UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Lasttests von Webanwendungen und korrespondierende Optimierungen

Load testing of web applications and corresponding optimizations

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Christian Friedrich

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Malte Schmitz

Lübeck, den 04. Dezember 2016

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne die Benutzung anderer als der angegebenen Hilfsmittel selbständig verfasst habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet.

(Christian Friedrich)
Lübeck, den 04. Dezember 2016

Kurzfassung Diese Arbeit beschäftigt sich mit mit dem Thema Lasttests von Webanwendungen und geht auf die Besonderheiten von Single-page Webanwendungen ein. Sie erläutert Merkmale, die bei einem Lasttest gemessen werden können und stellt Werkzeuge vor, mit denen ein Lasttest vorgenommen werden kann. Anhand der neu entwickelten Webseite des Hochschulsports der Universität zu Lübeck werden die Implementierung und Durchführung eines Lasttests erläutert. Bei dem durchgeführten Test kommt das User Behavior Graph Modell zur Anwendung, um ein besonders realistisches Nutzerverhalten nachzustellen. Des Weiteren werden Methoden erläutert, mit denen die Performance von modernen Webanwendungen optimiert werden kann. Dabei geht diese Arbeit auf die relativ neuen lokalen Webspeicher im Browser ein. An der Hochschulsport Webseite werden diese Verbesserungen vorgenommen und die Optimierung der Antwortzeiten, sowie die Ergebnisse des Lasttests, werden evaluiert.

Abstract This work deals with the topic of load tests of webapplications and addresses on the characteristics of single page applications. It explains features that can be measured during a load test and introduces tools for carrying out a load test. The implementation and carrying out of a load test are explained using the newly developed website of the Hochschulsport of the Universität zu Lübeck. In the performed test, the User Behavior Graph model is used to simulate a particularly realistic user behavior. Furthermore, methods are presented that can be used to optimize the performance of modern web applications. Thereby this work takes a look on the relatively new local webstorage in the browser. On the Hochschulsport website, these improvements are made and the optimization of the response times as well as the results of the load test are evaluated.

Inhaltsverzeichnis

1 Einleitung

Webanwendungen werden immer dynamischer und haben einen stetig steigenden Funktionsumfang. Mittlerweile existieren Office-Lösungen in Form von Webanwendungen, zum Beispiel Google Docs. Webanwendungen haben dabei einen entscheidenden Unterschied zu klassischen Desktopanwendungen. Sie werden von einer großen Nutzerzahl gleichzeitig verwendet. Das bedeutet, die Anwendung muss mit einer großen Anzahl gleichzeitiger Zugriffe zurecht kommen. Dieser Umstand erfordert, dass diese Anwendungen Lasttests unterzogen werden, um deren Zuverlässigkeit sicherzustellen. D.h. unter anderem der Anwender soll eine zuverlässige Verbindung aufbauen können, mit vertretbaren Wartezeiten. Außerdem ist diese Art von Test notwendig um einem Totalausfall wie beim Quelle-Ausverkauf [Rit10] oder bei der Health-Care Webseite in Amerika [SS13] vorzubeugen.

Bei einem Lasttest geht es darum, mittels vieler simulierter Anfragen herauszufinden wie belastbar eine Anwendung ist und sicherzustellen, dass sie korrekt funktioniert. Lasttests ersetzen jedoch keine funktionalen Tests wie Unit- oder Integrationstests. Sie unterscheiden sich dahingehend, dass sie ein oder mehrere funktionale und oder nicht funktionale Anforderungen untersuchen. Bei den funktionalen Anforderungen geht es insbesondere darum, Fehler in der Anwendung zu finden, die ausschließlich unter Last auftreten. Das sind z.B. Verschränkungen, die den Betrieb zum Erliegen bringen. Nicht funktionale Anforderungen sind festgelegte Kriterien, wie etwa Antwortzeiten, die auch bei einer definierten Last nicht überschritten werden dürfen.

Diese Arbeit beschäftigt sich insbesondere mit Lasttests von modernen Webanwendungen. Diese unterscheiden sich von klassischen dahingehend, dass sie neben gängigen Anfragen von HTML Dokumenten, Bildern und Javascript zusätzlich asynchrone Anfragen stellen. Im Rahmen dieser Arbeit wurden Lasttests für die neu entwickelte Verwaltungsseite des Hochschulsports der Universität zu Lübeck durchgeführt. Dabei handelt es sich um eine Single-page Webanwendung. Das ist eine besondere Form von moderner Webanwendung, die aus einem einzigen HTML Dokument besteht, dessen Inhalte dynamisch nachgeladen werden.

Darüber hinaus werden Maßnahmen erläutert, welche ergriffen wurden, um die Performance der Anwendung zu optimieren, damit sie den gestellten Anforderungen bei einem Lasttest genügt. Dies sind vor allem Maßnahmen, die sich bei modernen Webanwendungen ergeben, wie das Optimieren von Anfragen und das intelligente Zwischenspeichern von Daten für die spätere Verwendung.

1.1 Verwandte Arbeiten

Es gibt eine große Anzahl wissenschaftlicher Artikel, die sich mit dem Thema Lasttests auseinander setzen, jedoch beziehen sich diese auf klassische Webseiten, bei denen die anzuzeigende Webseite vollständig auf dem Server generiert und an den Browser ausgeliefert wird. Diese Arbeit beschäftigt sich vornehmlich damit, eine Single-page Webanwendung unter Last zu testen. Zudem liegt der zweite Schwerpunkt darauf, wie moderne Webanwendungen optimiert werden können. Sie geht also auf die Besonderheiten einer modernen Webanwendung bei einem Lasttest ein. Z. Jiang und A. Hassan haben in ihrem veröffentlichten Artikel *A Survey on Load Testing of Large-Scale Software Systems* [JH15] eine gute Übersicht über den aktuellen Stand zum Thema Lasttests erstellt. Des Weiteren gibt es umfangreiche Literatur, die sich mit diesem Thema auseinander setzt. Zum Beispiel von Microsoft das Buch *Performance Testing Guidance for Web Applications* [MFB+07], in welchem die Bearbeitungsschritte und Methoden für Lasttests erklärt werden.

Last kann über verschiedene Methoden erzeugt werden. Die einfachste ist es, sich ein Szenario, also eine feste Reihenfolge von Abfragen auszudenken und dieses simultan von vielen Clients abzuspielen. Bei diesem Vorgehen wird jedoch keine realitätsnahe Last auf dem *System unter Test* (SUT) erzeugt. Um realistisches Nutzerverhalten zu simulieren bieten sich Modelle wie der *User Behavior Graph (Markov Ketten)* [AL93] oder das *Formular Orientierte Modell* [DGH+06] an. In dieser Arbeit wurde das Modell des User Behavior Graph für den Lasttest verwendet, weil sich das Formular Orientierte Modell zu sehr an Webseiten mit Eingabeformularen orientiert und die untersuchte Software nicht auf jeder Seite ein Eingabeformular enthält. Beim User Behavior Graph können Zustände und Transitionen dagegen frei bestimmt werden.

1.2 Aufbau der Arbeit

Neben dieser Einleitung und der Zusammenfassung am Ende gliedert sich diese Arbeit in die folgenden Kapitel.

Kapitel 2 erläutert was Single-page Webanwendungen sind und welche Besonderheiten diese in Bezug auf Lasttests und Optimierungen haben.

Kapitel 3 beinhaltet eine Erklärung, welche Merkmale zur Definition von Anforderungen verwendet werden können und wie diese mittels eines Lasttests überprüft werden.

Kapitel 4 zeigt wie die Erfolgskriterien für den Lasttest festgelegt wurden und wie die Logdatei des alten Systems analysiert wurde, um daraus einen User Behavior Graph für das neue System zu erstellen. Des Weiteren wird auf die Implementierung und Testdurchführung in JMeter eingegangen.

Kapitel 5 gibt eine Übersicht über Maßnahmen die ergriffen wurden, um die Performance der Software zu optimieren.

Kapitel 6 beinhaltet die Ergebnisse das Lasttests und eine Gegenüberstellung der Antwortzeiten vor und nach den Optimierungen.

Kapitel 7 fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf weitere Möglichkeiten zur Optimierung der Performance.

2 Single-page Webanwendungen

In diesem Kapitel werden Single-page Webanwendungen beschrieben, weil sie zu den modernen Webanwendungen gehören und diese Arbeit auf deren Besonderheiten bei Lasttests und Optimierungen eingeht.

2.1 Was ist eine Single-page Webanwendung

Im Unterschied zu klassischen Webanwendungen, die aus mehreren miteinander verlinkten HTML Dokumenten bestehen, haben Single-page Webanwendungen nur ein einziges HTML Dokument. Dieses ist durch Javascript in der Lage Inhalte dynamisch und asynchron vom Server nachzuladen. Ein Vorteil liegt darin, dass der für die Darstellung notwendige HTML Code nur ein einziges mal zum Browser übertragen wird. Im Anschluss werden nur noch Daten von Ressourcen ausgetauscht, was den HTML Overhead, beim Seitenwechsel von klassischen Webseiten, erspart. Auf diese Weise werden die Lade- und Wartezeiten verkürzt. In Abbildung 2.1 wird der Unterschied zwischen klassischen und Single-page Webanwendungen grafisch Dargestellt.

Zu sehen ist, dass im klassischen Fall die Datenhaltung, Geschäftslogik und die Präsentationsschicht auf dem Server implementiert sind. Im Client findet nur die Darstellung der angeforderten Daten statt. Bei Single-page Webanwendungen hingegen findet die Entgegennahme der Daten auf dem Client statt. Dieser ist dann für die Präsentation verantwortlich. Wenn Daten zum Beispiel in Form von Tabellen oder Listen darzustellen sind, dann werden die dafür nötigen HTML Strukturen auf dem Client generiert. Das führt zur Entlastung des Servers, weil er diese Aufgabe bei einer klassischen Webanwendung normalerweise übernimmt.

Für die technische Umsetzung eignen sich Javascript Frameworks wie AngularJS [1], das von Google entwickelt und vorangetrieben wird. Dieses wurde auch verwendet, um die Hochschulsport Webseite zu implementieren. Es unterstützt den Entwickler dabei dynamische Webseiten zu erstellen. Ermöglicht wird das dadurch, dass die HTML Syntax erweitert wird, um z.B. Tabellen in Abhängigkeit von Daten flexibel

[1] https://angularjs.org/

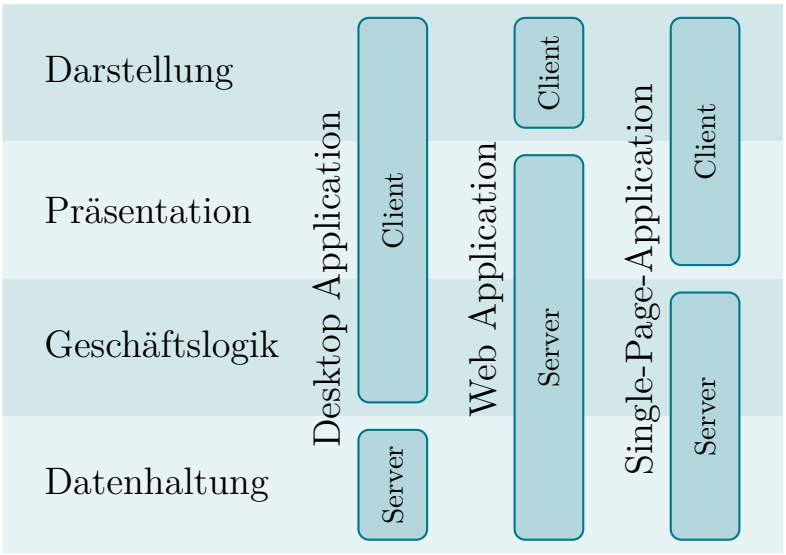

Abbildung 2.1: Vergleich Aufteilung der Schichten zwischen Server und Client bei Desktop-, Web- und Single-page Webanwendungen. Quelle: Malte Schmitz, 2012 aus dem Vortag „Moderne Webanwendungen mit Rails und Backbone.js“ veröffentlicht unter `https://github.com/malteschmitz/rails-backbone`

zu erzeugen. Zusätzlich ermöglicht AngularJS, dem Entwickler eigene HTML Direktiven zu implementieren. Damit der Quellcode übersichtlich bleibt, setzt es auf das Model-View-Controller Modell [LR01]. Mit AngularJS werden einfache Methoden zur Verfügung gestellt, um asynchrone Anfragen zu schreiben. Weil die Internetgemeinschaft sehr aktiv ist, gibt es zahlreiche weitere Frameworks, als Beispiele seien hier kurz BACKBONE.JS[2] und React[3] erwähnt.

2.2 Details der Hochschulsport Webanwendung

Damit eine Single-page Webanwendung Daten Anfragen kann, muss der Webserver in der Lage sein, solche Anfragen entsprechend zu verarbeiten. Bei der neu entwickelten Hochschulsport Webseite wurde dafür Ruby on Rails[4] eingesetzt. Es ist ein quelloffenes Web Application Framework, mit dem moderne Webanwendungen erstellt werden können. Programmiert wurde es in Ruby und es setzt, wie auch AngularJS, auf das Model-View-Controller Prinzip. Der Controller implementiert die CRUD Operationen zum Erstellen, Lesen, Bearbeiten und Löschen einer Ressource.

[2] http://backbonejs.org/
[3] https://facebook.github.io/react/
[4] http://rubyonrails.org/

Angesprochen werden die Operationen eines Controllers über konfigurierbare URLs. Ressourcen entsprechen im Rails Kontext Datenbanktabellen, die durch ein Model repräsentiert werden. Ressourcen können zueinander in Relation stehen, das heißt, die entsprechenden Entitäten in der Datenbank stehen über definierte Kardinalitäten in Beziehung zueinander. Zum Beispiel Sportart und Kurs mit den Kardinalitäten 1:n, was bedeutet, zu einer Sportart existiert eine beliebige Anzahl Kurse, ein Kurs gehört aber immer nur genau zu einer Sportart. In der View werden normalerweise das Model und der HTML Code für die Anzeige im Browser zusammengeführt. Das ist bei der Hochschulsport Webanwendung nicht der Fall, weil der Webclient dort die Zusammenführung übernimmt. Stattdessen wird in der View das Model ins JSON Format konvertiert, und dann an den Browser übertragen. JSON ist ein kompaktes Datenformat in einer einfach lesbaren Textform zum Zweck des Datenaustausches zwischen Anwendungen.

Die implementierte Schnittstelle der Hochschulsport Webanwendung bietet weitere Funktionalitäten an, als die bereits erwähnten CRUD Operationen. An die URLs, welche für die Controller konfiguriert wurden, können URL-Parameter angehängt werden, um die Lese Operation genauer zu spezifizieren. Beispielsweise werden nicht immer alle Felder eines Datensatzes benötigt, deswegen kann dem Controller durch den URL-Parameter „fields“ explizit mitgeteilt werden, welche Felder in der Antwort enthalten sein sollen. Des Weiteren gibt es einen URL-Parameter, mit dem eine Liste von Datensätzen gefiltert werden kann. Seine Bezeichnung ist „filter“ und im wesentlichen sorgt er dafür, dass nach einem beliebigen Datenbankfeld mittels LIKE Operator gefiltert wird.

2.3 Besonderheiten in Bezug auf Lasttests und Optimierungen

Klassische und Single-page Webanwendungen weisen einen großen Unterschied bei Lasttests auf. Im Fall der klassischen Webanwendung wird, das vom Browser anzuzeigende HTML Dokument vollständig auf dem Server generiert. Der Browser erhält also ein vollständiges HTML Dokument oder gar keins. Damit ist der gesamte Prozess zum generieren der Seite maßgebend für die Wartezeit des Nutzers. Bei einer Single-page Webanwendung hingegen tritt das nur beim ersten Öffnen der Webanwendung auf, weil dann das einzige HTML Dokument und der Javascript Code geladen werden müssen. Bei jedem darauf folgenden Seitenwechsel, kann die vom Anwender wahrgenommene Wartezeit, verkürzt werden, indem zum Beispiel Teilinformationen die bereits zur Verfügung stehen angezeigt werden, während andere Daten im Hintergrund weiter geladen werden. Möglich ist das dadurch, dass die Daten für eine Seite über mehrere asynchrone Anfragen vom Server geholt werden

können. Trotzdem sind die Antwortzeiten auch bei Single-page Webanwendungen wichtig. Es wird einen Nutzer nicht zufrieden stellen, wenn er vereinzelte Informationen früh angezeigt bekommt, auf die für ihn relevanten aber lange warten muss.

Single-page Webanwendungen bringen Änderungen in der Architektur mit sich, die besondere Optimierungen erfordern. Dazu gehört die erhöhte Kommunikation zwischen Server und Client aufgrund der separaten Anfragen für HTML Dokument und Daten. Ziel sollte es sein, Daten mit effizienten und möglichst wenigen Abfragen zu erhalten. Alle gängigen Webbrowser unterstützen mittlerweile lokalen Webspeicher [Wor16], in denen Webanwendungen Daten hinterlegen können, die nur von der jeweiligen Webanwendung einsehbar sind. Es wird dabei Unterschieden zwischen *sessionStorage*, der nach Ende einer Sitzung gelöscht wird und *localStorage*, der dauerhaft gespeichert wird. Eine Sitzung endet mit dem Schließen des Browsers. Dieser neue Speicher kann unter anderem dafür verwendet werden Ergebnisse von Abfragen zwischen zu speichern und somit z.B. die Aufgabe eines Caches übernehmen.

3 Anforderungen

In diesem Kapitel werden die Merkmale zum Festlegen von Anforderungen für einen Lasttest aufgezählt und erklärt. Weiterhin wird dargelegt, mit welcher Software Lasttests durchgeführt und die gestellten Anforderungen überprüft werden können.

3.1 Merkmale

Es wurden hier spezielle Eigenschaften gewählt, mit denen später in der Evaluation die Resultate der Optimierungen bewertet werden können. Dazu eigenen sich zum Beispiel der Durchsatz und die Antwortzeit. Zuverlässigkeit und APDEX eignen sich dazu, die Ergebnisse des Lasttests zu bewerten. Die genannten Merkmale können auch für die Definition von Anforderungen einer Webanwendung unter Last dienen.

3.1.1 Durchsatz

Der Durchsatz gibt die Anzahl der Anfragen an, die von der Webanwendung innerhalb einer Sekunde abgearbeitet werden. Aus diesem Wert lässt sich nicht direkt

$$Durchsatz = \frac{\#Anfragen}{Sekunde}$$

herleiten wie viele Nutzer die Webanwendung parallel verwenden können bis es zu Ausfällen wegen Überlastung kommt. Das liegt daran, dass ein Nutzer zwischen einzelnen Aktionen unterschiedliche Nachdenkzeiten hat, in denen er zum Beispiel Text liest. Der Durchsatz eignet sich aber gut dafür, Anfragen zu ermitteln, die ineffizient sind und optimiert werden sollten.

3.1.2 Antwortzeit

Das ist die Zeit, welche zwischen dem Absenden einer Abfrage und dem Erhalten einer Antwort vergeht. Die Antwortzeit steigt mit zunehmenden gleichzeitigen Anfragen an. Sie hängt von verschiedenen Faktoren ab. Das sind zum Beispiel die Ver-

bindung vom Client zum Server und die Verarbeitungszeit des Servers. Um die Antwortzeit zu bestimmen, sollten mehrere Anfragen nacheinander durchgeführt und aus diesen der Mittelwert berechnet werden, um Rauschen, welches durch Schwankungen in der Verbindungsgeschwindigkeit entsteht, zu filtern.

3.1.3 Zuverlässigkeit

Zuverlässigkeit ist ein wichtiges Merkmal von ständig verfügbaren Webanwendungen. Sie ist die Wahrscheinlichkeit für das ordnungsgemäße Funktionieren einer Software, unter spezifizierten Umgebungsbedingungen für einen festgelegten Zeitraum. Sie eignet sich dazu, in einem Wert zusammenzufassen, wie verlässlich eine Software funktioniert. Es gibt verschiedene Modelle, um die Zuverlässigkeit zu berechnen. An dieser Stelle wird auf das Nelson Modell [AW95] eingegangen.

Sei P das Programm und S die Spezifikation von diesem. Ein Element des Eingaberaums von P wird mit n bezeichnet und $p(n)$ ist die Wahrscheinlichkeit, dass n für P gewählt wird. Zum Bestimmen ob eine Eingabe erfolgreich war oder nicht dient

$$\alpha(n) = \begin{cases} 0 & wennP(n) = S(n) \\ 1 & sonst \end{cases} .$$

Die Wahrscheinlichkeit, dass sich ein Programm wie spezifiziert verhält, kann dann mit der Summe $\sum_{n \in D} p(n)a(n)$ berechnet werden. Die Zuverlässigkeit berechnet sich dann wie folgt.

$$R(P) = 1 - \sum_{n \in D} p(n)a(n) = \sum_{n \in D} p(n)(1 - a(n))$$

In dieser Arbeit wurde für den Lasttest ein stochastisches Modell verwendet, um ein möglichst realitätsnahes Benutzerverhalten zu simulieren. Im folgenden Kapitel wird das Modell im Detail erklärt. An dieser Stelle sei nur kurz erwähnt, dass das Modell aus Zuständen mit Kanten und Wahrscheinlichkeiten für Transitionen besteht. Bei dem verwendeten Modell berechnet sich die Zuverlässigkeit über erfolgreiche und fehlgeschlagene Zustandswechsel. Im folgenden werden Zustände mit $s(j)$ bezeichnet, wobei j die assoziierte Aktion ist. Weiterhin sind s_s gesendete und s_p durchgeführte Zustandswechsel. Fehlgeschlagene Vorgänge werden mit der folgenden Funktion berechnet.

$$d(s_s, s_p) = \sum_{j=1}^{k} s_s(j) - s_p(j)$$

Die Menge der untersuchten Zustände wird mit $1, 2, \ldots, z$ bezeichnet. s_s^i sind die Zustände im i-ten Testfall. Die Zuverlässigkeit berechnet sich dann wie folgt. Dabei sei $\alpha(s) = \sum_{j=1}^{k} s(j)$.

$$R[p, S](p) = \sum_{i=1}^{z} p(i) \left(1 - \frac{d(s_s^i, s_p^i)}{\alpha(s_s^i)}\right)$$

3.1.4 Apdex

Alleine mit gemessenen Werten, wie zum Beispiel Antwortzeiten eine Aussage über die Qualität einer Webanwendung zu treffen, ist schwierig. Aus einer gemessenen Antwortzeit wird nicht klar, ob ein Anwender zufrieden mit der Reaktionszeit ist. Das Verwenden von Mittelwerten bei vielen gemessenen Antwortzeiten löscht wichtige Details in der Messwerteverteilung und kann den Umstand verbergen, dass viele Nutzer mit signifikant größeren Antwortzeiten, als dem Mittelwert, frustriert sind. Der Apdex löst diese Probleme, indem er eine Menge von Messwerten auf ein gleich verteiltes Intervall von null bis eins abbildet. Null bedeutet dabei, dass kein Nutzer zufrieden ist und eins das alle zufrieden sind. Mit der folgenden Formel wird der Apdex bestimmt. Für die Bestimmung des Wertes müssen zuvor zwei Schwellwerte festgelegt werden. Der erste heißt Toleranzschwellwert, dieser gibt an, ab welcher Dauer die Wartezeit als toleriert eingestuft wird. Zeiten die darunter liegen gelten als zufriedenstellend und diejenigen die länger Dauern als toleriert, solange sie kleiner als der zweite Schwellwert sind. Das ist der Frustrationsschwellwert, Wartezeiten die länger dauern, sorgen für unzufriedene Nutzer.

$$Apdex = \frac{\#zufrieden + \frac{\#toleriert}{2}}{\#gesamt}$$

#zufrieden ist die Anzahl der Anfragen, deren Antwortzeit unter dem definierten Toleranzschwellwert liegt und *#toleriert* ist die Anzahl der zwischen Toleranz- und Frustrationsschwellwert liegenden Anfragen. *#gesamt* ist die Anzahl aller Anfragen. Die APDEX Formel ist die Anzahl der zufriedenstellenden Anfragen plus die Hälfte der Anzahl der tolerierten Anfragen plus keine der frustrierenden Anfragen geteilt durch die Anzahl aller Anfragen.

3.2 Möglichkeiten zum Überprüfen von Anforderungen durch Lasttests

Es gibt verschiedene Softwarelösungen, sowohl kostenpflichtige als auch kostenfreie, mit denen die in diesem Kapitel genannten Merkmale, überprüft werden können.

Die Firma Hewlett Packard bietet z.B. die Software HP LoadRunner[1] an. Diese ist kostenfrei, solange nicht mehr als 50 Nutzer simuliert werden. Für Tests, die darüber hinausgehen, müssen Lizenzen erworben werden, deren Kosten sich an der zu simulierenden Nutzerzahl orientieren. Der Vorteil dieser Lösung besteht darin, dass HP die Infrastruktur bereitstellt, um auch sehr große Lasten zu erzeugen. Die Software bietet aber keine fertige Lösung, um einen User Behavior Graph abzubilden, außer diesen selbst im Code zu implementieren. Für den Lasttest der Hochschulsport Webseite sind mehr als 50 Nutzer notwendig, aus diesem Grund und der fehlenden Unterstützung des User Behavior Graph Modells, wurde der HP LoadRunner nicht für den Test verwendet.

Es gibt auch komplett kostenfreie Werkzeuge, wie die quelloffene Software Gatling[2]. Diese ist in der Programmiersprache Scala geschrieben und wirbt damit, besonders viele Nutzer simulieren zu können. Gatling unterstützt auch eine stark vereinfachte Form des User Behavior Graph Modells. Stark vereinfacht bedeutet, dass der Graph einen Start- und Endknoten haben muss, sowie frei von Zyklen sein muss. Auf der Hochschulsport Webseite gibt es Zyklen in der Navigation, aus diesem Grund schied diese Software, für die Durchführung des Lasttests in dieser Arbeit, aus. Ebenfalls quelloffen ist das Java Framework JMeter[3]. Es bietet den Vorteil, dass es User Behavior Graphen in Form eines Plugins unterstützt. Diese Tatsache und das es kostenfrei ist, haben dazu geführt, dass sie für den Lasttest der Hochschulsport Webseite verwendet wurde. Weiterhin ist es mit JMeter möglich, ohne zusätzlichen Aufwand, den Test auf mehreren Rechnern gleichzeitig auszuführen, dies müsste bei Gatling manuell implementiert werden. In Unterabschnitt 4.2.3 wird genauer auf JMeter eingegangen.

Die oben genannten Werkzeuge haben gemeinsam, dass sie während des Testlaufs diverse Daten aufzeichnen. Dazu gehören zum Beispiel Antwortzeiten, Serverantworten, Anzahl der Anfragen und Zeitpunkte der Anfragen. Aus den aufgezeichneten Daten können dann, die in diesem Kapitel genannten Merkmale, bestimmt werden. Alle hier erwähnten Softwarelösungen erlauben es, nach dem Testdurchlauf einen automatischen Bericht zu erstellen, der die Ergebnisse in Diagrammen übersichtlich zusammenfasst.

[1] http://www8.hp.com/de/de/software-solutions/loadrunner-load-testing/
[2] http://gatling.io/
[3] http://jmeter.apache.org/

4 Lasttest

In diesem Kapitel wird erläutert, welche Mindestanforderungen für die zu analysierende Hochschulsport Webanwendung festgelegt wurden. Es folgt eine Beschreibung der Implementierung des Lasttests von Analyse der Logdatei zur Modellierung des User Behavior Graphs bis zur Umsetzung in JMeter. Im letzten Abschnitt werden die für den Test durchgeführten Schritte dargelegt.

4.1 Mindestanforderungen für Hochschulsport Webseite

Die zu testende Webseite ist eine Neuentwicklung der früheren Hochschulsport Webseite. Diese besteht aus einem öffentlichen Bereich, auf dem sich Studenten für angebotene Sportkurse einschreiben können. Wenn ein Kurs bereits vollständig belegt ist, besteht die Möglichkeit, sich auf eine Warteliste einzutragen. Wird ein Platz im Kurs frei, werden die Personen auf der Warteliste darüber informiert. Des Weiteren können Studenten sich auf der Seite informieren, zu welcher Uhrzeit und an welchem Ort ein Kurs stattfindet. Zusätzlich gibt es einen nicht öffentlichen Bereich, der für die Verwaltung des Hochschulsports dient. Dort werden unter anderem die Kurse der Semester, Teilnehmer und Finanzen verwaltet.

Weil das SUT eine Neuentwicklung der bestehenden Hochschulsport Webseite ist, wurden die Mindestanforderungen aus den Nutzerzahlen der aktuellen Webseite abgeleitet. Dafür wurde zunächst die Logdatei mit AWStats[1] analysiert. Das ist eine freie Webanalyse-Software, mit deren Hilfe Logdateien ausgewertet werden können. Sie wurde verwendet, weil ihre Handhabung einfach ist und eine gute Übersicht über Benutzerzahlen und Seitenaufrufe liefert. Auf diese Weise wurde schnell herausgefunden, in welchen Monaten und insbesondere an welchen Tagen die meisten Seitenaufrufe stattfanden.

In der Regel sind die Zugriffszahlen besonders hoch, wenn die Sportkurse für ein Semester freigeschaltet werden. In einem Jahr gibt es zwei solcher Termine, diese sind immer vor Semesterbeginn. An genau diesen Tagen sind die höchsten Zugriffszahlen während des gesamten Jahres. Der Abbildung Abschnitt 4.1 ist zu entnehmen, dass

[1] http://www.awstats.org/

im Monat der Freischaltung (März) die meisten Zugriffe stattfanden. Stichtag ist der 15. März, alleine an diesem wurden 26624 Zugriffe registriert. Am ersten Tag der Anmeldung registrierten sich ca 800 Personen für einen Kurs. 300 davon in den ersten drei Stunden zwischen 7 - 10 Uhr. Aus diesen Informationen wurde für das neue System als Mindestvoraussetzung festgelegt, dass innerhalb von drei Stunden mindestens 300 Personen in der Lage sein sollen einen Kurs zu buchen ohne, dass das System überlastet. Für eine weitere Mindestvoraussetzung wurden die parallelen

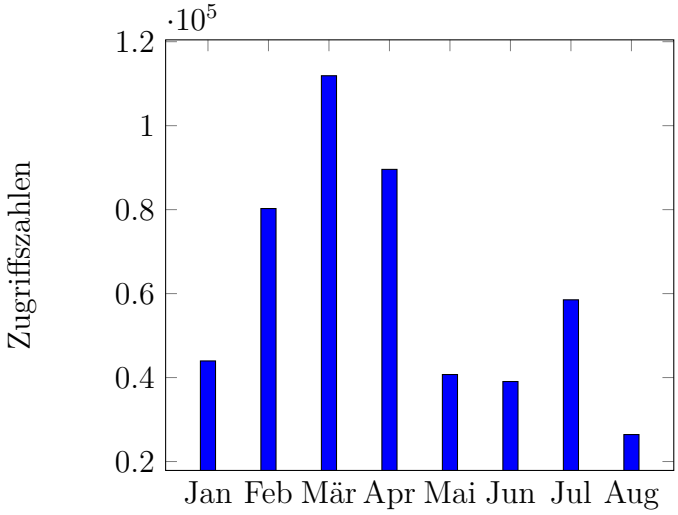

Abbildung 4.1: Verteilung der Zugriffszahlen auf die alte Hochschulsport Webseite im Jahr 2016.

Zugriffe betrachtet. In Abbildung 4.2 wird ersichtlich, dass in über 90 Prozent der Fälle nicht mehr als fünf Anfragen je Sekunde gestellt werden. In zwei Prozent der Fälle traten mehr als 10 Anfragen je Sekunde auf. Im Mittel erreichten das System 1,6 Anfragen je Sekunde. Daraus wurde als Mindestvoraussetzung abgeleitet, dass das System bei mindestens 10 parallelen Anfragen stabil funktionieren muss und mit vertretbaren Antwortzeiten arbeiten soll. Neben den hier aufgestellten Anforderungen, soll auch ermittelt werden, bei welcher Last die Ressourcen des Systems erschöpfen.

4.2 Lasttest Implementierung

Für die Implementierung wurde Apache JMeter verwendet, weil es unter anderem die Möglichkeit bietet, einen Lasttest mit probabilistischem Nutzerverhalten durchzuführen. Das bedeutet, dass simulierte Nutzer ihre Abfragen nicht in einer festen Reihenfolge durchführen. Sowohl die Reihenfolge als auch die Anzahl der Abfragen

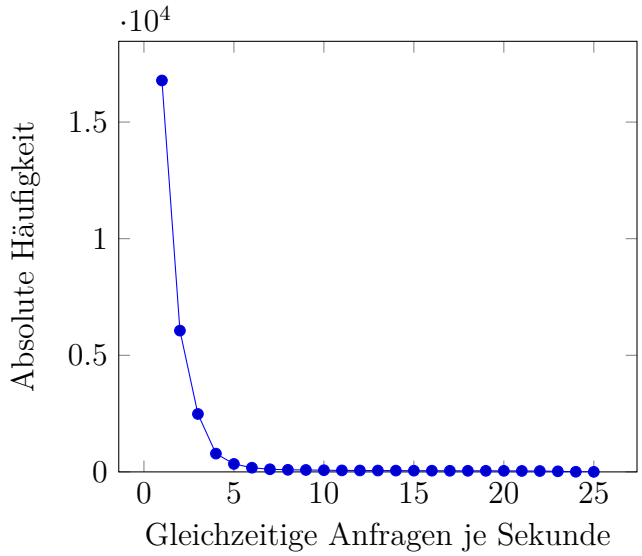

Abbildung 4.2: Verteilung gleichzeitiger Anfragen auf dem alten Hochschulsport System.

kann variieren. Erreicht wird das mit Markov Ketten(User Behavior Graph). Auf diese Weise ähnelt die Last der im Produktivbetrieb und sorgt so für reellere Testbedingungen. Zunächst wird im folgenden Unterabschnitt 4.2.1 dieses Modell erklärt. Anschließend wird angegeben wie die Daten für die Markov Kette erhoben wurden. Danach folgt die Erläuterung, was JMeter ist und die Implementierung des Lasttest wird skizziert.

4.2.1 User Behavior Graph

Der User Behavior Graph ist im wesentlichen eine Markov Kette. Er besteht aus einer Menge von Zuständen und einer Menge von Transitionen mit normalverteilten Wartezeiten. Bei klassischen Webanwendungen wird dabei jedes HTML Dokument durch einen Zustand repräsentiert. Wenn es einen Link zwischen zwei Webseiten gibt, dann existiert eine Kante, welche die beiden korrespondieren Zustände miteinander verbindet. Jede Kante im Zustandsgraphen hat eine Übergangswahrscheinlichkeit, wobei die Summe der Übergangswahrscheinlichkeiten aller ausgehenden Kanten eines Knoten 100 Prozent ist. Mit den Wahrscheinlichkeiten wird das Verhalten eines Nutzers simuliert, denn jeder Anwender verwendet eine Webanwendung anders. Das bedeutet, die Reihenfolge in der Seiten Besucht bzw. Interaktionen durchgeführt werden, unterscheidet sich. Single-page Webanwendungen bestehen aus einem einzigen HTML Dokument, darum müssen die Zustände anders abgeleitet werden als bei klassischen Webanwendungen. In der Regel enthält das HTML Dokument ver-

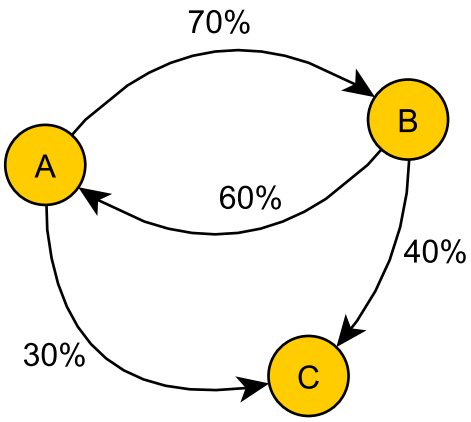

Abbildung 4.3: Beispiel einer Markov Kette

schiedene Ansichten, die unterschiedliche Daten vom Server laden. Zusätzlich gibt es Interaktionen die ein Anwender durchführen kann, die ebenfalls zu einer Abfrage führen. Die Zustände für den Zustandsgraphen ergeben sich darum aus den verschiedenen Ansichten und Interaktionen, die ein Anwender durchführen kann.

Die erwähnten Übergangswahrscheinlichkeiten für Kanten können z.B. aus alten Logdateien analysiert werden. Wenn das SUT eine vollständige Neuentwicklung ist, besteht diese Möglichkeit nicht. In diesen Fall müssen die Wahrscheinlichkeiten geschätzt werden.

4.2.2 User Behavior Graph aus Logdatei Analyse

Die analysierte Logdatei hat den folgenden Aufbau. An erster Stelle ist die IP-Adresse gespeichert. Position zwei und drei enthalten ausschließlich „-“, weil Remote Logname und Remote User nicht erfasst worden sind. Danach folgt Datum und Uhrzeit an dem die Anfrage geloggt wurde. An fünfter Stelle folgt die erste Zeile der Abfrage, sie enthält die HTTP Methode und die URL, sowie HTTP Protokollversion. An sechster Stelle steht der Status und die letzte Position erfasst die Größe der Antwort in Bytes. Die einzelnen Werte sind durch Leerzeichen getrennt.

Die Nutzer wurden durch die IP-Adressen unterschieden. Wenn bei einer IP-Adresse zwischen zwei Anfragen mehr als eine Stunde Zeit lag, wurde die zweite als neuer Nutzer gewertet.

Die alte Webseite besteht aus neun einzelnen HTML Dokumenten, die zum Auflisten von Sportarten und Sportkursen, zum Anmelden, Reservieren und Eintragen auf

Wartelisten, sowie zum Einsehen von Informationen der Sportstätten dienen. Aus jedem HTML Dokument ergibt sich ein Zustand. Zusätzlich wurden ein Start- und Endzustand definiert.

Um die Reihenfolge zu ermitteln, in der die Zustände von den Nutzern besucht werden, wurde ein Programm geschrieben. Dieses iteriert über jeden Eintrag im Log und führt eine Erkennung von Nutzer und Zustand durch. Für die Identifizierung werden IP-Adresse und URL verwendet. Das Programm zählt für jeden Zustand, wie häufig die einzelnen Transitionen zu den anderen Zuständen verwendet werden und wie viel Zeit zwischen zwei Transitionen vergeht. Aus den absoluten Werten berechnet es dann die in Tabelle 4.1 gezeigten relativen Häufigkeiten. Von den gemerkten Wartezeiten bestimmt es die Mittelwerte und deren Standardabweichung. Diese beiden Werte sind notwendig, um beim Lasttest zufällige Wartezeiten zu simulieren, die normal verteilt sind. Auf eine Abbildung als Graph wird an dieser Stelle verzichtet, weil eine übersichtliche Darstellung Aufgrund der vielen Kanten nicht möglich ist.

		Folgezustand									
		Sportlist	Courselist	Sportvenue	Course	SignUp	Man. SignUp	Confirmation	Reservation	Waitlist	End
Aktueller Zustand	Start	75%	15%	1%	2%	1%	1%	5%	0%	0%	0%
	Sportlist	5%	92%	0%	0%	0%	0%	0%	0%	0%	3%
	Courselist	13%	60%	3%	15%	0%	0%	0%	0%	0%	9%
	Sportvenue	13%	39%	19%	11%	2%	1%	0%	0%	0%	15%
	Course	10%	11%	1%	11%	52%	0%	1%	0%	7%	7%
	SignUp	1%	2%	0%	1%	6%	65%	0%	24%	0%	1%
	Man. SignUp	31%	19%	0%	6%	2%	6%	8%	0%	0%	28%
	Confirmation	16%	6%	0%	4%	1%	2%	21%	0%	0%	50%
	Reservation	32%	23%	1%	4%	2%	0%	7%	5%	0%	26%
	Waitlist	20%	34%	0%	20%	1%	1%	1%	1%	9%	13%

Tabelle 4.1: Wahrscheinlichkeiten für User Behavior Graph des alten Systems. Wahrscheinlichkeit gleich null bedeutet, dass es keine Kante gibt.

Die Werte in Tabelle 4.1 ergeben leider nur bedingt Sinn, weil dort Zustandswechsel enthalten sind, die nicht möglich sind. Das ein Nutzer z.B. als erste Seite die Sportliste, Kursliste oder Sportstättenbeschreibung aufruft ist mit Lesezeichen erklärbar aber die anderen Zustände sind nur von der Kursliste erreichbar, weil sie zur Registrierung gehören. Sie können damit nicht der erste Anlaufpunkt beim Öffnen der Hochschulsport Seite sein. Bei weiterer Betrachtung fällt auf, dass laut der Tabelle 60 Prozent der Nutzer von der Kursliste zur Kursliste navigieren. Erwartungsgemäß müsste vom Zustand Courselist sehr viel häufiger zu den Folgezuständen Sportlist oder Course gewechselt werden. Dieser Wert ist wahrscheinlich dadurch entstanden, dass der Zurück-Button des Browsers verwendet wurde, um zur Sportliste zu gelangen und die Kursliste einer anderen Sportart zu öffnen. In diesem Fall wird die Sportliste aus dem Browser-Cache geladen, wodurch kein Eintrag in der Logdatei des Webservers erfolgt. Es gibt aber durchaus einige Werte die nachvollziehbar sind. 75 Prozent der Nutzer beginnen auf der Sportliste, welches auch die Startseite des Systems ist und von dieser wechseln die meisten auf die Kursliste, was genau so

vorgesehen ist. Aus den Daten geht auch hervor, dass die meisten Nutzer nach dem Registrieren die Seite verlassen.

Der User Behavior Graph des alten Systems konnte also nicht direkt für das neue System angewendet werden. Das liegt unter anderem daran, dass Reservierungen in der Neuentwicklung noch nicht implementiert sind. Außerdem gibt es im neuen System einen Warenkorb, um mehrere Kurse gleichzeitig zu buchen. Vorher musste jeder Kurs einzeln gebucht werden, die Benutzerführung im neuen System ist also verändert. Aufgrund dessen konnten die Wahrscheinlichkeiten des alten System nur als Orientierung für das neue System dienen. In Tabelle 4.2 sind die Werte abgebildet, die für den Lasttest verwendet wurden.

		Folgezustand								
		Sportlist	Courselist	Sportvenue	Coursecart	E.P.I.	E.A.I	V.B.C.	Signup Waitlist	End
Aktueller Zustand	Start	86%	14%	0%	0%	0%	0%	0%	0%	0%
	SportList	0%	91%	0%	0%	0%	0%	0%	0%	9%
	CourseList	13%	10%	10%	44%	0%	0%	0%	13%	10%
	SportVenue	15%	24%	0%	22%	0%	0%	0%	0%	39%
	Coursecart	10%	10%	10%	0%	63%	0%	0%	0%	7%
	E.P.I.	0%	0%	0%	10%	0%	90%	0%	0%	0%
	E.A.I.	17%	0%	0%	0%	10%	0%	68%	0%	5%
	V.B.C.	13%	0%	0%	0%	0%	0%	0%	0%	87%
	Signup Waitlist	46%	0%	0%	33%	0%	0%	0%	0%	21%

Tabelle 4.2: Wahrscheinlichkeiten für User Behavior Graph des neuen Systems. Wahrscheinlichkeit gleich null bedeutet, dass es keine Kante gibt. (E.P.I. = Enter Participant Information, E.A.I. = Enter Accounting Information und V.B.C. = View Booking Confirmation)

Es wurde übernommen, dass die meisten Nutzer auf der Sportliste starten und von dieser zur Kursliste navigieren. Transitionen die keinen Sinn ergeben, wie die Eingabemaske für die Personaldaten als Startseite zu nutzen, wurden mit einer Wahrscheinlichkeit von null ausgeschlossen. Genau wie beim bestehenden System wurde davon ausgegangen, dass die meisten Nutzer nach erfolgreicher Registrierung die Seite verlassen.

Zusammenfassend lässt sich sagen, dass die Analyse der Logdatei des bestehenden System dazu gedient hat, eine grobe Orientierung für den User Behavior Graph des neuen System zu finden. Einen exakten User Behavior Graphen aus einer Logdatei zu gewinnen ist nicht möglich, weil im Webserver Log z.B. keine Aufrufe von Seiten auftauchen, die im Browser Cache sind. Bessere Ergebnisse könnten erzielt werden, indem die einzelnen Seiten mit einem Script versehen werden, dass die Nutzer erkennt und dem Webserver alle besuchten Seiten mitteilt.

4.2.3 JMeter

JMeter ist eine quelloffene Java Anwendung, die dazu entwickelt wurde Software funktionalen Lasttests zu unterziehen und deren Performance zu messen. Neben Webseiten können auch FTP Server, SOAP und REST Webservices sowie Datenbanken mit JMeter getestet werden. Es unterstützt Multithreading, um mehrere gleichzeitige Nutzer zu simulieren. Weiterhin bietet es die Funktionalität, verteilt auf mehreren Rechnern, parallel ausgeführt zu werden, um große Last auf dem SUT zu erzeugen. In JMeter wird mittels einer grafischen Nutzeroberfläche ein Testplan erstellt. In diesem ist festgelegt wie viele Threads (Nutzer) durch den Test simuliert werden, sowie die zu tätigenden Abfragen und deren Reihenfolge. Die Standardkomponenten von JMeter erlauben es, einen Nutzer durch Aufzeichnung einer Browsersitzung zu erzeugen oder ihn Komplett von Hand zu programmieren. Dafür stehen in der grafischen Oberfläche verschiedene Bausteine zur Verfügung, die hierarchisch in einer Baumstruktur im Testplan angeordnet werden. Den Standardkomponenten fehlt leider die Möglichkeit ein Markov Modell zu implementieren, welches ein probabilistisches Nutzerverhalten erzeugt. Glücklicherweise ist JMeter durch Plugins erweiterbar und der Wissenschaftler Van Hoorn hat das Plugin *Markov4Jmeter*[2] erstellt. Damit ist es möglich probabilistisches Nutzerverhalten in JMeter zu implementieren. Van Hoorn hat auch einen Artikel [VRH08] zu seinem Plugin veröffentlicht. Für den in dieser Arbeit durchgeführten Lasttest, wurde dieses Plugin verwendet.

Eine Übersicht der Implementierung ist in Abbildung 4.4 zu sehen. Der erste Knoten mit der Beschriftung „Teilnehmer ${__machineName()} Gruppe 1“ ist eine *Thread Gruppe*. In diesem ist die Anzahl der Nutzer konfiguriert und die Hochlaufzeit in der die gesamten Nutzer gestartet werden. Darunter befindet sich mit der Bezeichnung „Markov Model“ ein *Markov Session Controller*. Dieser erwartet als Parameter eine CSV Datei, mit den Werten des User Behavior Graph. Auf der gleichen Ebene befinden sich auch der Knoten für die Standardeinstellungen der HTTP Anfragen. In diesem sind Servername und Protokoll konfiguriert, die von jeder HTTP Anfrage verwendet werden. An dieser Stelle sind auch Timeouts mit 30000 Millisekunden festgelegt. Der *Header Manager* sorgt dafür, dass die Anfragen für den Server aussehen als kämen sie von einem Browser. Der *Cookie Manager* verwaltet die Cookies der simulierten Nutzer. Unterhalb des *Markov Session Controller* befinden sich neun *Markov States*.

In Abbildung 4.5 ist exemplarisch der Zustand „Enter Participant Information“ aufgeklappt. Alle Zustände haben gemeinsam, dass sie HTTP Requests besitzen, die nur beim ersten Betreten abgefragt werden, während die restlichen immer durchzuführen sind. In der Regel werden nur die HTML Teile ein einziges mal abgefragt,

[2] https://www.se.informatik.uni-kiel.de/en/research/projects/markov4jmeter

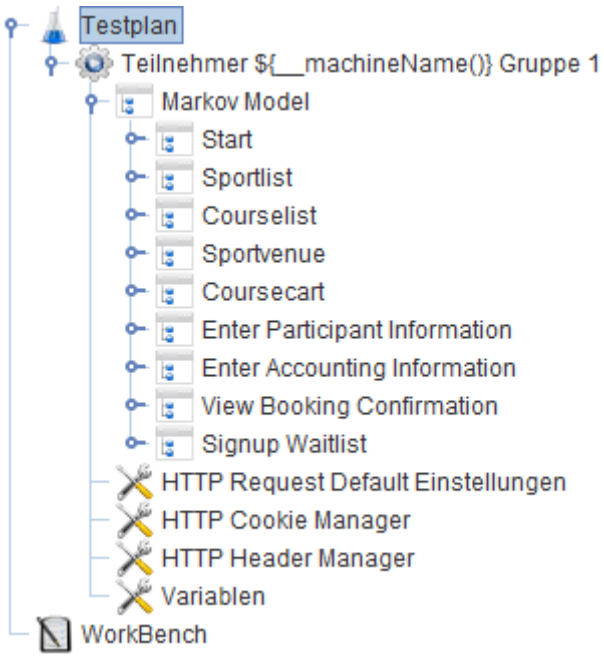

Abbildung 4.4: Aufbau des Testplan in JMeter.

da sie anschließend aus dem Browsercache geladen werden. Zu visualisierende Daten hingegen werden bei jedem Zustandswechsel neu vom Server geladen. Einmalige Abfragen befinden sich immer unter dem „Once Only Controller“, die restlichen sind mit diesem auf gleicher Ebene. Unterhalb des HTTP Requests „Enter Participant Information - Post Participant Data“ ist ein gauss'scher Zufallszeitgeber. Dieser simuliert die Wartezeit des Nutzers nach beantworteter Anfrage. Der *Content-Type application/json* Knoten sorgt in diesem Fall dafür, dass dem Server die Daten im JSON Format übermittelt werden.

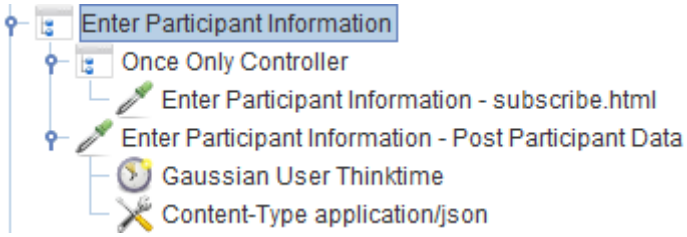

Abbildung 4.5: Aufbau eines MarkovStates in JMeter.

4.3 Anwendung der Implementierung

Der Lasttest wurde auf mehrere Rechner verteilt ausgeführt. In dieser Konfiguration arbeitet JMeter mit einem Master und einer beliebigen Anzahl von Slaves. Letztere führen den Test durch und melden die Ergebnisse an den Master zurück. Der Master dient dazu den Test auf allen Slaves gleichzeitig zu starten und die Ergebnisse zusammenzuführen. In Abbildung 4.6 ist der schematische Aufbau eines solches Tests skizziert.

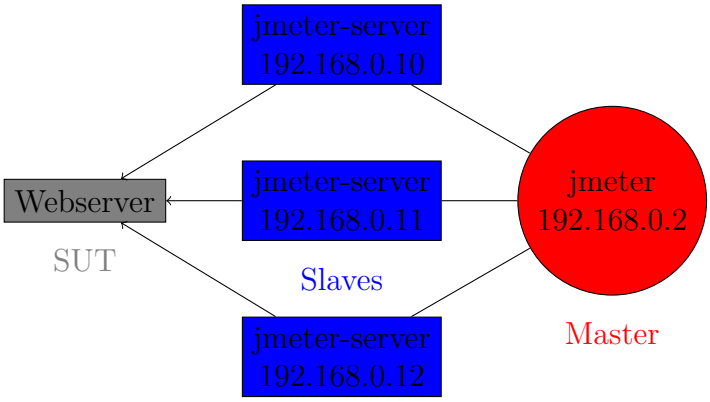

Abbildung 4.6: Übersicht von verteilten Test in JMeter.

Zunächst wurde auf den Slaves JMeter installiert und anschließend im Servermodus auf der Kommandozeile gestartet. Der implementierte Testplan ist auf dem Master gespeichert und wird von diesem an die Slaves verteilt, wenn der Test gestartet wird. Nachdem alle Slaves den Testplan abgearbeitet haben, wird vom Master zur Auswertung ein HTML Bericht generiert. Im Rahmen dieser Arbeit wurden mehrere Testreihen mit unterschiedlich vielen Nutzern durchgeführt. Dabei wurden zehn Poolrechner der Universität als Slaves verwendet. Begonnen wurde mit zehn Nutzern je Slave und einer Hochlaufzeit von 400 Sekunden. Das bedeutet, dass beim ersten Durchlauf alle 40 Sekunden zehn weitere Nutzer gestartet worden sind. Mit jedem weiteren Testdurchgang wurde die Nutzeranzahl erhöht, während die Hochlaufzeit gleich blieb. Auf diese Weise wurde die maximale Belastbarkeit des Systems ermittelt.

5 Optimierungen

In diesem Kapitel wird erklärt, welche Maßnahmen ergriffen wurden, um die Performance des Systems zu verbessern.

5.1 Reduktion der Anzahl der Anfragen

Für den Aspekt gleichzeitige Nutzer ist die Anzahl der Anfragen, die während einer Sitzung von der Webanwendung an den Server gestellt werden, von signifikanter Bedeutung. Denn je weniger Anfragen pro Nutzer gestellt werden, desto geringer ist die entstehende Last auf dem Webserver. Die niedrigere Last führt wiederum dazu, dass mehr Anwender die Webanwendung gleichzeitig verwenden können. Die Hochschulsport Webseite hat eine REST Schnittstelle implementiert, die es ermöglicht Ressourcen mittels CRUD Operationen zu verwalten. Vor den Optimierungen waren einige Abfragen von Daten sehr ineffizient. Als explizites Beispiel sei hier die Seite zur Auflistung der Kurse einer Sportart erwähnt. In Abbildung 5.1 befindet sich ein Screenshot einer solchen Kursliste.

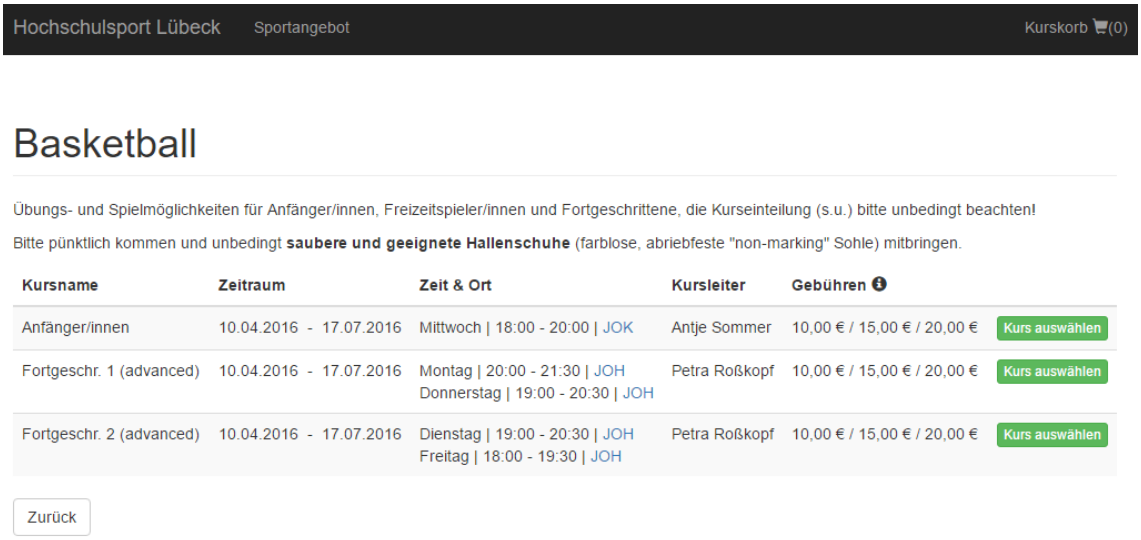

Abbildung 5.1: Screenshot der Kursliste im neu entwickelten Hochschulsport System.

Die Tabelle hat für jeden Kurs eine Zeile. Jedoch setzen sich die Informationen einer Zeile aus mehreren Ressourcen zusammen. In diesem konkreten Beispiel stammen die Gebühren aus einer weiteren Ressource, die mit dem Kurs in einer 1:n Beziehung steht, weil ein Kurs mehrere Gebühren haben kann, dass ist darin Begründet, dass z.B. interne und externe Teilnehmer unterschiedliche Gebühren bezahlen müssen. Vor den Optimierungen wurden auf dieser Seite vom Client zuerst eine Liste aller Kurse der Sportart abgefragt. Anschließend wurden zu jedem Kurs die in Relation stehenden anderen Ressourcen vom Server geladen. In Abbildung 5.2 ist die Reihenfolge der Abfragen im Detail zu sehen.

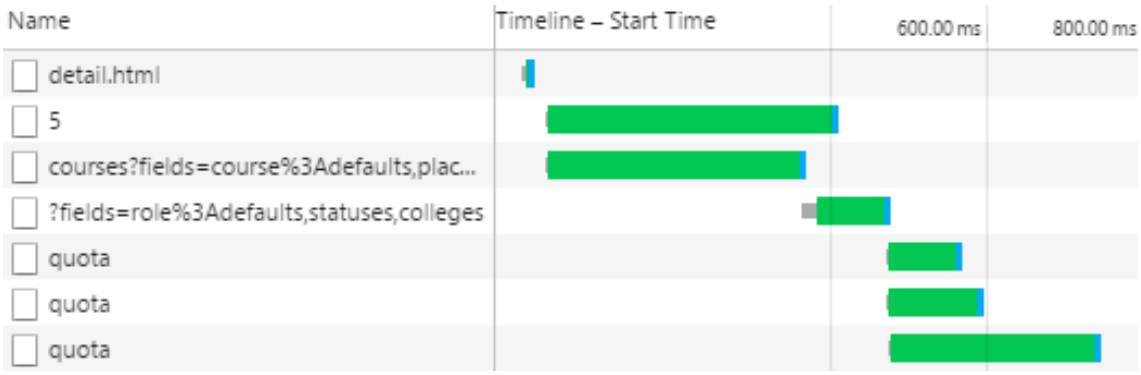

Abbildung 5.2: Übersicht über die Reihenfolge der einzelnen Abfragen der Kursliste ohne Optimierungen. Zu sehen ist, dass die Quota (Gebühren) drei mal abgefragt werden. Der Screenshot stammt aus den Chrome Developer Tools.

Alleine um die Tabelle in Abbildung 5.1 mit Daten zu füllen, wurden also vier Abfragen durchgeführt. Eine für die drei Kurse und je Kurs noch mal eine für die Gebühren. Bei Sportarten mit mehr Kursen steigt auch die Anzahl der Anfragen. Das ist sehr ineffizient, weil für jede Abfrage eine neue separate Verbindung zum Server aufgebaut wird, d.h. es werden mehrere Handshakes durchgeführt. Die Schnittstelle bietet die Möglichkeit, anzuzeigende Daten schneller und Ressourcensparender vom Server zu laden. An die URLs können URL-Parameter angehängt werden. Mit dem Parameter *fields* kann spezifiziert werden, welche Felder einer Ressource der Server liefern soll. Darüber können auch zueinander in Beziehung stehende Ressourcen angegeben werden. In dem konkreten Beispiel betrifft das die Kurse und Gebühren. Auf diese Weise konnte das Laden der Tabelle auf eine einzelne Abfrage beschränkt werden. Diese Optimierung ergibt auch dann noch Sinn, wenn der Server auf das HTTP/2 Protokoll [BPT15] umgestellt wird, weil auf diese Weise die Header der einzelnen Abfragen vollständig eingespart werden.

5.2 Datendurchsatz je Anfrage

Weitere wichtige Stellschrauben zum Optimieren sind, die Anfragen selbst. Sie sollten so präzise wie möglich sein, um jedes unnötige Byte einzusparen. Wenn eine Ressource vom Server geladen wird, ohne Angabe weiterer URL-Parameter, dann liefert die Schnittstelle alle Felder der Ressource und mit ihr in 1:1 Beziehung stehende Ressourcen mit deren Feldern zurück. Als konkretes Beispiel soll noch einmal die Kursliste aus Abbildung 5.1 dienen. Ein Kurs gehört zum Beispiel zu einer Sportart. Das bedeutet, bei einer Abfrage ermittelt der Server für einen Kurs auch die dazugehörige Sportart. Technisch betrachtet sind das Unterabfragen im SQL. Einerseits wird die Antwort unnötig groß, weil sie Informationen enthält, die auf der Seite nicht dargestellt werden. Andererseits werden auf dem Server Ressourcen unnötig verbraucht. Dieser Umstand kann durch explizite Angabe der gewünschten Felder im *fields* URL-Parameter vermieden werden. Durch diese Maßnahme werden die Antwortzeiten kürzer und der Traffic sinkt.

5.3 Datenhaltung

Neben dem Verfeinern und Reduzieren von Abfragen sind auch Datenhaltung und Caches wichtige Kriterien, um die Performance des gesamten System zu steigern. In modernen Webanwendungen ist für Entwickler neben Cookies, auch ein lokaler Webspeicher im Browser verfügbar. Dieser bietet eine höhere Privatsphäre als Cookies[WP12] und wird nicht bei jeder Abfrage an den Server übertragen. Zusätzlich ist sein Speicherlimit größer mit mindestens 5MB. Es bietet sich an, diesen als Cache zu verwenden für Daten die sich während einer Sitzung nicht ändern. Beim Buchungsprozess der Hochschulsport Webseite gibt es mehrere solche Daten, die auf verschiedenen Seiten verwendet werden. Diese wurden dafür immer bei Bedarf vom Server geladen. Im Zuge der Optimierungen wurde das geändert, die Daten werden jetzt im lokalen Webspeicher vorgehalten. Bei jeder Verwendung der Daten wird zunächst geprüft, ob diese bereits im Speicher sind und aus diesem geladen. Ist das nicht der Fall werden sie vom Server abgefragt und gespeichert. Auf diese Weise konnte die Kommunikation zwischen Server und Client reduziert werden.

Die Hochschulsport Webseite hat einen Warenkorb, in den ein Nutzer Kurse zum Buchen ablegen kann. Die Idee dahinter ist, dass auf diese Weise mehrere Kurse ausgewählt werden können und die Daten zur Buchung nur ein einziges mal eingegeben werden müssen. Ursprünglich wurde der Warenkorb auf dem Server zeitlich befristet für einen Nutzer gespeichert, wobei die Wiedererkennung des Nutzers anhand von Cookies stattfand, weil die Hochschulsport Webseite keine Anmeldung bereitstellt. Das bedeutet der Warenkorb ist immer nur auf einem Endgerät verfügbar.

Aus diesem Grund bot es sich an, den Warenkorb vollständig auf den Client auszulagern. Der Nachteil, dass der Warenkorb nur auf einem Endgerät zur Verfügung steht blieb dadurch erhalten. Die Auslagerung auf den Client bedeutet, dass vom Nutzer ausgewählte Kurse im lokalen Webspeicher des Browsers vorgehalten werden. Dies widerspricht zunächst dem Schema aus Abbildung 2.1, nachdem die Datenhaltung auf dem Server stattfindet. Beim Warenkorb handelt es sich aber nur um temporäre Daten, die keine dauerhafte Speicherung benötigen und die nur während der Sitzung des Nutzers verwendet werden. Bei einer Buchung werden dann ausschließlich die Identifikationsnummern der Kurse an den Server übertragen. Durch die Auslagerung ergeben sich mehrere Vorteile. Zum einen wird dadurch die Kommunikation mit dem Server reduziert. Vorher löste das Hinzufügen eines Kurses zum Warenkorb eine Abfrage aus und beim Öffnen wurde er vom Server geladen. Des weiteren vereinfachte sich die Datenhaltung im Server.

6 Evaluierung

Im ersten Teil dieses Kapitels werden die Antwortzeiten der Software vor und nach den Optimierungen verglichen, um die Auswirkungen der Verbesserungen zu evaluieren. Der zweite Teil befasst sich mit der Auswertung des probabilistischen Lasttests.

6.1 Antwortzeiten nach der Optimierungen

Im folgenden werden die Antwortzeiten der Seiten des Teilnehmerbereichs im Detail betrachtet. Dieser Bereich wird von vielen Nutzern gleichzeitig besucht, deswegen sind die Antwortzeiten seiner Seiten besonders kritisch. Ermittelt wurden die Zeiten mit JMeter. Dafür wurde der Aufruf der Seiten in JMeter implementiert. Anschließend wurden sequentiell 100 Abfragen je Seite durchgeführt und von den gemessenen Zeiten der Mittelwert bestimmt. Verglichen werden hier die Softwarestände zu Beginn und Fertigstellung dieser Arbeit.

Seite / Interaktion	ohne Optimierungen	mit Optimierungen
Sportartenliste	444	562
Kursliste	779	335
Kurs hinzufügen	165	0
Warenkorb	224	33
Eingabe Personendaten	303	40
Senden Personendaten	239	123
Eingabe Bankdaten	475	143
Senden Bankdaten	104	171
Bestätigung	186	69

Tabelle 6.1: Vergleich der durchschnittlichen Wartezeiten von Seiten und Interaktionen ohne und mit Optimierungen. Die Zeiten sind in Millisekunden angegeben.

In Tabelle 6.1 werden die durchschnittlichen Antwortzeiten direkt miteinander verglichen. Es ist zu sehen, dass sich fast alle Ladezeiten signifikant verbessert haben, jedoch gibt es auch eine Seite und eine Interaktion deren Zeiten sich verschlechtert haben. Das sind die Sportartenliste und das Senden der Bankdaten. Grund dafür ist, dass die Software parallel zu den Optimierungen weiterentwickelt wurde. Das Senden der Bankdaten ist langsamer, weil die Validierung der Daten auf dem Server umfangreicher geworden ist. Die Sportartenliste lädt etwas langsamer, weil diese zusätzliche Anfragen stellt, um zum Beispiel den aktuellen Zeitraum anzuzeigen. Vorher wurde statisch immer der gleiche angezeigt. Aus Tabelle 6.2 kann man entnehmen, dass die Anfrage für den Warenkorb komplett entfällt. Dafür sind Abfragen für Zeitraum und Rollen dazugekommen. Insgesamt wird jetzt also beim Starten der Anwendung eine Anfrage mehr durchgeführt. Die signifikantesten Verbesserungen in Bezug auf die Ladezeiten treten beim Hinzufügen eines Kurses und beim Öffnen des Warenkorbs auf. Dies wurde dadurch erreicht, dass der Warenkorb im Client gespeichert wird. Somit ist keine Kommunikation mit dem Server notwendig, wenn ein Kurs in den Warenkorb gelegt wird. Darum ist die Ladezeit für diese Interaktion, entsprechend mit Null in der Tabelle angegeben. Beim ersten Öffnen des Warenkorbs wird ein HTML Dokument vom Server geladen. Danach hat auch diese Seite keine Ladezeit mehr, weil der HTML Code dann aus dem Browser Cache geladen wird. Gleiches gilt für die Seite zur Eingabe der Personaldaten.

In Tabelle 6.2 ist für vier Seiten aufgelistet, welche Anfragen je Seite an den Server gestellt werden. Aufgrund der Größe der Tabelle wurde darauf verzichtet, dies für alle Seiten der Webanwendung aufzulisten. Anhand der gewählten Beispiele wird klar, dass die Seiten möglichst wenige Anfragen stellen.

Einzige Ausnahme ist die Sportartenliste. Diese lädt zum Beispiel Rollen. Eine Rolle der Hochschulsport Webanwendung besteht aus Name, Beginn der Registrierung, Hochschule und Status. Sie werden im Buchungsprozess an mehreren Stellen benötigt und deswegen von dieser Seite in den lokalen Webspeicher abgelegt.

Die Kursliste ist ein gutes Beispiel dafür, wie Abfragen zusammengefasst wurden. Vorher benötigte die Seite die sechs in der Tabelle aufgelisteten Abfragen, wobei die letzte für jeden gefundenen Kurs einzeln durchgeführt wurde. Das bedeutet bei drei Kursen schon acht Abfragen insgesamt. Die Abfrage der Kurse wurde dahingehend optimiert, dass die Quota, das sind die Preise der Kurse, in der Antwort enthalten sind. Weiterhin wurde dafür gesorgt nur Informationen abzufragen, die auch visualisiert werden. Die Abfrage der Kurse konnte auf diese Weise auf ungefähr 60% der ursprünglichen Dauer reduziert werden. Die Abfrage des Sports dient dazu, die Beschreibung zu laden. Diese konnte auf ca. 50% der vorherigen Ladezeit optimiert werden.

Weiterhin wird aus der Tabelle 6.2 ersichtlich, dass für den Warenkorb und die Eingabemaske der Personaldaten nur noch die HTML Dokumente geladen werden. Die

	Vorher		Nacher	
Seite / Interaktion	Datei / XHR	Zeit	Datei / XHR	Zeit
Sportartenliste	participants_index.html	57	participants_index.html	48
	bundle.js	92	participants.js	86
	navigation.html	33	navigation.js	31
	list.html	32	XHR Zeitraum	45
	XHR Kurskorb	85	fonts	33
	XHR Sportarten	137	XHR Rollen	193
	fonts	40	list.html	43
			XHR Sportarten	147
Kursliste	detail.html	31	detail.html	46
	XHR Sport	91	XHR Sport	48
	XHR Kurse	479	XHR Kurse	289
	XHR Rollen	120		
	XHR Quota je Kurs	149		
Kurskorb	coursecart.html	40	coursecart.html	33
	XHR Kurskorb	93		
	XHR Rollen	91		
Eingabe Personaldaten	subscribe.html	34	subscribe.html	40
	XHR Regitrierungsrollen	184		
	XHR Zeiträume	73		
	XHR Rollen	94		
	XHR Hochschulen je Rolle	85		
	XHR Status je Rolle	75		

Tabelle 6.2: Auflistung einzelner Serveranfragen je Seitenaufruf für vier Beispielseiten. HTML Dokumente werden nur beim ersten Aufruf der Seite geladen.

beiden Seiten verwenden jetzt ausschließlich Daten aus dem lokalen Webspeicher, die dort von vorherigen Seitenaufrufen gespeichert wurden. Aufgrund dessen verursachen diese auf dem Server minimale Last. Die Rollen, welche der Warenkorb benötigt, wurden bereits beim Starten der Anwendung geladen. Gleiches gilt für die Zeiträume, Rollen, Hochschulen und Status der Personaldaten Eingabemaske.

6.2 Evaluation des Lasttests

Wie in Abschnitt 4.3 erwähnt wurden mehrere Testreihen durchgeführt. Bei jedem der Tests wurde eine bestimmte Nutzerzahl in der festen Hochlaufzeit von 400 Sekunden simuliert. Jeder Nutzer führte für mehrere Minuten Anfragen aus. Die Ge-

samtlaufzeit der einzelnen Tests lag zwischen 8 und 15 Minuten. Aus Tabelle 6.3 können die Nutzerzahlen der einzelnen Testläufe entnommen werden. Sie gibt insbesondere Aufschluss darüber, wie viele Fehler aufgetreten sind. Der häufigste Fehler ist, die Serverantwort mit dem HTTP Status Code 503, sie bedeutet, dass der Dienst vorübergehend nicht verfügbar ist. Am zweithäufigsten treten Zeitüberschreitungen auf, der Grenzwert liegt bei 30 Sekunden. Das Übertreten der Zeitgrenze führt dazu, dass der Nutzer die letzte Anfrage wiederholt oder eine Neue durchführt. Aufgrund dessen gab es Nutzer, die versuchten sich doppelt für einen Kurs zu registrieren, was ebenfalls zu einem Fehler führt, weil dies vom System nicht zugelassen wird. Für die Bestimmung des APDEX Wertes wurde als Toleranzgrenzwert 1,5 Sekunden und als Frustrationsgrenzwert 3 Sekunden verwendet.

Nutzer	Buchungen	Fehler	APDEX	Dauer
100	53	0	0,995	7 min. 57 sek.
150	71	0	0,991	8 min. 14 sek.
200	99	0	0,956	10 min. 29 sek.
250	131	0	0,706	11 min. 34 sek.
300	155	6	0,625	12 min. 19 sek.
350	184	325	0,587	14 min. 32 sek.
400	223	1218	0,522	13 min. 54 sek.

Tabelle 6.3: Übersicht von den Ergebnissen der einzelnen Testreihen. Für jede Testreihe wurden 10 Slaves verwendet, die Anzahl der Nutzer ist hier über alle Slaves summiert.

Bei 200 Nutzern waren die Antwortzeiten vertretbar, das spiegelt sich auch im APDEX Wert wieder. Laut diesem wären etwa 95% der Nutzer zufrieden mit der Performance des Systems. Es gab nur vereinzelte Abfragen mit einer Antwortzeit über 10 Sekunden.

Durch die Erhöhung auf 250 Nutzer kam es zwar nicht zu Zeitüberschreitungen oder anderen Fehlern, jedoch stiegen die Antwortzeiten beträchtlich, was sich auf den APDEX niederschlägt. Nur etwa 70% der Nutzer wären noch zufrieden. Die Antwortzeiten betrugen in diesem Testlauf teilweise über 20 Sekunden.

In den darauf folgenden Testreihen stiegen die Antwortzeiten auf über 30 Sekunden an. Bei der Testreihe mit 300 Nutzern ist das an den sechs aufgetretenen Fehlern zu erkennen, dies waren ausschließlich Zeitüberschreitungen. Schließlich begann mit 350 simulierten Nutzern die Webanwendung zu überlasten. Das System antwortete in 278 Fällen mit dem HTTP Status Code 305, dass der Dienst vorübergehend nicht erreichbar ist. Diese Zahl stieg in der letzten Testreihe sogar auf 868 an. Auf die Durchführung weiterer Tests mit mehr Nutzern wurde verzichtet, weil die Webanwendung bei der letzten Testreihe bereits viele Anfragen aufgrund der Überlastung zurückgewiesen hat.

In Abbildung 6.1 ist die Anzahl der Abfragen je Sekunde für den Lasttest mit 200 Nutzern grafisch dargestellt. Es gibt wie beim alten System einige Spitzen mit 24 Abfragen je Sekunde. Im Mittel treten ca 10 Abfragen je Sekunde auf, was im Vergleich zum alten System deutlich höher ist. Von den 200 Testnutzern haben sich 99 in einen Kurs eingeschrieben und weitere 60 haben sich erfolgreich auf eine Warteliste eingetragen, in einem Zeitraum von 10 Minuten und 29 Sekunden. Damit sind die Anforderungen für den Hochschulsport erfüllt.

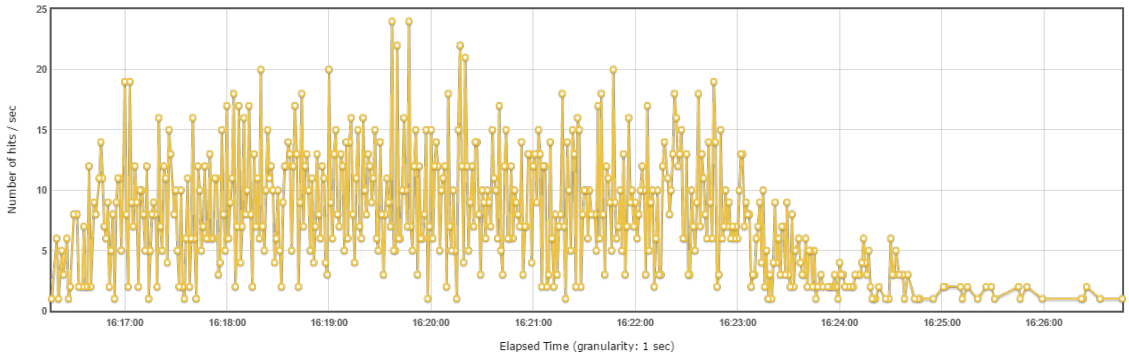

Abbildung 6.1: Anzahl der Anfragen je Sekunde bei 200 Nutzern.

7 Zusammenfassung und Ausblick

Dieses Kapitel enthält eine Zusammenfassung der Arbeit und gibt einen Ausblick, welche weiteren Optimierungen dazu führen können, die Performance weiter zu verbessern.

7.1 Zusammenfassung der Ergebnisse

In dieser Masterarbeit wurde für die neu entwickelte Webseite des Hochschulsports der Universität zu Lübeck ein Lasttest durchgeführt. Ziel des Tests war es sicherzustellen, dass die neue Seite den Nutzerzahlen des bisherigen Systems stand hält. Die neue Seite basiert auf modernen Internettechnologien wie AngularJS, welches ein Javascript Framework ist, mit dessen Hilfe dynamische Webseiten erstellt werden können. Eine weitere Besonderheit der Seite ist, dass sie eine Single-page Webanwendung ist. Das bedeutet, sie besteht aus einem einzigen HTML Dokument, dessen Inhalte zur Laufzeit dynamisch nachgeladen werden. In dieser Arbeit wird dabei speziell auf die Besonderheiten von solchen Webanwendungen gegenüber klassischen Webanwendungen bei Lasttests eingegangen. Um eine möglichst realistische Last auf dem Testsystem zu erzeugen, wurde das User Behavior Graph Model verwendet. In diesem werden Seiten und Interaktionen als Zustände modelliert. Existiert zwischen zwei Seiten ein Link, haben die Zustände im Modell eine Transition mit einer Eintrittswahrscheinlichkeit. Um die Zeit zu simulieren, in der ein Nutzer nachdenkt oder Text liest, gibt es nach jeder Transition eine normalverteilte Wartezeit. Der Lasttest startet dann in einem Startzustand und anhand der Wahrscheinlichkeiten der Transitionen wird in einen anderen Zustand gewechselt. In jedem Zustand werden dann die entsprechenden Anfragen durchgeführt. Der Test läuft solange, bis ein definierter Endzustand erreicht wird. Für die Durchführung des Lasttests wurde das quelloffene Framework JMeter verwendet. Dieses ist das einzige, welches über ein Plugin das User Behavior Graph Modell unterstützt.

Neben dem Lasttest wurden Methoden zur Optimierung von Webanwendung aufgezeigt. Darunter Fallen Punkte wie das Reduzieren von Server Anfragen, um die Last zu verringern, aber auch Schritte, um die Effizienz der Anfragen zu steigern. Weiterhin wurde erklärt, wie Daten für die spätere Wiederverwendung mittels moderner

Entwicklungen wie lokalen Webspeichern vorgehalten werden können. Lokaler Webspeicher wird von allen gängigen Browsern zur Verfügung gestellt.

Die in dieser Arbeit besprochenen Optimierungen wurden in der erwähnten Hochschulsport Webanwendung umgesetzt. Im Anschluss an die Umsetzung folgte ein Vergleich der Softwareversionen mit und ohne Optimierungen in Bezug auf ihre Antwortzeiten. Das Ergebnis zeigte eine signifikante Verbesserung der Antwortzeiten. Weiterhin stellte sich heraus, dass die neue Hochschulsport Webanwendung mit der Nutzerlast ihres Vorgängers durchaus zurecht kommt.

7.2 Ausblick

Es gibt weitere Möglichkeiten, die Performance der Software zu steigern. Beispielsweise könnte in Betracht gezogen werden von Rails auf ein anderes Framework zu wechseln. Rails ist gut, um eine Webapplikation in kurzer Zeit zu entwickeln. Es hat jedoch eine Schwäche. Für jede Verbindung, die zum Server aufgebaut wird, verwendet es einen eigenen Thread, wobei die maximale Anzahl von Threads limitiert ist. Wenn in Thread eine I/O Operation durchführt, zum Beispiel das Laden von Daten aus einer Datenbank, dann wird dieser pausiert bis die Daten fertig geladen sind. Man spricht hierbei auch von *Blocking I/O*. In der Regel benötigt das Verarbeiten von Daten sehr viel weniger Zeit als das Laden. Es könnten also mehr Daten und Verbindungen verarbeitet werden, wenn asynchrone I/O Operationen verwendet wird. Bekannte Frameworks die genau das machen sind zum Beispiel Play[1], welches sowohl in Scala als auch in Java implementiert ist oder Nodejs[2], welches eine Javascript Laufzeitumgebung ist, die Event-Basiert ist und ein Non Blocking I/O Modell verwendet.

Weiterhin können die HTML Dokumente in Bezug auf ihre Dateigröße verbessert werden. Beim Entwickeln werden die verschiedenen HTML Tags eingerückt, um mehr Übersichtlichkeit zu erlangen. Die Einrückungen haben absolut keinen Effekt bei der Darstellung der Seite selbst. Jedes verwendete Leerzeichen und Tab sind aber in der Datei codiert und werden beim Laden mit übertragen. Durch entfernen der Einrückungen kann die Dateigröße reduziert werden. Obwohl HTML Dateien gzip komprimiert werden, um das Laden zu beschleunigen, kann durch Entfernen der HTML Einrückungen, laut [Kri14] die Dateigröße des mit gzip komprimierten HTML Dokuments, zwischen 6 - 16 Prozent reduziert werden. Auf diese Weise könnten die Ladezeit beim Starten der Webanwendung weiter verbessert werden. Durchgeführt werden könnte dies zum Beispiel mit dem Node Packet html-minifier[3].

[1] https://www.playframework.com/
[2] https://nodejs.org/en/
[3] https://www.npmjs.com/package/html-minifier

Darüber hinaus besteht die Möglichkeit, den Server auf das HTTP/2 Protokoll umzustellen. Es bietet einige Vorteile gegenüber HTTP1.1. Derzeitig baut der Browser für jede Anfrage eine eigene Verbindung zum Server auf, jedoch nicht mehr als sechs gleichzeitig. Die Hochschulsport Webseite benötigt bei den meisten Seiten mehrere Verbindungen aufgrund der einzelnen XHR Abfragen. Je Abfrage wird eine neue Verbindung aufgebaut, was eine weitere Paketumlaufzeit zur Folge hat. Ferner werden auf dem Server für die zusätzlichen Verbindungen weitere Ressourcen belegt. HTTP/2 ermöglicht Pipelining, d.h. mehrere HTTP Anfragen können über eine einzelne TCP Verbindung übertragen werden. Damit würden sich die zusätzlichen Paketumlaufzeiten erübrigen. Zudem erlaubt HTTP/2 dem Webserver mittels HTTP Push dem Browser das geforderte HTML Dokument mitsamt weiteren Ressourcen wie Bildern Javascript und CSS Dateien mitzusenden. Davon würde besonders die Startzeit der Webanwendung profitieren, weil dann die participants_index.html, participants.js und die navigation.html aus der Sportliste in Tabelle 6.2 auf einmal vom Server geliefert werden könnten, statt sequentiell, wie es jetzt der Fall ist.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
FTP	File Transfer Protocol
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SUT	System unter Test, engl. *system under test*
SQL	Structured Query Language
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
XHR	XML Http Request

Literaturverzeichnis

[AL93] Avritzer, Alberto ; Larson, Brian: Load testing software using deterministic state testing. In: *ACM SIGSOFT Software Engineering Notes* 18 (1993), Nr. 3, S. 82–88. http://dx.doi.org/10.1145/174146.154244. – DOI 10.1145/174146.154244. – ISBN 0–89791–608–5

[AW95] Avritzer, Alberto ; Weyuker, Elaine J.: The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 9, S. 705–716. http://dx.doi.org/10.1109/32.464549. – DOI 10.1109/32.464549. – ISBN 00985589 (ISSN)

[BPT15] Belshe, Mike ; Peon, Roberto ; Thomson, M: RFC 7540: hypertext transfer protocol version 2 (HTTP/2). In: *Internet Engineering Task Force (IETF)//BitGo, Google Inc.//May* (2015)

[DGH+06] Draheim, Dirk ; Grundy, John ; Hosking, John ; Lutteroth, Christof ; Weber, Gerald: Realistic load testing of Web applications. In: *Conference on Software Maintenance and Reengineering (CSMR'06)*, IEEE, 2006. – ISBN 0–7695–2536–9, 11 pp.–70

[JH15] Jiang, Zhen M. ; Hassan, Ahmed E.: A Survey on Load Testing of Large-Scale Software Systems. In: *IEEE Transactions on Software Engineering* 41 (2015), Nr. 11, S. 1091–1118. http://dx.doi.org/10.1109/TSE.2015.2445340. – DOI 10.1109/TSE.2015.2445340. – ISSN 00985589

[Kri14] Kristensen, Mads: *Effects of GZipping vs. minifying HTML files.* http://madskristensen.net/post/effects-of-gzipping-vs-minifying-html-files, 2014. – Accessed: 2016-09-28

[LR01] Leff, Avraham ; Rayfield, James T.: Web-application development using the model/view/controller design pattern. In: *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International* IEEE, 2001, S. 118–127

[MFB+07] Meier, J.D. ; Farre, Carlos ; Bansode, Prashant ; Barber, Scott ; Rea, Dennis: Load-Testing Web Applications. In: *Performance Testing Guidance for Web Applications* (2007), S. 221. ISBN 9780735625709

[Rit10] Ritzer, Uwe: *Chaostage in Fürth*. http://www.sueddeutsche.de/wirtschaft/quelle-ausverkauf-chaostage-in-fuerth-1.136930, 2010. – Accessed: 2016-10-10

[SS13] Stolberg, Sheryl G. ; Shear, Michael: *Inside the Race to Rescue a Health Care Site, and Obama*. http://www.nytimes.com/2013/12/01/us/politics/inside-the-race-to-rescue-a-health-site-and-obama.html?_r=0, 2013. – Accessed: 2016-10-10

[VRH08] Van Hoorn, André ; Rohr, Matthias ; Hasselbring, Wilhelm: Generating probabilistic and intensity-varying workload for web-based software systdems. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5119 LNCS (2008), Nr. June, S. 124–143. http://dx.doi.org/10.1007/978-3-540-69814-2-9. – DOI 10.1007/978–3–540–69814–2–9. – ISBN 3540698132

[Wor16] World Wide Web Consortium: *Web Storage (Second Edition)*. https://www.w3.org/TR/webstorage/, 2016. – Accessed: 2016-11-28

[WP12] West, William ; Pulimood, S. M.: Analysis of Privacy and Security in HTML5 Web Storage. In: *J. Comput. Sci. Coll.* 27 (2012), Januar, Nr. 3, 80–87. http://dl.acm.org/citation.cfm?id=2038772.2038791. – ISSN 1937–4771