intensity-varying workload for web-based software systems. *Performance Evaluation: Metrics, Models and Benchmarks* (pp. 124-143): Springer.
Wade, J. (2012). Using Mobile Apps in Disasters. *Risk Management Magazine*.
Williams, L. G., & Smith, C. U. (1998). *Performance evaluation of software architectures.*

Paper presented at the 1st international workshop on Software and performance.
Woodside, M., Franks, G., & Petriu, D. C. (2007). *The future of software performance engineering.* Paper presented at the Future of Software Engineering, FOSE'07.
Xu, Q., Mehrotra, S., Mao, Z., & Li, J. (2013). *PROTEUS: network performance forecast for real-time, interactive mobile applications.* Paper presented at the 11th annual international conference on Mobile systems, applications, and services.
Yang, N. (2012). Testing Performance of Mobile Apps - Part 1: How Fast Can Angry Birds Run? *Methods & Tools, 20*(3), 16-27.
Zimmermann, F., Eschbach, R., Kloos, J., & Bauer, T. (2009). *Risk-based statistical testing: A refinement-based approach to the reliability analysis of safety-critical systems.* Paper presented at the 12th European Workshop on Dependable Computing, EWDC 2009.