

Masterarbeit

Eine Cloud-basierte Software-Plattform für den Betrieb
horizontal skalierbarer Web-Anwendungen

Andreas Wolke

Studienrichtung: Informatik

Studienschwerpunkt: Sichere Netze

Verfasser der Arbeit:

Andreas Wolke
Petristraße 5
86405 Meitingen
Tel.: +49 (0)821 4301200
jacksonicson@gmail.com

Hochschule Augsburg:

Hochschule für angewandte Wissen-
schaften
Fachhochschule Augsburg
An der Fachhochschule 1
86161 Augsburg

Tel.: +49 (0)821 5586-0
Fax: +49 (0)821 5586-3222
<http://www.hs-augsburg.de>
info@hs-augsburg.de

Erstprüfer: Prof. Dr. Gerhard Meixner

Zweitprüfer: Prof. Dr. Jürgen Scholz

Abgabe der Arbeit: 20.04.2010

Fakultät für Informatik:
Friedberger Straße 2a
(Gebäude J)
86161 Augsburg

Postanschrift:
Postfach 11 06 05
86031 Augsburg

Tel.: +49 821 5586-3450
Fax: +49 0821 5586-3499
inf@informatik.fh-augsburg.de

Masterarbeit

Eine Cloud-basierte Software-Plattform für den Betrieb horizontal skalierbarer Web-Anwendungen

Andreas Wolke

20.04.2010

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorgelegte Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt, keine Urheberrechtsschutz-Verletzungen begangen sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Andreas Wolke, 20.04.2010

Kurzfassung

Cloud Computing ist eine aufstrebende Technologie, die sich besonders für die Entwicklung und den Betrieb von Web-Anwendungen eignet. Gründe dafür sind u. A. eine einfache Entwicklung, Skalierbarkeit und das einfache und übersichtliche Pay-per-Use-Preismodell. Während der Entwicklung einer Web-Anwendung zur Verwaltung und Verarbeitung von Bilddaten traten allerdings die Grenzen aktueller Platform-as-a-Service-Angebote zutage. Somit entstand die Notwendigkeit einer neuen Cloud-Plattform namens TwoSpot. Diese Arbeit befasst sich mit den grundlegenden Design- und Architekturfragen von TwoSpot. Zunächst werden dazu die Architektur und der Leistungsumfang bestehender Platform-as-a-Service-Angebote analysiert. Besonderes Augenmerk wird dabei auf die horizontale Skalierbarkeit und die Mechanismen zur persistenten Datenspeicherung gelegt. Für die Realisierung der Plattform folgt eine Evaluation weiterer Kerntechnologien. Auf Basis dieser Ergebnisse wird dann, wieder unter dem Gesichtspunkt der horizontalen Skalierbarkeit, die Architektur der TwoSpot-Plattform vorgestellt. Abgerundet wird die Arbeit durch erste Tests, die sich mit der Performance und Skalierbarkeit von TwoSpot in einer realistischen Serverumgebung befassen.

Inhalt

1.	Hintergrund und Motivation	1
2.	Cloud Computing.....	4
2.1.	Klassifikationsschema	4
2.1.1.	Infrastructure-as-a-Service	4
2.1.2.	Platform-as-a-Service.....	6
2.1.3.	Software-as-a-Service.....	7
2.2.	Preismodell	8
2.3.	Sicherheitsaspekte.....	9
2.4.	Vor- und Nachteile.....	9
3.	Evaluation	12
3.1.	Google App Engine	12
3.1.1.	Architektur	12
3.1.2.	Datastore.....	14
3.1.3.	Memchache.....	15
3.1.4.	Weitere Dienste.....	15
3.1.5.	Development Environment	16
3.1.6.	Horizontale Skalierbarkeit.....	16
3.1.7.	Zusammenfassung.....	17
3.2.	Joyent Smart.....	18
3.2.1.	Architektur	18
3.2.2.	Datenbank.....	19
3.2.3.	Blob-Speicher.....	20
3.2.4.	Queue Worker	20
3.2.5.	Weitere Dienste.....	20
3.2.6.	Development Environment	20
3.2.7.	Horizontale Skalierbarkeit.....	21
3.2.8.	Zusammenfassung.....	21
3.3.	Heroku	21
3.3.1.	Datenbank	24
3.3.2.	Amazon Web Services	24

3.3.3. Horizontale Skalierbarkeit.....	24
3.3.4. Development Environment	24
3.3.5. Zusammenfassung.....	25
3.4. Windows Azure	25
3.4.1. Blob-Storage	26
3.4.2. Table-Storage.....	27
3.4.3. Queue-Storage.....	27
3.4.4. Weitere Dienste.....	28
3.4.5. Development Environment	28
3.4.6. Horizontale Skalierbarkeit.....	28
3.4.7. Zusammenfassung.....	29
3.5. AppScale.....	30
3.5.1. Weitere Dienste.....	32
3.5.2. Horizontale Skalierbarkeit.....	32
3.5.3. Zusammenfassung.....	33
3.6. Ergebnisse	33
3.6.1. Execution Environment.....	33
3.6.2. Horizontale Skalierung	35
3.6.3. Datenspeicherung	35
3.6.4. Monitoring-Mechanismen	36
3.6.5. Weitere Dienste.....	36
3.6.6. TwoSpot	36
4. Technologien	38
4.1. Anwendungsserver.....	38
4.2. Datenbank.....	39
4.2.1. MongoDB	42
4.2.2. Bigtable, Hypertable und HBase.....	45
4.2.3. Cassandra.....	48
4.2.4. Vergleich	50
4.3. Verteilte Dateisysteme.....	52
4.3.1. Hadoop Distributed File System	52
4.3.2. Cloudstore.....	55

4.3.3. MogileFS	55
4.3.4. Vergleich	57
5. TwoSpot	59
5.1. ZooKeeper	60
5.2. AppServer	60
5.3. Controller	62
5.4. Master	65
5.5. Frontend	66
5.6. Sicherheitsaspekte	68
5.7. Datenspeicher	71
5.8. Horizontale Skalierung	78
5.9. Benutzersitzungen	81
5.10. Anwendungsentwicklung	82
5.11. Portal-Anwendung	83
6. Skalierungs- und Performance-Tests	86
7. Ausblick	91
8. Abschluss	93
9. Abkürzungsverzeichnis	95
10. Abbildungsverzeichnis	96
11. Literaturverzeichnis	97

1. Hintergrund und Motivation

Der folgende Abschnitt bezieht sich im Besonderen auf die Flickr-API {Flickr Services #192} und meine Erfahrungen mit weiteren Web2.0-APIs. Hintergrund für die Entwicklung von TwoSpot bilden die Anforderungen einer Web2.0-Anwendung, die unter dem Projektnamen Fojobo entwickelt wird. Ihre Basisfunktionalität ähnelt dem sehr bekannten Foto-Portal Flickr {Flickr #102}, hebt sich aber in einer Reihe von Punkten ab. Einer davon betrifft die Integration von 3Party- bzw. Benutzeranwendungen. Ein typischer Ansatz ist die Bereitstellung einer Programmschnittstelle {API Directory #105}, wobei häufig ein REST-basierter Ansatz gewählt wird. Es existieren aber auch Schnittstellen auf Basis von SOAP oder XML-RPC. Die genannten Technologien ermöglichen eine programmiersprachenunabhängige Nutzung der API. Aus Sicherheitsgründen wird die Schnittstelle durch einen API-Schlüssel geschützt. Mit diesem authentifiziert sich die API und damit die Benutzeranwendung an der Kernanwendung {Flickr Hilfe #115}. Um einen API-Schlüssel zu erhalten, muss sich der Entwickler registrieren, womit der Schlüssel an ihn gebunden ist. Der API-Schlüssel allein berechtigt aber noch nicht zum Zugriff auf Nutzerdaten. Dazu muss der Eigentümer der Anwendung explizit die Zugriffsrechte erteilen. Dies erfolgt über eine einfache Bestätigungsseite (vgl. 1. Anhang).

Der beschriebene Ansatz weist einige Nachteile auf, weshalb eine Cloud-Plattform als Basis von Fojobo eingesetzt werden soll. Die Anbindung von Benutzeranwendungen erfolgt dann nicht mehr über eine Web-Schnittstelle. Stattdessen werden sie auf derselben Cloud-Plattform wie Fojobo ausgeführt und können somit direkt über die Plattform-API kommunizieren. Mit diesem Ansatz ergeben sich für Fojobo und die Benutzeranwendungen eine Reihe von Vorteilen:

- 1) Da sich die Benutzeranwendungen nicht direkt in die Oberfläche der Kernanwendung integrieren, wirken sie oftmals aufgesetzt. Es existieren allerdings einige Positivbeispiele (vgl. {Willkommen bei Facebook #106}). Die leistungsfähigere API einer Cloud-Plattform ermöglicht hingegen eine einfachere und flexiblere Integration.
- 2) Die Anbindung über eine REST-Schnittstelle weist Limitierungen bei der Verarbeitung großer Datenmengen auf. Die Fojobo-Anwendung verwaltet beispielsweise Bilddateien mit einem Datenvolumen zwischen 5 MB und 50 MB. Die Übertragung einiger Hundert Bilddateien mit einem durchschnittlichen Volumen von ca. 5 MB ist selbst über die schnelle Internetanbindung von Rechenzentren nicht realisierbar oder führt zu hohen Übertragungskosten. In einer Cloud-Plattform lassen sich die Daten hingegen direkt über die interne und kosten-günstige Vernetzung übertragen.
- 3) Zur Verarbeitung von Daten müssen diese zwangsläufig an die Benutzeranwendungen übertragen werden. In vielen Fällen ist dies aber nicht im Interesse des

Nutzers. Da die Datenübertragung zudem eine explizite Genehmigung erfordert, sinkt möglicherweise die Nutzungsakzeptanz. Cloud-Anwendungen müssen die Daten hingegen nicht an externe Server übermitteln. Außerdem lassen sie sich wesentlich einfacher zertifizieren, womit sich wiederum das Nutzervertrauen stärken lässt.

- 4) Die Infrastruktur einer Benutzeranwendung muss der Anwendungsentwickler bzw. Betreiber bereitstellen. Dies gilt auch für unprofitable aber oftmals nützliche und daher sehr erfolgreiche Anwendungen. Beim Einsatz einer Cloud-Plattform muss hingegen keine eigene Infrastruktur bereitgestellt werden. Außerdem kann der Plattform-Betreiber auf die Skalierbarkeit und Verfügbarkeit der Benutzeranwendungen Einfluss nehmen. Da sie häufig einen großen Nutzen für die Kernanwendung selbst darstellen, lässt sich zudem eine teilweise kostenfreie Nutzung der Plattform realisieren.
- 5) Kostenpflichtige Benutzeranwendungen müssen auf ein eigenes Bezahlungssystem zurückgreifen. Der Nutzer wird somit u. U. mit einer Vielzahl verschiedener Bezahlungssysteme konfrontiert. Über eine Cloud-Plattform lässt sich hingegen ein zentrales Bezahlungssystem realisieren.

Das Ziel dieser Arbeit besteht allerdings nicht in der Entwicklung von Fojobo auf Basis einer existierenden Cloud-Plattform. Vielmehr soll aufbauend auf einer Analyse bestehender Cloud-Plattformen das Fundament einer neuen Plattform entworfen und realisiert werden. Der Einsatz existierender Cloud-Plattformen scheidet aufgrund ihrer Einschränkungen und den Erfahrungen aus einem Fojobo-Prototyp {fojobo #104} aus.

An die zu realisierende Cloud-Plattform werden vorab eine Reihe von Anforderungen gestellt:

- 1) Alle Komponenten müssen auf eine horizontale Skalierung ausgelegt sein.
- 2) Der Implementierungs- und Integrationsaufwand einer Cloud-Anwendung soll möglichst gering sein (10 Minuten bis zur „Hello World“-Anwendung).
- 3) Unterstützung verschiedener Programmiersprachen für die Cloud-Anwendungen.
- 4) Die Verwaltung und der Betrieb einer Cloud-Anwendung müssen ressourcensparend und kostengünstig erfolgen. Falls eine Anwendung nicht aktiv genutzt wird, darf sie z. B. keine Ressourcen in Anspruch nehmen.
- 5) Die Plattform muss für den Betrieb unter verschiedenen Betriebssystemen ausgelegt sein.
- 6) Zur Unterstützung rapider Lastanstiege muss das Upscaling einer Anwendung möglichst schnell erfolgen. Um Betriebskosten einzusparen, ist ein effizientes Downscaling ebenfalls wünschenswert.

- 7) Hardwaredefekte und Ausfälle von Servern oder Softwarekomponenten treten häufig auf. Derartige Fehlerszenarien sollen sich nur geringfügig auf den Betrieb einer Anwendung auswirken.
- 8) Cloud-Plattformen unterliegen oftmals einem Vendor Lock-in. Die TwoSpot-Plattform soll auf offene Standards und Technologien setzen, um diesem entgegen zu wirken.

Aufgrund des zeitlichen Arbeitsumfangs wurde bereits vorab eine Realisierung der nachfolgend aufgeführten Funktionalität ausgeschlossen.

- 9) Kopplung mehrerer Rechenzentren bzw. Server-Cluster.
- 10) Monitoring- und Management-Funktionen zur Überwachung der Cloud-Plattform selbst und der Cloud-Anwendungen.
- 11) Nutzungsbasiertes Bezahlungsmodell auf Basis der Monitoring-Funktionen.

2. Cloud Computing

Vor einer Analyse bestehender Cloud-Computing-Angebote muss zunächst der Begriff Cloud Computing an sich definiert werden. Wie die folgenden Beispiele zeigen, existiert bislang noch keine allgemeingültige Definition:

„Today, people use the term cloud computing in many ways - some consider it to be a pool of virtualized computer resources, whereas others say it's the dynamic development, composition, and deployment of software fragments.“ {Christof Weinhardt 3/6/2009 #8}

„Cloud computing doesn't yet have a standard definition, but a good working description of it is to say that clouds, or clusters of distributed computers, provide on-demand resources and services over a network, usually the Internet, with the scale and reliability of a data center.“ {Robert L. Grossman 3/6/2009 #7}

„Fortunately most computer savvy folks actually have a pretty good idea of what the term 'cloud computing' means: outsourced, pay-as-you-go, on-demand, somewhere in the Internet, etc.“ {Eicken #13}

Auch wenn eine Definition von Cloud Computing schwerfällt, so hat sich der Begriff ähnlich wie Web2.0 zu einem der Schlagwörter im Internet entwickelt {Google Trends #116}. Da daher viele Anbieter auf den Trend aufspringen und bestehende Produkte mit Cloud Computing oder Cloud-Ready bewerben (vgl. {Maguire 2009 #17}), erschwert sich die Suche nach einer Definition zunehmend.

2.1. Klassifikationsschema

Diese entstandene Vielzahl von Cloud-Angeboten macht ein Klassifikationsschema notwendig. Allgemein lassen sich die Angebote in drei Kategorien einordnen: 1) Infrastructure-as-a-Service; 2) Platform-as-a-Service; 3) Software-as-a-Service. Diese Kategorien wurden auch von Alexander Lenk et al. {Lenk 2009 #89} aufgegriffen und weiter ausgeführt. Das entstandene Klassifikationsschema ordnet die Kategorien in Ebenen an und führt weitere Unterteilungen ein (vgl. 2. Anhang). Die einzelnen Ebenen werden nachfolgend beschrieben und später zur Klassifikation und Beschreibung verschiedener Cloud-Provider herangezogen.

2.1.1. Infrastructure-as-a-Service

Die IaaS¹-Ebene bildet die Grundlage aller Cloud-Provider und umfasst die Hardware an sich und weitere sog. Enabling Technologies {Weinhardt 2009 #8: 31}. Als Enabling Technology wird oftmals ein Virtual Resource Set (VRS) verwendet {Lenk 2009 #89: 24}. Darunter versteht man den Einsatz der Virtualisierungs-Technologie mit einem

¹ Infrastructure-as-a-Service

Hypervisor, um mehrere virtuelle Computer (Virtual Machine) auf einem physikalischen Server auszuführen. Aus Perspektive der Anwendungen ist eine VM² identisch zu einem gewöhnlichen Server. Zum Betrieb einer IaaS-Cloud mit einem VRS greifen die Betreiber meist auf bekannte Hypervisor wie XEN {Welcome to xen.org #180}, KVM {Main Page 1/19/2010 #181}, VmWare ESX {VMware #183} oder HyperV {Microsoft Hyper-V Server 3/4/2010 #184} zurück {Microsoft, Citrix and VMware #151}. Zudem enthält das VRS Management-Funktionen zur automatisierten Verwaltung und Konfiguration der VMs. Sie betreffen das Starten und Beenden, die Netzwerkkonfiguration, Kapazitätskonfiguration oder auch Failover-Funktionen. Die bereitgestellten VMs lassen sich entweder vollständig selbst vom Nutzer installieren (vgl. {Amazon 2010 #71}) oder basieren auf einer vorgegebenen Basiskonfiguration (vgl. {Rackspace HOSTING #26}).

Besonders in Rechenzentren weist die Virtualisierung einige Vorteile gegenüber klassischer Server auf:

- 1) Auf einem Server lassen sich mehrere VMs gleichzeitig betreiben. Normalerweise würde statt jeweils einer VM ein eigener Server benötigt. Diese werden nun auf einem Server konsolidiert. Durch die gemeinsame Nutzung der Hardware lässt sich diese wesentlich effizienter ausnutzen, was sich positiv auf den Energieverbrauch, die Kühlung und die Raumnutzung auswirkt. Darüber hinaus reduzieren sich die Wartungskosten aufgrund der geringeren Serveranzahl. {The Advantages of Using Virtualization #136}{Top 5 benefits of server #152}.
- 2) Da die VMs und deren Betriebssysteme nicht an die Hardware gebunden sind, lassen sie sich einfach von einem Server zu einem anderen verschieben. Dies ist z. B. sinnvoll, um Hotspots (sehr stark belastete Server) zu entlasten. Bei Hardwaredefekten lässt sich die VM einfach auf einem anderen, funktionsfähigen Server wiederherstellen. Die Kosten eines Hardwaredefekts reduzieren sich somit. {The Advantages of Using Virtualization #136}{Top 5 benefits of server #152}

Im Vergleich zum klassischen Server-Hosting ermöglichen VMs auch eine wesentlich kürzere Einrichtungszeit und damit Vertragslaufzeit {The Advantages of Using Virtualization #136}. Daher stellen IaaS-Provider üblicherweise eine Programmschnittstelle bereit, über die sich die Verwaltung des Resource Set automatisieren lässt {Lenk 2009 #89: 24}. In Kombination mit dem Pay-per-Use-Preismodell ist diese Funktionalität für skalierbare Server-Anwendungen von besonderem Interesse. In Abhängigkeit zur Belastung können sie neue VMs starten oder beenden. Die Betriebskosten sind folglich direkt von der tatsächlichen Belastung abhängig. Feste Kosten für Reserve-Server zur

² Virtual Machine, ein virtueller Computer

Bewältigung kurzer Lastspitzen oder zur Redundanz entfallen ebenfalls {Grossman 2009 #7: 24}.

Anstelle eines VRS kann aber auch ein klassisches Physical Resource Set (PRS) zum Einsatz kommen {Lenk 2009 #89: 24}. Darunter versteht man die Hardware selbst ohne einen Hypervisor. Die Unterscheidung zwischen VRS und PRS ist erforderlich, da nicht alle Cloud-Provider eine Virtualisierung einsetzen. Gründe dafür sind z. B. der Overhead eines Hypervisors, die Nutzung nicht virtualisierbarer Ressourcen oder der Einsatz spezieller Virtualisierungs-Technologien, die eine direkte Schnittstelle zur Hardware bieten. Beim PRS kommen, ähnlich zum VRS, Mechanismen zur Verwaltung der Ressourcen zum Einsatz. Sie umfassen z. B. Funktionen zur Konfiguration des Netzwerkes oder dem Starten und Beenden bestimmter Hardwareressourcen.

Darüber hinaus enthält die IaaS-Ebene einfache Dienste, deren Basis das VRS oder PRS bildet {Lenk 2009 #89: 26}. Diese werden als Basic Infrastructure Service (BIS) bezeichnet. Beispiele dafür sind Dateisysteme wie das Google File System {Ghemawat #110}, Hadoop File System {Apache Software Foundation #81}, MogileFS {Danga Interactive #77} oder Amazon S3 {Amazon Simple Storage Service Amazon 3/1/2010 #179}. In der IaaS-Ebene können sich aber auch höherwertigere Dienste befinden, die entsprechend als High Infrastructure Services (HIS) bezeichnet werden. Beispiele sind Speicherdienste wie Amazon Dynamo {DeCandia #185}, Cassandra {Apache Software Foundation #67}, Google Bigtable {Fay 2006 #82} oder HBase {Apache Software Foundation #66}. Die Unterscheidung zwischen BIS und HIS erfolgt entsprechend dem Funktionsumfang. Die BIS-Dienste stellen lediglich grundlegende Funktionen z. B. zur Arbeit mit Dateien zur Verfügung, wohingegen HIS-Diensten komplexere Funktionen wie eine Query-Sprache bereitstellen. Der Übergang zwischen BIS und HIS ist allerdings als fließend zu betrachten.

2.1.2. Platform-as-a-Service

Die PaaS³-Ebene ist im Vergleich zur IaaS-Ebene abstrakter. Sie umfasst ein Execution Environment und ein Programming Environment {Lenk 2009 #89: 24}. Das Execution Environment ist ähnlich einem Anwendungsserver für die Ausführung von Anwendung zuständig. Zusätzlich unterstützt es die Anwendung oftmals bei der Skalierung. Unter dem Programming Environment versteht man das Framework, auf dem die Anwendung betrieben wird. Beispiele sind z. B. das Django-Framework {Django | The Web framework #150} oder das Web-Framework der Google App Engine {Google Inc. #69}. Üblicherweise umfasst ein PaaS-Angebot sowohl ein Execution als auch ein Programming Environment. Einige Provider ermöglichen auch den Austausch des Programming Environments.

³ Platform-as-a-Service

Als Grundlage der PaaS-Ebene und des Execution Environments wird häufig eine IaaS-Ebene mit einem PRS eingesetzt. Ein VRS wird aufgrund des Overheads der Virtualisierung und der fehlenden Notwendigkeit nicht verwendet (vgl. S. 33, Execution Environment). Damit lassen sich die Ressourcenanforderungen pro Anwendung und somit die Betriebskosten senken. Im Vergleich zur IaaS-Ebene erhält der Benutzer keinen Zugriff auf eine VM, sondern lädt die Anwendung direkt in die PaaS-Plattform. Dort wird sie automatisch in einem Execution Environment ausgeführt.

Das Execution Environment ist oftmals eine proprietäre Entwicklung der PaaS-Provider. In extremen Fällen kommen dabei sogar proprietäre Programmiersprachen zum Einsatz (vgl. Salesforce.com Apex; {CRM Software & Online CRM #137}) {Grossman 2009 #7: 26}. Die Folge ist ein Vendor Lock-in {Lawton 2008 #73: 15}. Aufgrund der proprietären Umgebung und Schnittstellen lässt sich eine Software, die auf Basis eines PaaS-Providers entwickelt wurde, nicht oder nur mit erheblichem Aufwand an das Execution Environment eines anderen PaaS-Providers anpassen. Folglich sind die Anwendungen an den Provider gebunden {Weinhardt 2009 #8: 30}.

Das Execution Environment und der Vendor Lock-in weisen aber noch einen weiteren Nachteil auf. Die Anwendungen können lediglich auf der API des Execution Environments operieren. Oftmals sind aber Zugriffe auf Funktionen des Betriebssystems nicht möglich (vgl. S. 33, Execution Environment). Als Folge lassen sich viele bestehende Software-Bibliotheken nicht ohne Modifikation einsetzen (vgl. {Will it play in App #95}).

PaaS-Angebote umfassen meist auch ein Development Environment. Es ermöglicht eine Anwendungsentwicklung und Anwendungstests direkt im Browser und somit der PaaS-Plattform (vgl. {Platform-as-a-Service 9/15/2009 #138}). Als Development Environment werden aber auch allgemeine Tools und APIs für die Anwendungsentwicklung bezeichnet. Darunter zählen z. B. Tools zum Deployment oder ein Test-Server für die lokale Entwicklung {Lawton 2008 #73: 13}.

2.1.3. Software-as-a-Service

Die SaaS⁴-Ebene bildet die letzte Ebene im Cloud-Stack. In ihr befinden sich nutzbare Anwendungen, die entweder auf Basis der IaaS- oder der PaaS-Ebene betrieben werden {Lenk 2009 #89: 24}.

Die Unterteilung von SaaS-Anwendungen erfolgt in zwei Klassen {Lenk 2009 #89: 24}: Basic Application Services sind eigenständige Anwendungen, die von Endkunden genutzt werden (z. B. Office-Pakete oder Bildverarbeitungs-Anwendungen); Composite Application Services sind Anwendungskomponenten, die sich zur Erstellung von Basic Application Services zusammensetzen lassen. Ein typisches Beispiel sind Web2.0

⁴ Software-as-a-Service

Mashups. Der Begriff Mashup entstammt ursprünglich aus der Welt der Musik und bezeichnet Kompositionen, die zwei oder mehr musikalische Werke in Form eines neuen musikalischen Werkes zusammenführen {Mashup music 3/3/2010 #140}. Übertragen auf Web2.0-Anwendungen ist ein Mashup eine Web-Anwendung, die Daten aus zwei oder mehr externen Quellen zusammenführt und daraus eine neue Anwendung bildet {Mashup web application hybrid 3/3/2010 #139}. Ein Beispiel ist die Einblendung von Gesundheitsdaten in Google Maps {Google Maps #141} {HealthMap | Global disease alert #186}.

2.2. Preismodell

Das von Cloud-Providern bevorzugte Pay-per-Use-Preismodell {Weinhardt 2009 #8: 31} unterscheidet sich stark von dem klassischen Abonnement-Preismodell typischer Hosting-Anbieter. Dieses basiert auf vergleichsweise langfristigen Verträgen mit einer Laufzeit von Monaten oder sogar Jahren. Außerdem werden pauschale Gebühren eingesetzt, die unabhängig zu der tatsächlich genutzten Leistung sind {Weinhardt 2009 #8: 32}.

Das Pay-per-Use-Preismodell ist schon länger aus der Telekommunikationsbranche bekannt. Es basiert auf der Bereitstellung von Diensten, die sich nach Bedarf nutzen lassen. Kosten entstehen lediglich bei Nutzung und in Abhängigkeit zur Menge der Nutzung. Pay-per-Use hat für Cloud-Provider einige wesentliche Vorteile: 1) Es fallen keine Investitionskosten an, da nur die Nutzung bezahlt wird; 2) Die Dienste lassen sich erst dann nutzen und bezahlt, wenn sie tatsächlich benötigt werden {Grossman 2009 #7: 24}; 3) Das Preismodell ist sehr einfach, womit die entstehenden Kosten vorhersehbar sind; 4) Es muss nur die tatsächlich erbrachte Leistung bezahlt werden {Weinhardt 2009 #8: 32}. IaaS-Provider verwenden das Preismodell, um VMs als Dienst anzubieten. Diese lassen sich dabei in Zeiteinheiten nutzen, wobei meist eine Stunde als kleinste Zeiteinheit verwendet wird. Die Kosten richten sich nach der Leistungsfähigkeit der VM und eventuell dem genutzten Betriebssystem.

Im Gegensatz zu IaaS-Providern gestaltet sich die Anwendung von Pay-per-Use auf PaaS-Provider schwierig. Ein Grund dafür ist der häufige Einsatz eines PRS in der IaaS-Ebene. Da sich nicht einfach die Nutzungsdauer einer VM oder Anwendung berechnen lässt, ermitteln PaaS-Provider die genutzten Rechner-Ressourcen wie CPU-Zyklen, Internet-Bandbreite, Anzahl der Datenbanktransaktionen oder Menge der gespeicherten Daten. Die Kosten richten sich dann nach der Anzahl an genutzten Ressourcen-Einheiten. Die Kosten einer Ressourcen-Einheit sind abhängig vom Ressourcen-Typ. Zum Beispiel ist die Übertragung von 1 GB Daten teurer als eine Datenbanktransaktion.

2.3. Sicherheitsaspekte

Im Bereich des Cloud Computing ist das Thema Sicherheit und das Vertrauen in den Cloud-Provider von besonderer Bedeutung {The biggest cloud-computing issue #142}. Balachandra Reddy et al. führen eine Reihe von Sicherheitsbedenken bezüglich des Cloud Computing an {Kandukuri #143}:

Besondere Bedenken bestehen beim Speichern kritischer Unternehmensdaten innerhalb der Infrastruktur eines Cloud-Providers. Im Vergleich zu Inhouse-Anwendungen befinden sich die Daten nicht mehr unter der vollständigen Kontrolle des Unternehmens. Oftmals ist unklar, welche Mitarbeiter, z. B. im Rahmen von Wartungsarbeiten Zugriff auf die Daten erhalten. Zusätzlich besteht die Gefahr, dass sich Mitarbeiter des Cloud-Providers unbefugt Zugang zu den Daten verschaffen. {Kandukuri #143} {Grossman 2009 #7: 26} {Lawton 2008 #73: 15}

Weitere Probleme ergeben sich durch Datenschutzgesetze. Entsprechend der Richtlinie 95/46/EG zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, ist die Übermittlung von Daten an Drittländer der EU nur unter bestimmten Bedingungen möglich {Schutz von personenbezogenen Daten 2009 #145}. Für Unternehmen ist die Lokalität der gespeicherten Daten somit von besonderem Interesse. Oftmals ist der Speicherort aufgrund der Datenreplikation zwischen mehreren Rechenzentren aber unklar {Kandukuri #143}.

Viele Cloud-Provider speichern die Daten aller Kunden in einem gemeinsam genutzten Speichersystem. Dabei können z. B. durch Software- oder Sicherheitsfehler Datenlecks entstehen, die den Zugriff auf fremde Daten ermöglichen. Beim Einsatz von Verschlüsselungsmechanismen besteht darüber hinaus die Möglichkeit eines Datenverlustes durch eine fehlerhafte Implementierung der Verschlüsselungsverfahren. {Kandukuri #143} {Grossman 2009 #7: 26}

Sicherheitsbedenken ergeben sich auch bezüglich des Ausfalls einer größeren Menge an Hardware oder sogar gesamter Rechenzentren. Da oftmals proprietäre Speicherdienste zum Einsatz kommen, sind die Backup-Mechanismen von besonderem Interesse. {Kandukuri #143}

2.4. Vor- und Nachteile

Gegenüber klassischer Server kann der Einsatz einer Cloud-Computing-Lösung einige Vorteile, aber auch Nachteile mit sich bringen. Im Folgenden sind die häufig genannten Vorteile aufgeführt:

- 1) Die Vorteile großer Rechenzentren lassen sich nutzen {Grossman 2009 #7: 26}{Leavitt 2009 #14: 17}: Verfügbarkeit, Infrastruktur bezüglich Stromversorgung, Kühlung, Netzwerk- und Internetanbindung.

- 2) Es fallen keine Investitionskosten für Hardware oder Software an {Grossman 2009 #7: 24}{Leavitt 2009 #14: 17}.
- 3) Die Konfiguration und Wartung der Hardware entfällt vollständig {Leavitt 2009 #14: 17}. Bei PaaS-Providern gilt dies auch für das Betriebssystem. {Lawton 2008 #73: 14}
- 4) Das Execution Environment der PaaS-Provider vereinfacht den Zugriff und die Verwendung von Diensten wie der Datenbank, Cache-Mechanismen oder Suchfunktionen und vereinfacht somit die Entwicklung skalierbarer Web-Anwendungen {Lawton 2008 #73: 14}{Grossman 2009 #7: 27}.
- 5) Durch die bedarfsabhängige Nutzung von Ressourcen lassen sich die Kosten für Reserve-Systeme einsparen {Grossman 2009 #7: 26}.
- 6) Die Plattform bietet die Möglichkeit, kurzfristig hohe Ressourcen-Anforderungen zu erfüllen und die Anwendung nach Bedarf zu skalieren {Grossman 2009 #7: 26}.
- 7) Das Pay-per-Use-Preismodell ist einfach zu überblicken und die Kosten stehen in direkter Abhängigkeit zu der tatsächlich genutzten Leistung {Grossman 2009 #7: 24}.
- 8) Die Virtualisierungs-Technologie erlaubt eine sichere und effiziente Nutzung der Hardware, womit wiederum die Betriebskosten sinken {Weinhardt 2009 #8: 30}.

Zusätzlich zu den zuvor beschriebenen Sicherheitsproblemen können beim Einsatz eines Cloud-Providers aber noch zwei weitere Nachteile auftreten:

- 1) Der Zugriff auf Cloud-Anwendungen erfolgt grundsätzlich über das Internet. Folglich wird eine schnelle Internetanbindung vorausgesetzt. Trotzdem ergeben sich im Vergleich zum lokalen Netzwerk eine Reihe von Nachteilen: erhöhte Latenzzeit, erhöhte Störanfälligkeit, zusätzliche Kosten für die Datenübertragung {Leavitt 2009 #14: 18}.
- 2) Die Execution Environments der PaaS-Provider führen häufig zu Einschränkungen bei der Anwendungsentwicklung und einem Vendor Lock-in {Leavitt 2009 #14: 18}{Lawton 2008 #73: 15}.

Aufgrund der aufgeführten Vor- und Nachteile wird deutlich, dass besonders Web-Anwendungen wie Fojobo vom Einsatz eines Cloud-Providers profitieren. Durch eine effiziente Nutzung der Ressourcen lassen sich Betriebs- und Investitionskosten senken. Besonders Web-Anwendungen unterliegen aus verschiedenen Gründen, wie dem Slashdot Effect {Geeknet, Inc #41} oder gewöhnlichen Lastspitzen zur Mittagszeit {OmniTI #118} einer stark schwankenden Belastung. In diesem Zusammenhang lassen sich durch die On-Demand-Nutzung von Ressourcen in Kombination mit dem Pay-per-Use-Preismodell Betriebskosten einsparen. Gleichzeitig verbessert sich die Qualität, da in jedem Fall genug Ressourcen zur Bewältigung von Lastspitzen zur Verfügung

stehen. Darüber hinaus wird die Entwicklung hochskalierbarer Anwendungen durch die bereitgestellten Basistechnologien und Dienste oftmals erleichtert.

Die aufgeführten Nachteile fallen für Web-Anwendungen hingegen kaum ins Gewicht. Besonders da viele Nachteile auch für gewöhnliche Hosting-Lösungen gelten. Zwei spezielle Nachteile der PaaS-Angebote sind der Vendor Lock-in mit den Einschränkungen des Execution Environments und die fehlende Transparenz bezüglich der Datenlokalität. Besonders der Vendor Lock-in kann sich negativ auf die Entwicklung und den Betrieb von Anwendungen auswirken. Daher soll die TwoSpot-Plattform auf offenen Standards und Technologien basieren (vgl. S. 1, Hintergrund und Motivation), was dem Vendor Lock-in entgegenwirkt.

3. Evaluation

Im Rahmen der Arbeit wurde eine Reihe bekannter PaaS-Provider analysiert. Aufgrund der großen Menge an verschiedenen Anbietern war eine Betrachtung aller Anbieter allerdings nicht möglich. Mithilfe einer Internetrecherche und den beiden Quellen: {Maguire 2009 #17}, {Laird 2008 #59} wurden daher die vielversprechendsten PaaS-Provider für eine genauere Analyse ausgewählt:

- 1) Google App Engine
- 2) Joyent Smart
- 3) Heroku
- 4) Microsoft Windows Azure
- 5) AppScale

3.1. Google App Engine

Die App Engine {Google Inc. #69} basiert auf der Infrastruktur von Google und ist auf den Betrieb hochskalierbarer Web-Anwendungen ausgelegt. Anwendungen lassen sich dabei entweder in Java oder Python programmieren {Google 9/10/2009 #37}. Die Quellcodes der App Engine-Plattform selbst sind nicht verfügbar. Da sie aber stark auf die interne Google-Infrastruktur zurückgreifen, wäre eine externe Installation ohnehin nicht möglich. Die nachfolgende Darstellung basiert daher auf dem Vortrag „From Spark Plug to Drive Train: Life of an App Engine Request“ von der Google IO 2009 Developer Conference {Levi 28.05.2009 #12}.

3.1.1. Architektur

Die App Engine orientiert sich an der mehrschichtigen Architektur skalierbarer Web-Anwendungen. Diese besteht aus einer Schicht zum Load-Balancing, einer Schicht aus Anwendungsservern und abschließend einer Datenbank-Schicht {Levi 28.05.2009 #12: 19}. Die App Engine-Architektur stellt eine Weiterentwicklung dieses Ansatzes dar {Levi 28.05.2009 #12: 20} (vgl. Abb. 1).

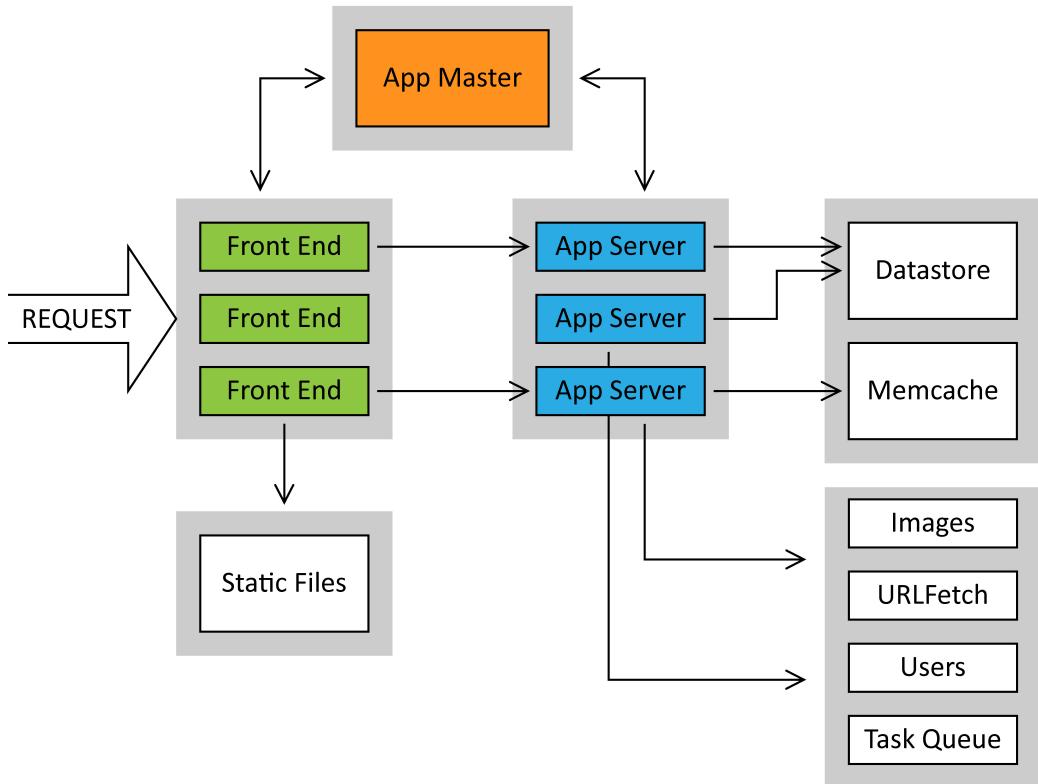


Abb. 1 Architektur der Google App Engine {Levi 28.05.2009 #12: 20}

Jede HTTP-Request wird vom nächstgelegenen Google-Rechenzentrum entgegengenommen und über das interne Google-Netzwerk an einen App Engine-Cluster weitergeleitet. Dieses Vorgehen beschleunigt laut Google {Levi 28.05.2009 #12: 24} die Datenübertragung und reduziert die Latenzzeiten, da das Google-Netzwerk im Vergleich zum öffentlichen Internet eine bessere Übertragungsqualität bietet. Im App Engine-Cluster leiten Load-Balancer die Requests an die Front End-Server weiter. Sie analysieren die Requests und ermitteln zunächst den Request-Typ. Static Content-Requests leiten sie direkt an den Google Static Content Serving-Dienst weiter. Dieser ist auf die Bereitstellung unveränderlicher bzw. statischer Dateien wie z. B. Bild-, CSS- oder JavaScript-Dateien optimiert. Allerdings ist der Dienst kein zentraler Bestandteil der App Engine und wird auch von verschiedenen anderen Google-Produkten wie der Suche genutzt {Levi 28.05.2009 #12: 26}.

Bei einer Dynamic Content-Request startet das Frontend zunächst eine Anfrage beim App Master. Anhand der Monitoring-Daten der App Server und den Request-Informationen wählt der Master einen oder mehrere App Server zur Verarbeitung der Request aus. Bevorzugt werden dabei App Server, unter denen die entsprechende Anwendung bereits ausgeführt wird. Falls solche App Server nicht existieren, wählt der Master einen möglichst gering belasteten App Server aus. Das Ergebnis gibt er an das Frontend zurück, das die Request zum gewählten App Server weiterleitet. Dieser prüft zunächst, ob die entsprechende Anwendung zur Verarbeitung der Request bereits ausgeführt wird. Falls nicht, startet er ein neues Execution Environment zusammen mit

der Anwendung. Nun leitet er die Request an die Anwendung weiter. Im Anschluss an die Request-Verarbeitung wird das Execution Environment und die Anwendung für eine bestimmte Zeit weiter betrieben. Dadurch können nachfolgende Requests direkt auf die bereits laufende Anwendung zurückgreifen und müssen nicht auf den Start eines Execution Environments warten. Die Verarbeitung nachfolgender Requests ist damit wesentlich effizienter.

Ein Nachteil dieser Anordnung aus Frontend und App Server ist das Session-Handling. Das Frontend unterstützt keine Sticky-Sessions. Folglich sind die Benutzersitzungen nicht an einen App Server und eine Anwendungsinstanz gebunden. Stattdessen wird jede Request unabhängig zur Benutzersitzung an einen App Server weitergeleitet. Dieser Ansatz vereinfacht maßgeblich das Load-Balancing {Levi 28.05.2009 #12: 47}, erschwert aber die Verwaltung der Sitzungsdaten. Diese müssen direkt in einem zentralen Speicherdienst oder einem Cookie abgelegt werden. Java-Anwendungen können zudem das Servlet Session-Interface nutzen, das ebenfalls auf den zentralen Speicherdiensten agiert {Google #204}.

Normalerweise führt ein App Server mehrere Execution Environments und Anwendungen gleichzeitig aus. Da die App Engine kein VRS einsetzt, ist das Execution Environment für die Anwendungsisolation verantwortlich. Isolation bedeutet dabei, dass sich die Anwendungen auf einem App Server unter keinen Umständen gegenseitig beeinträchtigen können und die verfügbaren Ressourcen gerecht auf alle Anwendungen verteilt werden. Um diese Anforderung zu gewährleisten, schränkt das Execution Environment die Anwendungen in mehreren Punkten ein {Levi 28.05.2009 #12: 46}. Erstens dürfen sie auf keine Funktionen des Betriebssystems zurückgreifen. Zweitens dürfen große, ressourcenhunggrige Anwendungen keine kleineren Anwendungen ausbremsen. Daher ist die maximale Verarbeitungsdauer einer Request auf 30 Sek. beschränkt {Google App Engine Blog 3/5/2010 #153}. Benötigt sie länger, wird ihre Bearbeitung abgebrochen.

Um die Entwicklung skalierbarer Web-Anwendungen zu ermöglichen und zu erleichtern, stellt die App Engine verschiedene Plattform-Dienste zur Verfügung. Im Folgenden werden lediglich die zentralen Dienste beschrieben.

3.1.2. Datastore

Der Datastore-Dienst basiert auf der, von Google entwickelten Bigtable-Datenbank. Bigtable ist ein „Distributed Storage System for Structured Data“ {Fay 08.2006 #82: 1} und wird bereits von verschiedenen Google-Diensten wie Google Analytics, Google Finance oder auch Google Earth eingesetzt {Fay 2006 #82}. Diese Beispiele verdeutlichen die Stabilität, Flexibilität und die Skalierbarkeit der Datenbank.

Bigtable unterscheidet sich signifikant von typischen RDBs⁵ und bietet z. B. keine SQL-Schnittstelle an, was ihren Einsatz im Vergleich zu RDBs erschwert. Der Datastore-Dienst stellt eine Art Middleware dar, mit der sich Bigtable ähnlich einer dokumentenorientierten Datenbank nutzen lässt. Die Speicherung von Daten erfordert eine Schema-Definition. In Python wird diese über die Deklaration einer Klasse mit bestimmten Attribut-Typen erstellt. In Java kommen die JDO⁶-Annotationen zum Einsatz. Anschließend lassen sich Objekte dieser Klasse über einfache Get-, Set- und Delete-Funktionen im Datastore speichern oder löschen. Genau wie bei einer dokumentenorientierten Datenbank, lässt sich das Schema jederzeit ohne eine Migration des bestehenden Datenbestandes verändern.

Zum Auslesen gespeicherter Daten wird eine SQL-ähnliche Sprache mit dem Namen GQL⁷ verwendet. Da der Datastore und die Bigtable-Datenbank dem Funktionsumfang einer RDB weit nachstehen, ist der Funktionsumfang von GQL ebenfalls sehr eingeschränkt. GQL-Queries lassen sich entweder direkt als Zeichenkette oder aber über den Aufbau eines Objektbaums erstellen. Google empfiehlt die zweite Variante, da sie weniger Angriffspunkte gegenüber GQL-Injection-Angriffen aufweist {Google 9/5/2009 #38}. Für Java-Anwendungen besteht darüber hinaus die Möglichkeit, über JDO oder JPA⁸ auf die Daten zuzugreifen.

3.1.3. Memcache

App Engine-Anwendungen erhalten Zugriff auf einen Memcache-Cluster. Memcache wurde zur Beschleunigung von Web-Anwendungen entwickelt {Memcached #39}. Sein Einsatz wird dabei besonders zur Zwischenspeicherung langsamer Datenbank-Abfragen empfohlen. Damit wird die Reaktionszeit der Anwendung verbessert und die Belastung auf den Datastore reduziert {Effective memcache #94}. Allerdings ist der Memcache als unsicherer Speicher zu betrachten. Die Daten können jederzeit z. B. durch Hardwaredefekte oder Speicherengpässe verloren gehen {Effective memcache #94}. Außerdem muss zu jedem gespeicherten Wert ein Haltbarkeitszeitraum angegeben werden. Nach dessen Ablauf wird der Wert mit einer hohen Wahrscheinlichkeit gelöscht.

3.1.4. Weitere Dienste

Ergänzend zum Datastore- und Memcache-Dienst stellt die App Engine weitere Dienste bereit. Da diese aber keinen Kernbestandteil der App Engine-Architektur bilden, werden sie an dieser Stelle lediglich aufgelistet.

⁵ Relationale Datenbank (Relational Database)

⁶ Java Data Objects

⁷ Google Query Language

⁸ Java Persistence API

- 1) Der URLFetch-Dienst ermöglicht den Zugriff auf Internetressourcen über das HTTP-Protokoll. Notwendig ist dies, da die Anwendungen aufgrund der Isolation des Execution Environments keinerlei Netzwerkverbindungen aufbauen können.
- 2) Integration des Google Mail-Dienstes {Google Mail #130} zum Versand von E-Mails.
- 3) Anbindung an den Google Jabber Server, womit die Anwendungen über Instant Messaging-Nachrichten kommunizieren können.
- 4) Der Task Queue-Dienst bildet einen Linux Cron-Dienst nach. Anhand eines Zeitplans lassen sich damit Anwendungsressourcen zu bestimmten Zeiten oder in regelmäßigen Abständen ausführen.
- 5) Der Image-Dienst ermöglicht die eingeschränkte Verarbeitung von Bilddaten (beschränkt auf max. 1 MB pro Bild).
- 6) Die Google Benutzerkonten-Verwaltung lässt sich ebenfalls nutzen. Die Einbindung erfolgt ähnlich zu dem Ansatz des OpenID-Dienstes {OpenID Foundation website 3/2/2010 #131}.

3.1.5. Development Environment

Um eine komfortable und effiziente Anwendungsentwicklung zu ermöglichen, stellt die App Engine einen eigenen Entwicklungsserver zur Verfügung {Downloads 3/6/2010 #155}. Er erlaubt eine lokale Entwicklung und lokale Tests von Anwendungen. Da in der Testumgebung die Plattform-Dienste nicht zur Verfügung stehen, enthält der Entwicklungsserver entsprechende Mock-Implementierungen. Somit lassen sich App Engine-Anwendungen vollständig auf Basis des Entwicklungsservers entwickeln und testen. Anschließend können sie ohne Migrationsaufwand in die Produktivumgebung deployt werden. Zur Entwicklung von Java-Anwendungen existiert außerdem ein Eclipse-Plugin {Google Plugin for Eclipse 3/6/2010 #154}. Es enthält verschiedene Assistenten zum Erstellen und Deployen von Anwendungen. Über das Plugin kann auch der Entwicklungsserver komfortabel verwaltet, gestartet und beendet werden.

3.1.6. Horizontale Skalierbarkeit

Die App Engine setzt das Konzept der horizontalen Skalierung vollständig um. Im Unterschied zu den anderen Plattformen muss die Skalierung nicht vom Benutzer oder der Anwendung selbst gesteuert werden, sondern ist durch die App Engine automatisiert. Für jede eingehende Request entscheidet das Frontend zusammen mit dem App Master, ob eine weitere Anwendungsinstanz benötigt wird. Die Anwendungen selbst enthalten damit keine Mechanismen bezüglich der Skalierung.

Da die App Engine auf einem PRS basiert, laufen mehrere Anwendungen gleichzeitig auf einem Server und unter demselben Betriebssystem. Sie teilen sich somit die verfügbaren Ressourcen. Daher kann die App Engine einer Anwendungsinstanz keine Mindestleistung zusichern. Falls ein Leistungsengpass eintritt, startet die Plattform au-

tomatisch weitere Anwendungsinstanzen auf anderen Servern. Die Skalierbarkeit der App Engine wurde bereits durch verschiedene Anwendungen demonstriert. Ein Beispiel dafür ist die „Open for Questions“ Crowdsourcing-Anwendung vom Weißen Haus. Sie hatte eine Laufzeit von 48 Stunden in denen 100.000 Fragen und 2,6 Millionen Stimmabgaben erfasst wurden {Levi 28.05.2009 #12: 51}. Die Lastkurve dieser Anwendung (vgl. Abb. 2) verdeutlicht, dass die App Engine auch eine stark und sehr schnell schwankende Auslastung verarbeiten kann.

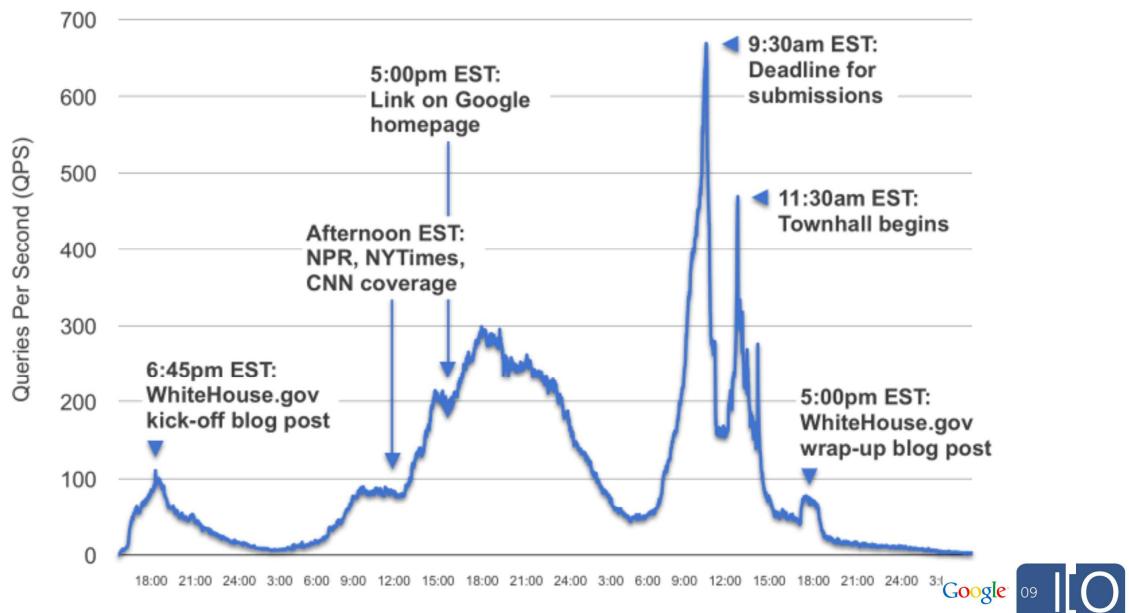


Abb. 2 Auslastung der App Engine-Anwendung „Open for Questions“ Quelle: {Levi 28.05.2009 #12: 51}

3.1.7. Zusammenfassung

Die App Engine vereinfacht die Entwicklung und den Betrieb horizontal skalierbarer Web-Anwendungen. Aufgrund verschiedener Sicherheitsanforderungen müssen die Anwendungen allerdings Einschränkungen in Kauf nehmen. Besonders Java-Anwendungen leiden unter diesen Einschränkungen, da sie den Einsatz verschiedener Java-Frameworks erschweren {Will it play in App #95}. Die App Engine enthält eine Reihe von Diensten, z. B. zur Datenspeicherung, Bildverarbeitung oder zum E-Mail-Versand. Die API dieser Dienste ist allerdings proprietär, was zu einem Vendor Lock-in führen kann. Ein Vorteil dieser proprietären APIs ist allerdings die künstliche Beschränkung des Funktionsumfangs auf skalierbare Funktionen. GQL unterstützt z. B. nur Funktionen, die sich effizient auf dem Datastore ausführen lassen {Queries and Indexes 3/6/2010 #156}. Damit wird die Entwicklung skalierbarer Anwendungen zusätzlich vereinfacht.

Darüber hinaus führt die App Engine ein Monitoring aller Anwendungen durch. Die erfassten Daten lassen sich zu einer umfangreichen Performance- und Fehleranalyse und zu Abrechnungszwecken einsetzen {The Administration Console 3/6/2010 #157}.

Die App Engine überwacht aber nicht nur die Anwendungen, sondern auch sich selbst. Diese Daten werden ebenfalls veröffentlicht und ermöglichen die Abgrenzung von Plattform- und Anwendungsfehlern {Google App Engine System Status #96}.

3.2. Joyent Smart

Die Smart-Plattform wird von Joyent als Open Source-Projekt entwickelt {Joyent Smart #76}. Joyent betreibt die Plattform auf der eigenen Infrastruktur, sie lässt sich aber auch auf jedem beliebigen Server installieren. Smart hebt sich von vielen anderen Plattformen durch den Einsatz von serverseitigem JavaScript ab. Die Plattform selbst ist allerdings in Perl implementiert. Die folgenden Ergebnisse entstammen einer Analyse der frei verfügbaren Quellcodes {joyent's smart-platform at master #158}.

3.2.1. Architektur

Die Plattform besteht aus zwei Anwendungen: Server und Slave. Auf jedem physikalischen Server werden genau ein Server-Prozess und beliebig viele Slave-Prozesse ausgeführt (vgl. Abb. 3). Die Anzahl der Slaves ist fest über eine Konfigurationsdatei vorgegeben.

Der Server nimmt alle HTTP-Requests entgegen, pausiert sie und reiht sie in eine Warteschlange ein. Jeder Slave führt eine Endlosschleife aus, die im ersten Schritt eine Request aus der Warteschlange des Servers entnimmt. Anschließend analysiert er die Request und ermittelt die Anwendung anhand des Hostnamens. Das Anwendungsarchiv wird dann über das verteilte Dateisystem geladen und extrahiert. Nun erstellt der Slave eine neue Instanz des JavaScript-Interpreters und lädt die bootstrap.js-Datei. Diese muss sich in jeder Anwendung befinden und markiert deren Einstiegspunkt. Abschließend übergibt er die Request an die Anwendung, wo sie verarbeitet wird. Nach dem Senden der Response löscht der Slave die Instanz des JavaScript-Interpreters wieder.

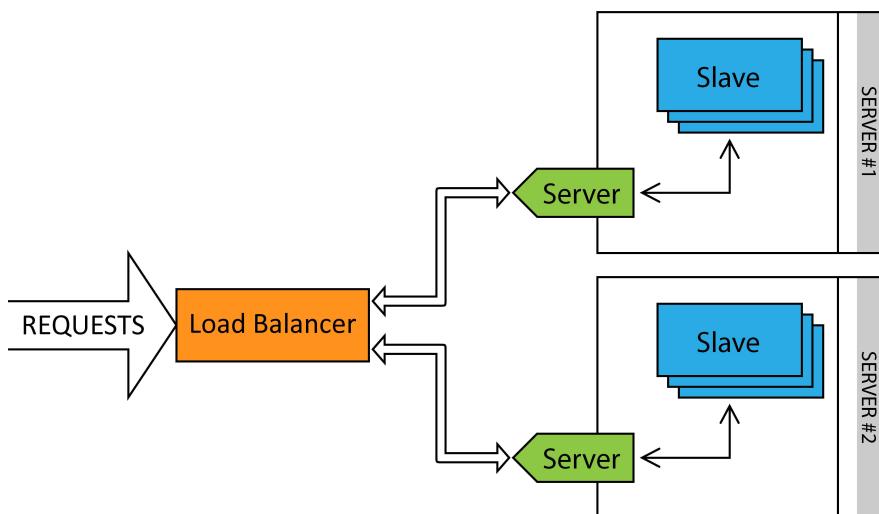


Abb. 3 Architektur der Joyent Smart-Plattform

Ein Slave ist somit zustandslos und nicht an eine einzelne Anwendung gekoppelt. Aus diesem Grund lassen sich die Slaves nach einem Anwendungsfehler oder Absturz problemlos neu starten. Diese Aufgabe übernimmt ein Watchdog im Server.

Ein wesentlicher Nachteil der zustandslosen Slaves ergibt sich bei der Verarbeitung von Initialisierungscode. Dieser wird mit jedem Anwendungsstart und damit jeder Request ausgeführt. Falls dieser Code zeitintensiv ist, reduziert er die Verarbeitungsgeschwindigkeit aller Requests. Ebenso können keine Verarbeitungsdaten im Arbeitsspeicher zwischengespeichert werden, da sie im Anschluss an die Request-Verarbeitung wieder gelöscht werden.

Aufgrund der Zustandslosigkeit werden auch keine speziellen Mechanismen zur Last-Verteilung benötigt. Im Unterschied zur App Engine werden die Requests nicht über ein Push-Verfahren an die Slaves und Anwendungen weitergeleitet. Stattdessen verwenden die Slaves ein Poll-Verfahren, um die Requests vom Server abzurufen. Dies garantiert eine optimale Auslastung eines Slaves. Um die Belastung über mehrere physikalische Server zu verteilen, kommt ein gewöhnlicher Load-Balancer zum Einsatz.

Da die Smart-Plattform wie die App Engine ein PRS einsetzt, ist das Execution Environment für die Anwendungsisolation verantwortlich. Aus diesem Grund stellt die JavaScript-Umgebung keine Funktionalität zum Zugriff auf das Betriebssystem zur Verfügung. Damit wird eine Beeinträchtigung der Slave-Prozesse durch die Anwendungen ausgeschlossen. Da die Anzahl der serverseitigen JavaScript-Bibliotheken im Vergleich zu Java oder Python wesentlich geringer ist, sind diese Einschränkungen unter dem Aspekt der Kompatibilität aber vernachlässigbar.

Wie die App Engine auch, stellt die Smart-Plattform eine Reihe von Plattform-Diensten zur Verfügung. Diese kompensieren die Einschränkungen des Execution Environments und erleichtern die Anwendungsentwicklung.

3.2.2. Datenbank

Zur Speicherung strukturierter Daten können die Anwendungen auf eine relationale Datenbank zugreifen. Joyent setzt z. B. die MySQL-Datenbank ein. Der Zugriff erfolgt aber nicht über eine SQL-Schnittstelle, sondern über eine proprietäre Speicher-API. Diese orientiert sich an der API dokumentenorientierter Datenbanken. Zum Speichern von Daten muss zunächst ein Schema in Form einer JavaScript-Klasse definiert werden. Objekte dieser Klasse lassen sich dann speichern oder über eine einfache Query-Sprache abfragen. Die relationale Datenbank eignet sich aber nicht für ein horizontal skalierbares System (vgl. S. 39, Datenbank). Da sich die API allerdings an dokumentenorientierten Datenbanken orientiert und generisch implementiert ist, besteht die Möglichkeit, die relationale Datenbank durch eine horizontal skalierbare Datenbank auszutauschen.

3.2.3. Blob-Speicher

Zur Ablage großer Datenmengen (Dateien größer als ein MB) stellt die Smart-Plattform den Blob-Speicherdiensst zur Verfügung. Der Zugriff erfolgt ebenfalls über eine proprietäre API. Als Speicher-Backend kommt das MogileFS-Dateisystem {Danga Interactive #77} zum Einsatz (vgl. S. 55, MogileFS). Eine sinnvolle Verwendung des Dienstes ist allerdings nur in Kombination mit der Datenbank möglich. Jede gespeicherte Datei erhält einen, vom System generierten eindeutigen Bezeichner. Dieser Bezeichner wird später zum Laden der Datei benötigt. Da keinerlei Mechanismen zur Dateiverwaltung verfügbar sind, müssen die Datei-Bezeichner in der Datenbank verwaltet werden.

3.2.4. Queue Worker

Über den Queue Worker-Dienst lassen sich JavaScript-Funktionen asynchron ausführen. Dies ist z. B. für die Verarbeitung größerer Datenmengen nützlich: Ein Request-Handler speichert eine größere Datenmenge im Blob-Speicher. Bei einer direkten Verarbeitung der Daten (z. B. bei einer ZIP-Kompression) im Request-Handler würde die Request blockiert und mit hoher Wahrscheinlichkeit wäre ein Browser-Timeout die Folge. Stattdessen lassen sich die Daten mithilfe des Queue Workers asynchron verarbeiten. Dazu erstellt der Request-Handler einen oder mehrere Jobs und registriert sie in einer Job-Queue. Ein Job besteht aus einer JavaScript-Datei, einem Funktionsnamen innerhalb der Datei und den Funktionsparametern. Die Jobs werden ähnlich zu den Requests in einem eigenen Slave bearbeitet. Die Ausführung erfolgt anhand der Job-Queue, sobald die Plattform über genug freie Ressourcen verfügt. Durch diesen Ansatz blockiert die ursprüngliche Request nicht, da die Datenverarbeitung unabhängig und zu einem späteren Zeitpunkt stattfindet.

3.2.5. Weitere Dienste

Zusätzlich zu den beschriebenen Diensten stellt die Smart-Plattform noch einige weitere Bibliotheken für die Verarbeitung von Bilddaten und Templating zur Verfügung. Allerdings entfällt an dieser Stelle eine Beschreibung, da sie keinen Kernbestandteil der Plattform-Architektur bilden.

3.2.6. Development Environment

Da die Smart-Plattform als Open Source-Projekt entwickelt wird, lässt sie sich auch auf jedem beliebigen Entwicklungsrechner installieren. Aufgrund verschiedener Einschränkungen lassen sich dabei aber nicht alle Plattform-Dienste nutzen. Darüber hinaus ist die Installation bislang nur unter Linux problemlos möglich. Für eine Windows-Umgebung existiert noch kein Installationspaket.

Das Deployment von Anwendungen in die von Joyent betriebene Smart-Plattform erfolgt über eine GIT-Versionsverwaltung. Dazu erhält jede Anwendung ein eigenes GIT-Repository. Für das Deployment muss die Anwendung einfach in das Repository gel-

den werden. Die Slave-Prozesse greifen immer auf die Head-Revision vom Master-Branch zurück, womit sie stets die aktuellste Anwendungsversion ausführen.

3.2.7. Horizontale Skalierbarkeit

Die Smart-Plattform wurde ähnlich der App Engine im Hinblick auf eine horizontale Skalierung entworfen. Im Unterschied zur App Engine wird aber zu jeder Request ein neues Execution Environment gestartet. Die Requests lassen sich damit gleichmäßig auf alle Server verteilen, womit eine horizontale Skalierbarkeit erreicht wird. Smart setzt ein PRS ein und führt damit mehrere Anwendungen gleichzeitig unter einem Betriebssystem aus. Folglich ist keine klare Aussage bezüglich der tatsächlichen Leistungsfähigkeit einer Anwendungsinstantz möglich. Allerdings kann jedem Slave-Prozess und damit jeder Anwendung bzw. Request ein CPU-Kern exklusiv zur Verfügung gestellt werden. Die Smart-Plattform greift zur Datenspeicherung auf eine relationale Datenbank zurück. Diese eignet sich allerdings nicht für eine horizontale Skalierung (vgl. S. 39, Datenbank) und stellt damit einen Schwachpunkt bezüglich der horizontalen Skalierbarkeit dar.

3.2.8. Zusammenfassung

Die Smart-Plattform erleichtert die Entwicklung skalierbarer Web-Anwendungen maßgeblich. Sie ermöglicht eine horizontale Skalierung der Anwendungen selbst, schränkt diese aber durch den Einsatz einer relationalen Datenbank ein. Als Programmiersprache kommt serverseitiges JavaScript zum Einsatz, denkbar ist aber auch der Einsatz ähnlicher Sprachen wie Python oder Ruby. Vorausgesetzt wird allerdings eine hohe Startgeschwindigkeit des Interpreters. Aufgrund des proprietären Execution Environments und der proprietären Schnittstellen unterliegen Smart-Anwendungen einem starken Vendor Lock-in.

Das Development Environment erscheint zum aktuellen Zeitpunkt noch nicht ausgereift, besonders aufgrund der schwerfälligen Installation. Der Einsatz eines GIT-Repositories zum Deployment weist gegenüber einem einfachen Upload-Mechanismus keine Vorteile auf. Zur Verwaltung der Projekt-Quellcodes existieren wesentlich leistungsfähigere Angebote wie z. B. GitHUB {Secure source code hosting #132}. Darüber hinaus ist die Einrichtung eines GIT-Clients unter einer Windows-Umgebung vergleichsweise schwerfällig. Damit erhöht sich die Hürde zur Entwicklung der ersten „Hello World“-Anwendung maßgeblich.

3.3. Heroku

Die Heroku-Plattform ist für den Betrieb von Ruby Web-Anwendungen konzipiert. Die Anwendungen werden mithilfe der Rack-Bibliothek {Neukirchen 2008 #133} an die

Plattform angebunden. Sie bildet ähnlich zur WSGI-Schnittstelle⁹ in Python eine universelle Schnittstelle zwischen Web-Server und Web-Framework. Heroku basiert auf dem Thin Ruby Web-Server {Thin 9/6/2009 #100}, der eine Weiterentwicklung des bekannten Mongrel-Servers {Mongrel #99} ist. Er ist die Grundlage vieler bekannter Ruby Web-Frameworks wie Sinatra {Sinatra 2/1/2010 #160}, Merb {Merb | Looking 12/23/2008 #161}, Camping {Camping, the Documentation File 10/4/2006 #162} oder das sehr bekannte Ruby On Rails {Ruby on Rails #163}. Die von Heroku eingesetzte Server-Software ist somit vollständig frei verfügbar und lässt sich auch auf privaten Servern betreiben. Ein Vendor Lock-in existiert daher nicht. Die Quellcodes der Plattform selbst sind allerdings nicht öffentlich zugänglich. Daher bezieht sich der folgende Abschnitt auf die Dokumentation der Plattform {Heroku | Docs #159}.

Die Heroku-Plattform ist vollständig in Ruby entwickelt und ermöglicht auch nur den Betrieb von Ruby-Anwendungen. Jede Anwendung wird in einem oder mehreren Dyno-Prozessen gleichzeitig ausgeführt, deren Basis eine Ruby VM bildet. In dieser wird der Thin-Server mit der Rack-Schnittstelle ausgeführt. Darauf aufbauend befindet sich das Rack-kompatible Web-Framework und abschließend die Anwendung selbst (vgl. Abb. 4).

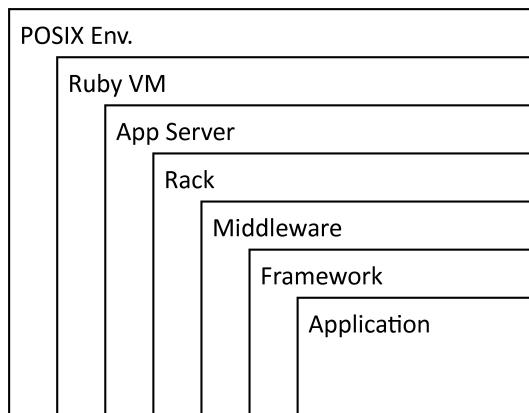


Abb. 4 Software-Stack eines Heroku Dyno-Prozesses

Die Dynos stellen somit das Execution Environment der Heroku-Plattform dar. Anders als bei der App Engine- oder Smart-Plattform sind sie aber nicht für die Anwendungsisolation verantwortlich. Jeder Dyno wird in einer eigenen VM ausgeführt. Folglich sind die VM und der Hypervisor für die Anwendungsisolation verantwortlich. Aufgrund dieses Ansatzes muss das Execution Environment die Anwendungen nicht einschränken und kann ihnen den vollen Zugriff auf das Betriebssystem gestatten. Die Veränderungen am Betriebssystem sind allerdings nicht dauerhaft und gehen mit dem Beenden eines Dynos verloren.

⁹ Web Server Gateway Interface

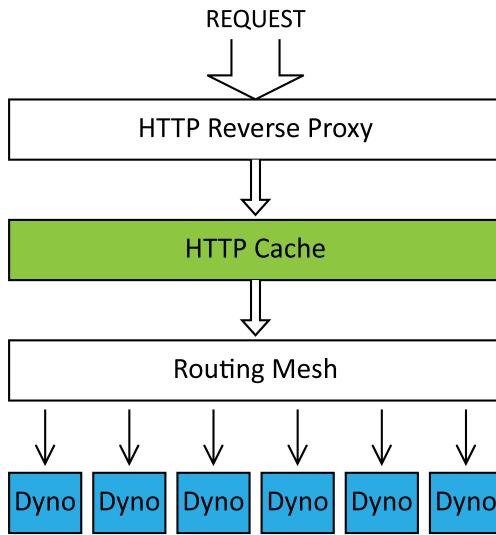


Abb. 5 Architektur der Heroku-Plattform

Eine eingehende HTTP-Request wird zunächst von einem Reverse-Proxy angenommen. Er analysiert die Request und prüft, ob sie sich mithilfe des Cache beantworten lässt. Beispielsweise lassen sich statische Ressourcen wie CSS-, Bild- oder JavaScript-Dateien sehr einfach im Cache zwischenspeichern und damit effizient ausliefern. Die Plattform fügt geeignete Ressourcen automatisch zum Cache hinzu. Falls sich die Request nicht über den Cache verarbeiten lässt, wird sie an das Routing Mesh weitergeleitet. Es verteilt die Requests gleichmäßig auf alle Dynos einer Anwendung. Darüber hinaus ist das Routing Mesh für die Verwaltung (Starten und Beenden) der Dynos zuständig (vgl. Abb. 4).

Eine Besonderheit der Dynos ist, dass sie nur eine Request gleichzeitig verarbeiten können. Falls nicht genug Dynos zur Verarbeitung aller Requests zur Verfügung stehen, verwaltet sie das Routing Mesh in einer Warteschlange. Dieses Vorgehen eliminiert potenzielle Multithreading-Probleme in den Anwendungen und Dynos. Darüber hinaus lässt sich über die Warteschlangenlänge die Auslastung der Anwendung erkennen.

Die Anwendungen werden serverseitig in einem Slug gespeichert. Dabei handelt es sich um ein Paket, das die Anwendung selbst und die benötigten Anwendungs-Bibliotheken enthält. Eine Anwendung kann von beliebigen Ruby-Bibliotheken abhängen. Beim Deployment werden die entsprechenden Bibliotheken aber nicht mit an die Server übertragen. Stattdessen lädt er sie über die RubyGems-Paketverwaltung herunter und fügt sie zum Slug hinzu. Dieses Vorgehen ermöglicht die Implementierung einer Black- bzw. Whitelist. Die Plattform gestattet aber auch den Einsatz systemnaher Pakete, die direkt auf Betriebssystem-Funktionen zugreifen.

3.3.1. Datenbank

Jede Heroku-Anwendung kann auf eine relationale Datenbank zugreifen. Der Zugriff erfolgt über die SQL-Schnittstelle und die entsprechenden Ruby-Bibliotheken. Aktuell verwendet Heroku die PostgreSQL-Datenbank. Diese ist allerdings nicht für die horizontale Skalierung ausgelegt. Ein Datenbankserver kann entweder von mehreren Anwendungen gleichzeitig oder von einer Anwendung exklusiv genutzt werden. Zusätzlich lässt sich zwischen verschiedenen Leistungsprofilen (hoher Traffic oder hohe Anzahl an Transaktionen) wählen. Diese Profile bestimmen im Wesentlichen die Speicher- und CPU-Ausstattung des Servers. Somit realisiert die Heroku-Plattform eine vertikale Skalierung der Datenbankserver.

3.3.2. Amazon Web Services

Heroku selbst stellt außer der Datenbank keine weiteren Dienste zur Verfügung. Besonders für die Ablage großer Binärdaten muss auf zusätzliche externe Dienste zurückgegriffen werden, da die PostgreSQL-Datenbank dafür nicht geeignet ist. Da die gesamte Plattform auf Basis der Amazon EC2-Plattform betrieben wird, lassen sich z. B. die Speicherdienste der Amazon Web Services nutzen (S3, SimpleDB). Dabei fallen aber weitere Nutzungskosten an. Darüber hinaus erfolgt die Kommunikation mit den AWS-Diensten über eine proprietäre Schnittstelle und führt damit zu einem Vendor Lock-in.

3.3.3. Horizontale Skalierbarkeit

Jede Heroku-Anwendung lässt sich in beliebig vielen Dynos gleichzeitig ausführen und ist damit horizontal skalierbar. Die Anzahl der Dynos wird aber nicht automatisch durch die Plattform gesteuert, sondern muss vom Benutzer über das Web-Interface fest konfiguriert werden. Bislang besteht auch keine Möglichkeit die Dynos programmgesteuert über eine API zu verwalten.

Die eingesetzte PostgreSQL-Datenbank lässt sich hingegen nicht horizontal, sondern lediglich vertikal skalieren. Da Heroku aber selbst auf dem EC2-Dienst basiert, können auch die Anwendungen auf die horizontal skalierbaren AWS¹⁰-Dienste zurückgreifen.

3.3.4. Development Environment

Die Entwicklung von Heroku-Anwendungen verläuft analog zu der Entwicklung gewöhnlicher Ruby Web-Anwendungen. Daher entfällt die Notwendigkeit eines Development Environments. Allerdings wurde die Plattform bevorzugt für den Betrieb von Ruby On Rails-Anwendungen konzipiert. Für deren Entwicklung kann auf die Tools des Ruby On Rails-Frameworks zurückgegriffen werden, das auch einen Testserver enthält. Darüber hinaus stellt die Heroku-Plattform einige Kommandozeilen-Tools zur Verfügung,

¹⁰ Amazon Web Services

über die sich die Anwendungen und die Datenbank verwalten lassen. Sie enthalten z. B. Funktionen, um neue Anwendungen zu registrieren oder die PostgreSQL-Datenbank zu administrieren.

Das Deployment einer Anwendung erfolgt ähnlich zur Smart-Plattform über eine GIT-Versionsverwaltung. Jede Anwendung erhält dazu ein eigenes GIT-Repository. Bei einem Commit erstellt Heroku einen neuen Slug aus der Head-Revision des Master-Branch. Anschließend werden alle entsprechenden Dynos neu gestartet und laden damit den aktualisierten Slug.

3.3.5. Zusammenfassung

Die Heroku-Plattform stellt eine solide Grundlage für den Betrieb von Ruby Web-Anwendungen dar. Ein Vendor Lock-in entsteht aufgrund des konsequenten Einsatzes freier Technologien nicht, außer bei Nutzung der AWS-Dienste. Den elementaren Bestandteil der Plattform bilden Dynos, über die eine horizontale Skalierung aller Anwendungen erreicht wird. Leider kann Heroku die Anzahl der Dynos nicht selbstständig anhand der Anwendungsbelastung steuern. Eine API zur Verwaltung der Dynos fehlt ebenfalls. Folglich ist eine automatische Skalierung nicht möglich. Stattdessen muss die Anzahl der Dynos manuell über die Web-Oberfläche konfiguriert werden, was besonders für Web-Anwendungen mit stark schwankender Auslastung von Nachteil ist.

Jeder Dyno wird in einer eigenen VM innerhalb der Amazon EC2-Cloud ausgeführt. Da die VM die Anwendungsisolation übernimmt, erhalten die Anwendungen vollen Zugriff auf das Betriebssystem. Allerdings besteht keine direkte Zugriffsmöglichkeit auf die VM, z. B. in Form eines SSH-Zugangs.

Die Speicherung strukturierter Daten erfolgt über eine PostgreSQL-Datenbank. Diese schränkt die horizontale Skalierbarkeit ein und unterstützt lediglich eine vertikale Skalierung des Datenbankservers.

Heroku-Anwendungen werden genau wie gewöhnliche Ruby Web-Anwendungen entwickelt. Aus diesem Grund existiert auch kein Development Environment. Das Deployment erfolgt ähnlich zur Smart-Plattform über ein GIT-Repository und weist damit dieselben Nachteile auf.

3.4. Windows Azure

Microsoft betreibt mit Windows Azure ebenfalls eine PaaS-Plattform. Die Quellcodes der entsprechenden Software-Komponenten sind nicht öffentlich verfügbar. Aus diesem Grund bezieht sich der folgende Abschnitt auf „Introducing Windows Azure“ von David Chappell {Chappell 2009 #51}.

Auf IaaS-Ebene setzt Azure eine VRS ein. Zur Hardware-Virtualisierung kommt die Virtualisierungs-Technologie des Windows Server 2008 und damit eine modifizierte Variante des HyperV-Hypervisors zum Einsatz. Innerhalb der VMs wird ein modifiziertes Windows Server 2008-Betriebssystem mit dem Namen Windows Azure ausgeführt. Auf die virtuellen Instanzen selbst erhält der Benutzer allerdings keinen Zugriff. Es ist also z. B. nicht möglich, das System über eine Remotedesktop-Verbindung zu konfigurieren. Ebenso wenig lassen sich fremde Betriebssystem-Abbilder innerhalb einer Windows Azure VM ausführen.

Beim Deployment lädt der Benutzer ein Anwendungspaket mit einer oder mehreren Anwendungen in die Plattform. Die Anwendungen werden in zwei Rollen unterteilt: Web Role-Anwendungen sind Web-Anwendungen, die im IIS¹¹ laufen und HTTP-Anfragen bearbeiten. Worker Role-Anwendungen sind hingegen beliebige Windows-Anwendungen, die als Dienst ausgeführt werden.

Jede Anwendung wird über eine Konfigurationsdatei beschrieben, in der die Rolle und Anzahl der Anwendungsinstanzen definiert wird. Für jede Anwendungsinstantz startet die Azure-Plattform eine eigene VM. Zudem wird auf jeder VM ein Fabric Agent gestartet. Er ist für die Verwaltung und das Monitoring der Anwendung zuständig und kommuniziert mit dem Fabric Controller. Dieser verwaltet wiederum alle physikalischen und virtuellen Ressourcen des Rechenzentrums bzw. eines Server-Clusters. Er ist damit auch für das Starten, Beenden und Verteilen der VMs verantwortlich.

Die Anwendungen können auf der VM beliebige Daten ablegen, diese werden aber mit dem Beenden der VM gelöscht. Für eine langfristige Datenspeicherung und zur Kommunikation zwischen den Anwendungsinstanzen stellt Azure drei verschiedene Speicherdienste bereit. Diese lassen sich konsequent über eine REST-Schnittstelle ansprechen.

3.4.1. Blob-Storage

Der Blob-Storage ist für die Speicherung großer Dateien wie z. B. Audio-, Video- oder Bilddaten konzipiert. Die Daten lassen sich mithilfe von Containern organisieren. Sie sind vergleichbar mit einem Verzeichnis, lassen sich aber nicht verschachteln. Innerhalb eines Containers lassen sich Blobs ablegen. Das sind Binärdateien mit einer maximalen Größe von 4 MB. Um eine größere Datenmenge zu speichern, können in einem Blob mehrere Blocks abgelegt werden. Die Größe eines Blocks ist wiederum auf 4 MB begrenzt. Alle Blocks eines Blobs sind verkettet, womit sich aus Sicht des Blobs eine zusammenhängende Datei ergibt. Mit diesem Ansatz lassen sich pro Blob maximal 5 TB Daten speichern.

¹¹ Internet Information Service

3.4.2. Table-Storage

Der Table-Storage ist für die Speicherung strukturierter Daten ausgelegt. Die Daten lassen sich in Form einer Tabelle ablegen. Trotz dieser Repräsentation basiert der Dienst aber nicht auf einer relationalen Datenbank. Eine Tabelle setzt sich aus einer Menge von Entities zusammen. Diese wiederum bestehen aus einer Reihe von Properties, die sich aus einem Namen, Typ und Wert zusammensetzen (vgl. Abb. 6)

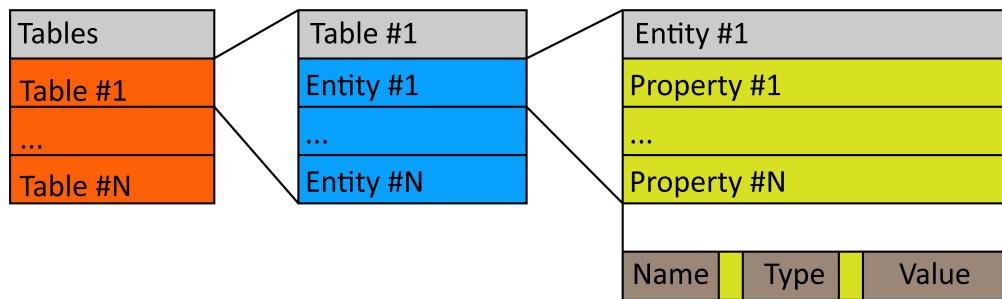


Abb. 6 Aufbau des Windows Azure Table-Storage

Vergleicht man diesen Aufbau mit Tabellen einer relationalen Datenbank, so repräsentiert eine Entity eine Zeile und eine Property eine Zelle. Im Unterschied zu einer relationalen Tabelle speichert jede Property selbst den Datentyp, den Namen und den Wert. Folglich kann innerhalb einer Tabelle jede Entity ein unterschiedliches Schema aufweisen. Durch diesen Ansatz kann sich das Tabellen-Schema jederzeit, ohne eine Migration der existierenden Daten verändern.

Der Table-Storage unterstützt keine weiteren Funktionen wie z. B. Primärschlüssel, Fremdschlüssel oder Indizes. Auch SQL-Abfragen unterstützt der Dienst nur in einer sehr eingeschränkten Form.

3.4.3. Queue-Storage

Der Queue-Storage ist nicht zur langfristigen Datenspeicherung konzipiert. Er stellt vielmehr einen robusten Kommunikationskanal zwischen den VMs dar. Im Queue-Storage kann eine beliebige Anzahl an Queues registriert werden. Jede Anwendung kann in eine Queue neue Nachrichten einfügen oder mithilfe eines Polling-Ansatzes auslesen. Damit wird eine nachrichtenbasierte Kommunikation zwischen zwei oder mehr VMS bzw. Anwendungsinstanzen ermöglicht.

Für die Entwicklung robuster Anwendungen weist der Queue-Storage ein besonderes Verhalten bei der Entnahme einer Nachricht auf. Die Nachricht wird bei ihrer Entnahme nicht automatisch aus der Queue entfernt, sondern für eine vordefinierte Zeit unsichtbar. Erst nach einer expliziten Lösch-Operation wird die Nachricht tatsächlich aus der Queue entfernt. Die explizite Lösch-Operation kann allerdings nur ausgeführt werden, solange die Nachricht unsichtbar ist. Das Verhalten ermöglicht eine ausfallsichere Verarbeitung der Nachrichten: Jede Anwendung kann eine Nachricht aus der Queue ent-

nehmen und diese verarbeiten. Nach der erfolgreichen Verarbeitung wird die Nachricht explizit aus der Queue entfernt. Schlägt die Verarbeitung allerdings fehl, verbleibt die Nachricht in der Queue und wird nach einer vordefinierten Zeit wieder sichtbar. Folglich wird die Nachricht erneut verarbeitet.

3.4.4. Weitere Dienste

Die Azure-Plattform stellt bislang keine weiteren Dienste zur Verfügung. Da die Azure-Anwendungen aber in einer eigenen VM ausgeführt werden, existieren hinsichtlich der verwendbaren Bibliotheken und Anwendungs-Komponenten keine Einschränkungen.

3.4.5. Development Environment

Für die Entwicklung von Azure-Anwendungen stellt Microsoft die Visual Studio Web Express-Version {Microsoft Express Home #134} mit einem entsprechenden Azure-Plugin zur Verfügung. Für den Test von Azure-Anwendungen auf einem Entwicklungsrechner existieren zwei Programme: Der Development Storage {About Development Storage #168} simuliert die drei Dienste Table-, Blob- und Queue-Storage unter Verwendung des Microsoft SQL-Servers. Für jeden der drei Speicherdiene implementiert er eine, zur Produktivumgebung identische REST-Schnittstelle.

Die Development Fabric ist die zweite Komponente {About the Development Fabric #167}. Sie simuliert den Fabric Controller der Produktivumgebung und ist somit für das Starten, Stoppen und Monitoring von Anwendungsinstanzen zuständig. Im Vergleich zur Produktivumgebung werden die Anwendungsinstanzen aber nicht in einer eigenen VM, sondern in einem Host-Prozess gestartet. Dieses Vorgehen ist effizienter und ressourcensparender als der Start einer VM. Zusätzlich vereinfacht dieses Konzept maßgeblich das Debugging der Anwendungen.

Zum Deployment werden die Anwendungen über die Visual Studio IDE in ein Anwendungspaket verpackt. Dieses lässt sich dann über ein Web-Interface in die Azure-Plattform laden.

3.4.6. Horizontale Skalierbarkeit

Grundsätzlich können von jeder Windows Azure-Anwendung (Web Role oder Worker Role) beliebig viele Instanzen parallel ausgeführt werden. Die Anzahl der Instanzen lässt sich über eine REST-API programmgesteuert anpassen, womit eine automatisierte Skalierung realisierbar ist. Beim Einsatz mehrerer Web Role-Instanzen verteilt ein Load-Balancer die eingehenden HTTP-Requests gleichmäßig auf alle Instanzen. Aufgrund der Skalierbarkeit unterstützt der Load-Balancer aber keine Sticky-Sessions. Damit werden die Requests einer Benutzersitzung an verschiedene Web Role-Instanzen weitergeleitet. Die Anwendungen müssen dies berücksichtigen und entsprechende Mechanismen zum Session-Handling implementieren.

Zur Skalierung von Worker Role-Anwendungen werden ebenfalls mehrere Instanzen erstellt. Typischerweise kommunizieren Worker Role-Anwendungen mit Web Role-Anwendungen über den Queue-Service. Meist erfolgt dabei aber keine direkte 1:1-Zuordnung zwischen einer Web Role- und einer Worker Role-Instanz. Dieser Aspekt muss bei der Implementierung von Anwendungen beachtet werden. Darüber hinaus müssen sie ein reentrant Verhalten aufweisen. Reentrant bedeutet in diesem Fall, dass die mehrfache Verarbeitung einer Queue-Nachricht zu demselben Endergebnis führt. Dies gilt auch im Fehlerfall, wenn eine Nachricht nur teilweise verarbeitet wurde.

Um eine geeignete Anzahl an Worker Role-Instanzen zu ermitteln, ist die Größe der Queues von Interesse. Da die Queues die Verbindung zwischen den Web Role- und Worker Role-Instanzen darstellen, ist ihre Länge direkt von der Auslastung der Worker Role-Instanzen abhängig.

3.4.7. Zusammenfassung

Die Windows Azure-Plattform wurde für den Betrieb allgemeiner Server-Anwendungen konzipiert. Aus diesem Grund unterscheidet sich ihre Architektur von der Architektur zuvor beschriebener PaaS-Provider. Dies hat allerdings zur Folge, dass die Komplexität zur Entwicklung skalierbarer Web-Anwendungen im Vergleich zu spezialisierten PaaS-Plattformen höher ist. Dies lässt sich mit dem Mehraufwand zur Implementierung der Skalierungsmechanismen begründen.

Zur Skalierung einer Azure-Anwendung lassen sich beliebig viele VMs starten. Dieser Vorgang lässt sich auch über eine REST-API automatisieren, womit die Anwendung die Kontrolle über die VMs erhält. Zur effizienten und skalierbaren Kommunikation zwischen den Anwendungsinstanzen stellt die Plattform einen speziellen Speicherdiensst bereit. Trotzdem müssen die Anwendungen speziell in Hinblick auf die horizontale Skalierung entwickelt werden. Auf der Azure-Plattform lassen sich ebenso gewöhnliche Windows-Dienste betreiben.

Zur persistenten Speicherung der Daten können verschiedene skalierbare Dienste genutzt werden. Diese ermöglichen die Speicherung großer Binärdaten oder strukturierter Daten in Form von Tabellen.

Für die Entwicklung und das Deployment der Anwendungen steht mit der Visual Studio IDE ein umfangreiches Werkzeug bereit. Zusätzlich steht ein Testserver für die lokale Entwicklung und den Test auf Entwicklungsrechnern zur Verfügung. Dieser bildet alle Schnittstellen der Azure-Plattform ab und ermöglicht eine komfortable Anwendungsentwicklung.

3.5. AppScale

AppScale {Chohan 2009 #75} wird von der University of California als Open Source-Projekt entwickelt. Die folgenden Erläuterungen beziehen sich daher auf eine Analyse der verfügbaren Quellcodes {appscale #164}.

Das AppScale-Fundament bildet eine IaaS-Plattform mit einem VRS. Als Hypervisor kann entweder XEN oder KVM verwendet werden. Alternativ ist auch die Installation auf bestehenden IaaS-Providern wie z. B. Amazon EC2 oder EUCLYPTUS {Eucalyptus | Your environment #165} möglich. Vorausgesetzt wird dabei allerdings die Kompatibilität mit den Amazon Management Tools {Amazon Web Services Developer Community #166} {Chohan 2009 #75: 5}.

Die AppScale-Architektur setzt sich aus drei Komponenten zusammen {Chohan 2009 #75: 4}:

- 1) Der AppLoadBalancer (ALB) ist für die Last-Verteilung und Benutzer-Authentifizierung zuständig.
- 2) Der AppServer (AS) führt die Anwendungen aus und stellt damit das Execution Environment dar.
- 3) Der AppController (AC) verwaltet die Plattform-Prozesse auf einem Server und stellt die Kommunikations-Schnittstelle zwischen den Anwendungs-Komponenten dar.

Eine weitere Komponente ist der DatabaseMaster (DBM), der als Schnittstelle zwischen den AppScale-Anwendungen und der HBase- oder Hypertable-Datenbank agiert. Die HBase oder Hypertable Region-Server werden als DatabaseSlaves (DBS) bezeichnet.

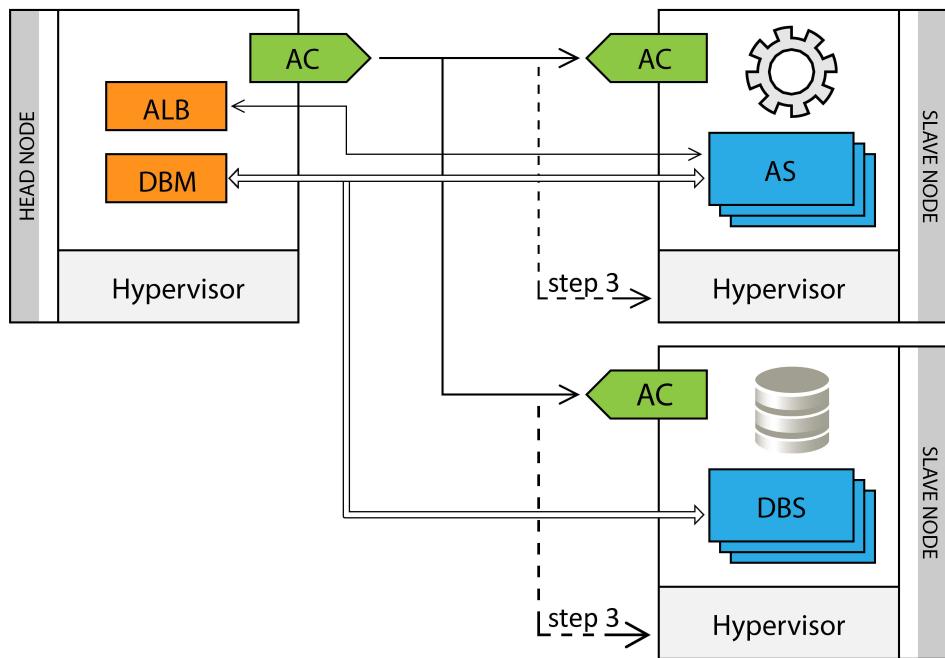


Abb. 7 Architektur der AppScale-Plattform mit mehreren VMs

Der AC¹² wird mit jeder VM beim Start des Betriebssystems ausgeführt und ist für die Verwaltung der AppScale-Prozesse und für die Kommunikation zwischen den VMs verantwortlich. Eine wesentliche Aufgabe ist somit das Bootstrapping der Plattform (vgl. Abb. 7). Der AC im Head-Node enthält dazu eine Liste aller Slave-Nodes. Der Bootstrapping-Prozess läuft in folgenden Schritten ab:

- 1) Start vom ALB¹³ auf der eigenen VM.
- 2) Start vom DBM¹⁴ auf der eigenen VM.
- 3) Initialisierung und Start der VMs für die Slave-Nodes.
- 4) Über die ACs der Slave-Nodes werden weitere AppScale-Prozesse (DBS¹⁵, AS¹⁶) gestartet. Die Liste der zu startenden Prozesse wird als Argument übergeben.

Eine weitere Aufgabe des ACs ist das Monitoring der VM. Zur Übertragung der Management-Daten wird ein Polling-Mechanismus verwendet. Der AC im Head-Node führt regelmäßig eine Ping-Methode auf den ACs der Slave-Nodes aus. Die Management-Daten werden dann als Rückgabewert übertragen. Mithilfe der Daten lassen sich Ausfälle der Slave-Nodes erkennen. Der ALB¹⁷ verwendet sie zudem zum Load-Balancing.

¹² ApplicationController

¹³ ApplicationLoadBalancer

¹⁴ DatabaseMaster

¹⁵ DatabaseSlave

¹⁶ AppServer

¹⁷ ApplicationLoadBalancer

Der ALB ist für die Verteilung der Benutzersitzungen auf die verfügbaren AS einer Anwendung verantwortlich. Die erste HTTP-Request richtet der Client-Browser grundsätzlich an den ALB. Dieser wählt anhand der Lastdaten einen geeigneten AS aus und leitet den Browser dann permanent auf diesen um. Folglich kommuniziert der Browser fortan immer direkt mit diesem AS. Darüber hinaus implementiert der ALB einen zentralen Dienst zur Authentifizierung von Anwendungsbenutzern. Die Anwendungen können diesen Dienst über eine entsprechende Schnittstelle nutzen.

Der AS bildet das Execution Environment und führt genau eine Anwendung aus. Seine Implementierung basiert auf dem Python-Entwicklungsserver, der für die Google App Engine {Google #69} bereitgestellt wird. Dieser wurde aber in verschiedenen Punkten modifiziert. Zunächst wurden die Mechanismen zur Benutzer-Authentifizierung angepasst und greifen nun, wie zuvor beschrieben, auf den ALB zurück. Die wichtigste Anpassung betrifft das Speicher-Backend. Da der originale Entwicklungsserver der App Engine keine skalierbare Datenbank implementiert, greift AppScale auf die HBase- oder Hypertable-Datenbank zurück. Diese lassen sich aber nicht direkt ansprechen. Stattdessen greift AppScale die existierende Datastore-API der App Engine auf. Die Datastore-Aufrufe werden mithilfe eines Proxies serialisiert und an den DBM übertragen. Dieser deserialisiert die Aufrufe und führt sie mithilfe der DBSs aus.

Unter einer VM lassen sich mehrere AppServer und Anwendungen parallel betreiben. Allerdings enthält der AS keine Mechanismen zur Anwendungsisolation. Somit ist dieser Ansatz nur für vertrauenswürdige Umgebungen zu empfehlen. Alternativ dazu kann pro VM auch nur ein AS ausgeführt werden. Damit sind die Anwendungen über den Hypervisor und die VM isoliert. Aufgrund der Virtualisierung lassen sich auch weiterhin mehrere Anwendungen auf einem Server betreiben.

3.5.1. Weitere Dienste

Da AppScale auf dem Entwicklungsserver der App Engine beruht, können AppScale-Anwendungen auf dieselben Dienste wie App Engine-Anwendungen zurückgreifen. Dabei ist allerdings zu beachten, dass die Dienste im Unterschied zur App Engine direkt im AS implementiert sind. Sie sind damit nicht auf eine horizontale Skalierung ausgelegt.

3.5.2. Horizontale Skalierbarkeit

Ähnlich zur App Engine kann AppScale eine Anwendung auf beliebig vielen Servern parallel ausführen. Die Anwendungen selbst lassen sich mit diesem Ansatz also horizontal skalieren. Im Unterschied zur App Engine sind die Client-Browser aber an eine AS-Instanz und damit an einen Server gekoppelt. Dieser Ansatz vereinfacht das Session-Handling, wirkt sich aber negativ auf die horizontale Skalierbarkeit aus {Henderson 2006 #55: 20}.

Die Basis der AppScale-Plattform bildet eine modifizierte und optimierte Version des App Engine Entwicklungsservers. Dieser wurde aber nicht im Hinblick auf einen produktiven Einsatz in einer horizontal skalierenden Umgebung entworfen. Daher wurde seine Datenbank-Anbindung neu implementiert. Diese Implementierung basiert auf einer horizontal skalierbaren Datenbank. Die Schnittstelle zwischen der Datenbank und den Anwendungen bildet allerdings ein einzelner, nicht horizontal skalierbarer Master-Prozess.

Aufgrund der sich ergebenden Einschränkungen bezüglich des Session-Handlings und der Datenbank-Schnittstelle, ist die horizontale Skalierbarkeit noch eingeschränkt.

3.5.3. Zusammenfassung

AppScale verfolgt das Ziel, Google App Engine-Anwendungen aus der Google-Infrastruktur zu lösen und eine Ausführung auf einer privaten Infrastruktur zu ermöglichen. Anders als in Projekten wie AppDrop {GitHub #101} wurde dabei auch der Aspekt der horizontalen Skalierung berücksichtigt.

Verschiedene Tests der Plattform zeigen {Chohan 2009 #75: 13}, dass AppScale bestimmte Anwendungen horizontal skalieren kann. Da sich die Plattform aber noch in einer frühen Entwicklungsphase befindet, lassen sich viele Anwendungen noch nicht skalieren. Besondere Einschränkungen wurden bei der Speicherschicht festgestellt {Chohan 2009 #75: 13}.

Die Installation der Plattform auf einer privaten Infrastruktur oder VMs eines IaaS-Providers ist mit einem erheblichen Administrationsaufwand verbunden. Darüber hinaus skalieren Anwendungen auf der App Engine erheblich besser als auf der AppScale-Plattform {Chohan 2009 #75: 14}. Der Einsatz von AppScale ist daher nur sinnvoll, falls sich die Anforderungen der Anwendungen nicht mit der App Engine erfüllen lassen. Ein Beispiel dafür ist die Abhängigkeit zu bestimmten proprietären Diensten.

3.6. Ergebnisse

Auf Basis der vorausgegangenen Analyse ergeben sich die wesentlichen Komponenten einer Cloud-Plattform:

- Execution Environment.
- Mechanismen zur horizontalen Skalierung.
- Dienst zur Speicherung strukturierter Daten.
- Dienst zur Speicherung von Binärdaten.

3.6.1. Execution Environment

Es kommen zwei verschiedene Execution Environments zum Einsatz. App Engine, Heroku und Smart verwenden die Laufzeitumgebung einer interpretierten Sprache wie

Java, Python oder Ruby. Microsoft schlägt mit Windows Azure eine andere Richtung ein. Grundsätzlich kommt eine VM mit dem Windows Azure Betriebssystem zum Einsatz. Auf dieser Basis lässt sich dann, abhängig zur Anwendung und Programmiersprache, ein beliebiges Execution Environment einsetzen.

Auch werden zwei verschiedene Ansätze zur Anwendungsisolation verfolgt. Windows Azure, Heroku und AppScale führen in jeder VM genau eine Anwendung aus. Diese wird damit durch die VM und den Hypervisor von anderen Anwendungen isoliert. Der Ansatz ermöglicht auch weiterhin eine relativ effiziente Nutzung der Hardware, da sich auf einem Server mehrere VMs und Anwendungen betreiben lassen. Windows Azure gibt kein Execution Environment vor, Heroku und AppScale hingegen schon. Diese schränken die Anwendungen aber nicht ein, womit sich weiterhin die gesamte Funktionalität des Betriebssystems nutzen lässt.

Die App Engine- und Smart-Plattform setzen beide ein PRS ein. Auf einem Server und unter einem Betriebssystem werden damit mehrere Execution Environments mit verschiedenen Anwendungen ausgeführt. Folglich ist das Execution Environment für die Isolation einer Anwendung verantwortlich. Dies führt zu verschiedenen Einschränkungen. Zum Beispiel sind Funktionen zum Zugriff auf das Dateisystem, Netzwerk oder das Betriebssystem nicht verfügbar. Weitere Einschränkungen betreffen z. B. die maximale Verarbeitungszeit einer HTTP-Request. Aufgrund dieser Einschränkungen lassen sich viele bestehende Anwendungs-Frameworks und Bibliotheken nicht einsetzen {Will it play in App #95}. AppScale stellt einen Sonderfall dar. Eine Anwendungsisolation lässt sich nur über VMs und einen Hypervisor erreichen. Werden auf der Plattform allerdings ausschließlich vertrauenswürdige Anwendungen betrieben, lässt sich auch auf das VRS und damit die Isolation verzichten. In diesem Fall werden ebenfalls mehrere Execution Environments gleichzeitig unter einem Betriebssystem ausgeführt.

Der Hauptnachteil einer Anwendungsisolation mithilfe eines VRS ist die signifikant höhere Ressourcen-Belastung. Jede VM benötigt eine bestimmte Mindestmenge an Ressourcen. Einige dieser Ressourcen sind für alle anderen Anwendungen blockiert, auch wenn sie nicht aktiv genutzt werden. Ein weiterer Nachteil ist die vergleichsweise hohe Konfigurations- und Startzeit einer VM, die im Minuten-Bereich anzusiedeln ist. Ein Execution Environment lässt sich hingegen innerhalb weniger Sekunden starten.

Der Einsatz eines VRS bietet aber auch Vorteile. Im Vergleich zu einem PRS werden die Anwendungen nicht durch ein Execution Environment eingeschränkt. Sie erhalten vollen Zugriff auf alle Funktionen und Ressourcen des Betriebssystems. Über den Hypervisor lässt sich eine bestimmte Menge an Ressourcen statisch einer VM zuweisen. Folglich sind klare Angaben bezüglich der Leistung und damit Servicequalität möglich.

3.6.2. Horizontale Skalierung

Alle analysierten PaaS-Provider unterstützen die horizontale Skalierung. Realisiert wird dies durch die Ausführung einer Anwendung in mehreren Execution Environments oder VMs.

Die Smart- und App Engine-Plattform stechen hervor, da sie die Anwendungen vollautomatisch skalieren. Beide Plattformen beruhen auf einem PRS. Steigt die Belastung einer Anwendung an, werden der Anwendung mehr Execution Environments zur Verfügung gestellt. Bei einer sinkenden Belastung wird ihre Anzahl wieder reduziert. Dieser Ansatz ist allerdings nur möglich, da sich Execution Environments wesentlich schneller starten und beenden lassen als VMs. Zudem ermöglichen sie aufgrund der vergleichsweise geringen Ressourcenanforderungen einen kostengünstigeren Betrieb. Aufgrund der automatischen Skalierung müssen die Anwendungen keinen Code zur Skalierung implementieren. Um allerdings eine optimale Skalierung zu erreichen, müssen bei der Entwicklung einige Regeln beachtet werden.

Die Windows Azure-, Heroku- und AppScale-Plattformen unterstützen im Gegensatz dazu keine automatische Skalierung. Im Fall von Heroku und AppScale muss die Anzahl der Execution Environments und VMs über eine Konfigurationseinstellung vorgegeben werden. Eine automatische Anpassung mithilfe einer API ist nicht möglich. Die Windows Azure-Plattform stellt hingegen eine API zur Verwaltung der VMs bereit. Darüber kann die Anwendung selbst neue VMs starten oder beenden. Eine automatische Anpassung der genutzten Ressourcen an die tatsächliche Anwendungs-Belastung ist daher möglich.

3.6.3. Datenspeicherung

Alle Plattformen stellen einen Dienst zur Speicherung strukturierter Daten bereit. Smart und Heroku verwenden dazu eine relationale Datenbank. Die Smart-Plattform kapselt diese allerdings über eine proprietäre API.

Windows Azure, App Engine und AppScale verwenden hingegen keine relationale Datenbank, sondern horizontal skalierbare und proprietäre Speicherlösungen. Die App Engine setzt die spaltenbasierte Bigtable-Datenbank auf Basis des GFS-Dateisystems ein. Ähnlich dazu verwendet AppScale das HDFS-Dateisystem mit der HBase- oder Hypertable-Datenbank. Windows Azure basiert hingegen auf einem Entity-basierten, horizontal skalierbaren Speicherdiensst, zu dem keine genaueren Informationen vorliegen.

Die Smart- und Azure-Plattform enthalten darüber hinaus noch einen Dienst zur Speicherung großer Datenmengen. Dies sind Dateien mit einer Größe ab ca. 1-5 MB. Die Smart-Plattform greift auf das MogileFS-Dateisystem (vgl. S. 55, MogileFS) zurück. Windows Azure verwendet wiederum einen proprietären, horizontal skalierbaren

Dienst, in dem sich große Datenmengen in fragmentierter Form speichern lassen. Der Zugriff auf die Speicherdiene ist erfolgt in beiden Fällen über eine proprietäre API. Die Heroku-Plattform bietet selbst keine Möglichkeit, Binärdaten effizient zu speichern. Allerdings können die Anwendungen auf die Amazon AWS-Dienste zurückgreifen, die ebenfalls auf eine horizontale Skalierbarkeit ausgelegt sind. App Engine und AppScale verwenden beide eine spaltenbasierte Datenbank, die grundsätzlich mit Binärdaten arbeitet. Aus diesem Grund lassen sich Dateien bis zu einer Größe von ca. 10 MB effizient in der Datenbank ablegen. Größere Dateien können hingegen nur in fragmentierter Form, ähnlich zu Azure, gespeichert werden.

3.6.4. Monitoring-Mechanismen

Über Monitoring-Mechanismen wird die Performance aber auch der Ressourcenverbrauch einer Anwendung überwacht und protokolliert. Mithilfe der gesammelten Informationen lassen sich Anwendungsfehler oder Performanceengpässe identifizieren, womit sich die Anwendung gezielt optimieren lässt. Bislang implementieren lediglich Smart und die App Engine detaillierte Monitoring-Mechanismen. Die App Engine überwacht die ausgeführte Anwendung selbst und die Plattform. Damit können Fehler der Plattform von Anwendungsfehlern abgegrenzt werden. Die Smart-Plattform implementiert ebenfalls Monitoring-Mechanismen. Die erfassten Daten werden allerdings noch nicht aufbereitet und in nützlicher Form dargestellt.

3.6.5. Weitere Dienste

Um die Anwendungsentwicklung zu vereinfachen, die Performance zu verbessern oder die durch das Execution Environment entstandenen Defizite auszugleichen, enthalten die analysierten PaaS-Angebote meist zusätzliche Plattform-Dienste. Häufig kann z. B. auf einen Memcache-Cluster, Dienste zur Bildverarbeitung, E-Mail oder Instant Messaging zurückgegriffen werden. Diese Plattform-Dienste stellen allerdings keinen zentralen Bestandteil der Plattformen dar, weshalb sie an dieser Stelle nicht weiter betrachtet werden.

3.6.6. TwoSpot

Unter Berücksichtigung der Anforderungen des Fojobo-Projekts (vgl. S. 1, Hintergrund und Motivation) wird zur Realisierung von TwoSpot eine IaaS-Ebene mit einem PRS eingesetzt. Auf Basis des PRS kommt ein neu entwickeltes Execution Environment zum Einsatz, das auch die Anwendungsisolation übernimmt.

Eine Anforderung an TwoSpot ist die Vereinfachung der Entwicklung skalierbarer Web-Anwendungen. Wie zuvor beschrieben (vgl. S. 35), unterstützen Execution Environments den Entwickler bei der Entwicklung skalierbarer Web-Anwendungen. Daher kommt in TwoSpot ein Execution Environment zum Einsatz. Ähnlich dem Ansatz der Heroku-Plattform, lassen sich die Vorteile eines VRS und Execution Environments

kombinieren. Dieser Ansatz ist aber nur von Interesse, wenn eine vordefinierte Service-Qualität und Leistung garantiert werden muss, oder die Isolations-Mechanismen eines Execution Environments zu nicht hinnehmbaren Einschränkungen führen. Beides ist bei Fojobo allerdings nicht der Fall.

Gegen die Isolation mit einem VRS sprechen aber noch weitere Gründe. Entsprechend der Anforderungen aus dem Fojobo-Projekt muss die Plattform sehr viele Anwendungen gleichzeitig verwalten und ausführen. Allerdings ist davon auszugehen, dass ein signifikanter Bruchteil nicht oder nur sehr selten genutzt wird. Solche brachliegenden Anwendungen dürfen keine Ressourcen in Anspruch nehmen. Diese Anforderung lässt sich mit einem VRS allerdings nicht realisieren. Wie zuvor beschrieben (vgl. S. 34), ist die Startzeit einer VM signifikant höher als die Startzeit eines Execution Environments. Aus diesem Grund lässt sich die VM und Anwendung nicht erst im Bedarfsfall starten. Folglich muss für jede Anwendung immer mind. eine VM ausgeführt werden. Damit verbrauchen auch brachliegende Anwendungen Ressourcen.

Ein Execution Environment auf Basis eines PRS verhält sich im Vergleich zu einem VRS agiler bezüglich Lastveränderungen. So lassen sich äußerst kurzfristig neue Anwendungsinstanzen starten oder bestehende beenden. Besonders für Web-Anwendungen ist diese Funktionalität wichtig, um sprunghafte Lastanstiege (vgl. Slashdot Effect, {Geeknet, Inc #41}) zu bewältigen.

Die Isolation des Execution Environments führt allerdings zu Einschränkungen bei der Anwendungsentwicklung. In Kombination mit proprietären APIs der Plattform-Dienste führt dies oftmals zu einem Vendor Lock-in (vgl. App Engine, Smart und Azure). Daher ergibt sich für TwoSpot die Anforderung: Der Vendor Lock-in soll durch den Einsatz offener Technologien und Standards so weit wie möglich reduziert werden.

4. Technologien

Für die Implementierung von TwoSpot wird aus mehreren Gründen Java eingesetzt. Ein wesentlicher Aspekt ist, dass im Java-Umfeld bereits eine beachtliche Menge an wiederverwendbaren Komponenten zur Entwicklung von Server-Technologien existiert. TwoSpot selbst soll primär auf Linux-Systemen betrieben werden, muss aber für die Entwicklung auf möglichst vielen Plattformen ausführbar sein. Somit ist auch die Plattformunabhängigkeit von Java ein wesentlicher Aspekt. Da die Realisierung von TwoSpot strengen Zeitgrenzen unterlegen war, spielte auch meine Erfahrung im Java-Umfeld eine ausschlaggebende Rolle. Der Einsatz einer vergleichsweise fremden Programmiersprache hätte die Entwicklung ausgebremst und das Risiko einer Fehlentwicklung erhöht. Für die Realisierung von Web-Anwendungen selbst erweist sich Java allerdings als umständlich. Auf Basis der Java VM lassen sich aber weitere Sprachen wie z. B. JavaScript (Rhino {Happy Cog Studios - <http://www.happycog.com> #124}), Python (Jython {Python Software Foundation #63}) und Ruby (JRuby {JRuby.org 2/23/2010 #128}) betreiben. Aufgrund dieser Eigenschaften bildet die Java VM ein solides Fundament für die Entwicklung von TwoSpot.

4.1. Anwendungsserver

Für den Betrieb von Java Web-Anwendungen wird ein Anwendungsserver benötigt, der mindestens die Java Servlet 2.5-Spezifikation implementiert. Zwei sehr bekannte Produkte sind Apache Tomcat {The Apache Software Foundation #83} und Jetty {The Eclipse Foundation #64}. Zusätzlich existiert aber noch eine Vielzahl weiterer, häufig kleinerer und speziellerer Server {Open Source Web Servers 2/21/2010 #127}. Aufgrund des zeitlichen Arbeitsumfangs wurden diese aber nicht näher betrachtet.

Zwei wesentliche Anforderungen an den Anwendungsserver sind eine hohe Startgeschwindigkeit und ein geringer Ressourcenverbrauch. Aufgrund der speziellen Anforderungen von TwoSpot ist davon auszugehen, dass Modifikationen am Server-Quellcode erforderlich sind. Folglich sind auch die Quellcode-Qualität und die Dokumentation wesentliche Auswahlkriterien.

Bezüglich der Startgeschwindigkeit und des Ressourcenverbrauchs erscheinen Jetty und Tomcat in etwa gleichwertig. Eine Quellcode-Analyse ergab hingegen signifikante Unterschiede. Die Tomcat-Quellcodes und die Architektur wirken monolithisch, womit Anpassungen maßgeblich erschwert werden. Jetty ist hingegen vollständig modularisiert und die Größe eines Moduls ist durchwegs überschaubar. Dieses Baukastenprinzip eignet sich besonders für den embedded Betrieb und ermöglicht eine Anpassung an die speziellen Bedürfnisse von TwoSpot. So lassen sich z. B. Server-Konfigurationen erstellen, die lediglich die tatsächlich benötigten Server-Module enthalten. Unnötige Module müssen damit nicht in den Speicher geladen werden.

Bezüglich der Dokumentation ist der Tomcat-Server besser aufgestellt. Für Tomcat existiert bereits eine Vielzahl von Zeitungsartikeln, Büchern, Foren-Beiträgen und auch die Community erscheint im Vergleich zu Jetty größer. Dies ist besonders für die Behebung seltener Probleme von Vorteil. Die Dokumentation des Jetty-Servers wirkt hingegen schlecht strukturiert. Außerdem leidet sie unter den vielen Server-Versionen und deren Differenzen.

Jetty sticht besonders bei der Entwicklungsgeschwindigkeit und der Anpassung an neue Standards hervor. Aber auch sinnvolle Technologien wie z. B. Continuations werden sehr schnell, auch ohne eine vorliegende Spezifikation eingebunden. Die proprietären Schnittstellen werden später an die Spezifikationen angepasst oder neu entwickelt. Ein Nachteil dieser Aktualität ist die entstehende Inkompatibilität neuer Versionen gegenüber Älteren. Dieser Nachteil wird aber durch eine klare Versions-Struktur und die ständige Wartung älterer Versionen entschärft.

Aufgrund der modularen Architektur und der hohen Entwicklungsgeschwindigkeit wird Jetty für die Entwicklung von TwoSpot eingesetzt. Die beiden Vorteile gleichen die vergleichsweise schlechte Dokumentation aus.

4.2. Datenbank

Klassische relationale Datenbanken (RDB) sind nicht als verteilte Datenbanken konzipiert und weisen daher Einschränkungen bezüglich der Skalierung auf (vgl. CAP-Theorem) {Schlossnagle 2006 #53}{Elson #147: 173}. Es existieren trotzdem verschiedene Ansätze, um Lese- und Schreibzugriffe auf Basis relationaler Datenbanken zu skalieren. Ein Ansatz ist der Master/Slave-Betrieb {Schlossnagle 2006 #53}. Statt einem Datenbankserver werden mind. zwei eingesetzt. Einer übernimmt die Rolle des Masters, die restlichen Server werden als Slave bezeichnet. Alle Schreibzugriffe müssen über den Master ausgeführt werden. Dieser repliziert die Datenänderungen an alle Slave-Server (vgl. Abb. 8). Da somit alle Server über denselben Datenbestand verfügen, lassen sich Lesezugriffe auf dem Master- und jedem Slave-Server ausführen. Folglich können die Lesezugriffe auf alle verfügbaren Server verteilt werden, womit eine horizontale Skalierung erreicht wird. Schreibzugriffe lassen sich hingegen nicht skalieren, da sie immer über einen einzelnen Master-Server abgewickelt werden. Die Skalierung der Lesezugriffe ist allerdings von besonderer Bedeutung, da Web-Anwendungen häufig ein Lese-/Schreibverhältnis zwischen 80/20 und 90/10 aufweisen {Henderson 2006 #55: 44}. Da lediglich ein Master eingesetzt wird, entsteht allerdings ein SPOF¹⁸. Dies ist besonders in hochverfügbar Systemen problematisch.

¹⁸ Single Point of Failure

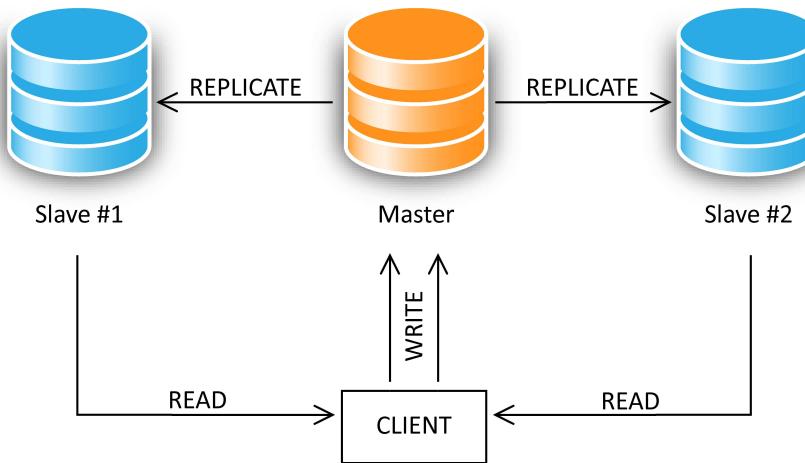


Abb. 8 Master/Slave-Betrieb einer relationalen Datenbank

Die Multi-Master-Replikation, eine Erweiterung des Master/Slave-Ansatzes, wird oftmals zur Skalierung der Schreibzugriffe aufgeführt. Anstelle eines einzigen Masters kommen mehrere zum Einsatz. Damit lassen sich die Schreibzugriffe über mehrere Server verteilen. Jeder Schreibzugriff wird vom Master an alle anderen Master repliziert. Damit muss auch in diesem Szenario jeder Server alle Schreibzugriffe verarbeiten. Folglich wird die Schreibleistung wieder durch die Leistungsfähigkeit der einzelnen Server limitiert. Somit eignet sich die Multi-Master-Replikation nicht zur Skalierung von Schreibzugriffen {Schlossnagle 2006 #53}{How to scale writes #171}. Weiterhin entsteht durch die Replikation zwischen den Master-Servern ein Overhead, der sich negativ auf die Schreibleistung auswirkt. Ein weiteres Problem tritt bei Netzwerk-Partitionen in Form des sog. Split-Brain-Szenarios auf {Managing Computers with Automation #172}. Auch wenn die Master-Server die Schreibzugriffe nicht replizieren können, führen sie weitere Schreibzugriffe aus. Die Folge ist ein inkonsistenter Datenbestand zwischen den Servern. Im Vergleich zu einem Master/Slave-Ansatz weist der Multi-Master-Ansatz allerdings keinen SPOF mehr auf.

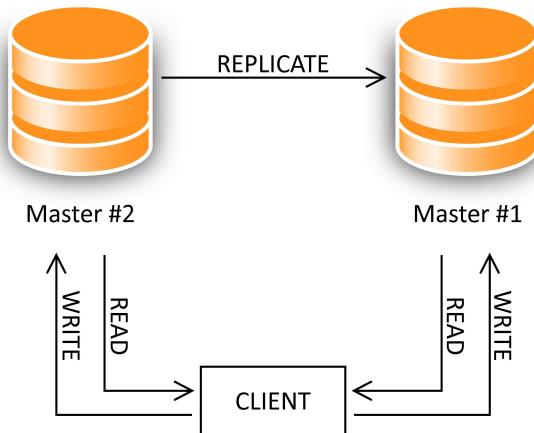


Abb. 9 Multi-Master-Replikation einer relationalen Datenbank

Die Master/Slave- und Multi-Master-Ansätze ermöglichen also nur eine Skalierung der Lesezugriffe. Die Schreibzugriffe und die Datenbankgröße lassen sich hingegen nicht skalieren. Abhilfe schafft das Sharding bzw. die vertikale oder horizontale Partitionierung einer Datenbank {High Scalability #173}. Dabei wird der Datenbestand in mehrere Shards zerschnitten, wovon jeder einem Datenbankserver oder einer Master/Slave-Kombination zugewiesen wird. Die Aufteilung des Datenbestandes kann z. B. anhand funktionaler Einheiten erfolgen (vertikale Partitionierung). Alternativ dazu lassen sich große Tabellen in mehrere Teile zerlegen (horizontale Partitionierung).

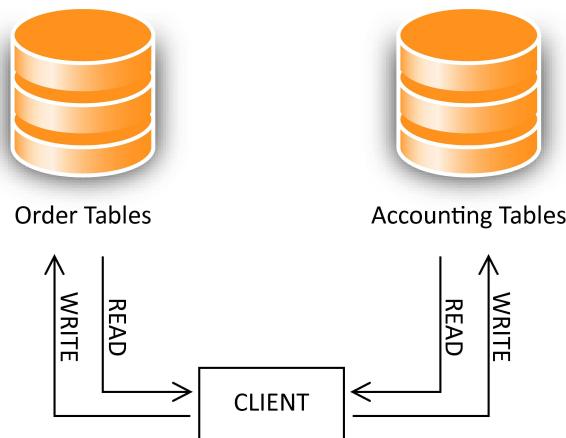


Abb. 10 Sharding (vertikale Partitionierung) mit zwei Datenbankservern

In jedem Fall übernimmt ein Server alle Lese- und Schreibzugriffe für einen Shard und damit nur für einen Teil der Datenbank. Die Zugriffe für die gesamte Datenbank verteilen sich somit auf alle Server. Folglich ermöglicht dieser Ansatz eine Skalierung der Lese-, Schreibzugriffe und der Datenbankgröße.

Ein wesentliches Problem entsteht, wenn die Datenmenge für einen Shard zu groß wird. Dann muss ein Rebalancing durchgeführt werden, das die Daten auf mehrere Shards verteilt {High Scalability #173}. Weiterhin wird die Ausführung von Queries erschwert. Da die Daten auf mehreren Servern gespeichert werden, müssen auch mehrere Queries ausgeführt werden {High Scalability #173}. Außerdem ist die Anwendung selbst für die Aufteilung und Verwaltung der Shards verantwortlich. Das Datenbank-Schema muss also bereits auf Sharding ausgerichtet sein, womit sich wiederum die Anwendungskomplexität erhöht {High Scalability #173}.

Die beschriebenen Ansätze lassen sich mit frei verfügbaren RDBs implementieren. Kommerzielle Lösungen wie z. B. Oracle RAC {Oracle Real Application Clusters 11g #175} oder der Microsoft SQL Server {SQL Server 2008 Overview 3/4/2010 #176} bieten weitere Ansätze, die eine wesentlich bessere Skalierung ermöglichen. Da als Basis von TwoSpot aber nur freie Technologien eingesetzt werden sollen und der Fokus auf der horizontalen Skalierung liegt (vgl. S. 1, Hintergrund und Motivation), wurden diese Lösungen nicht genauer betrachtet.

Relationale Datenbanken weisen speziell in Bezug auf Web-Anwendungen noch einen weiteren Nachteil auf: Sie benötigen ein fest definiertes Schema. Eine Veränderung des Schemas erfordert eine Migration des gesamten Datenbestandes. Dies wird z. B. über Skripte realisiert, die den alten Datenbestand an das neue Schema anpassen. Bei vielen Datenbanken sind die Daten während einer Schema-Migration nicht oder nur teilweise nutzbar, womit auch die Web-Anwendung nicht mehr nutzbar ist. Dieses Verhalten ist besonders bei großen Datenbeständen kritisch {How FriendFeed uses MySQL #205}. Web-Anwendungen unterliegen typischerweise kurzen Entwicklungszyklen, womit sich das Schema häufig verändert. Durch die Schema-Migration entsteht damit ein hoher Arbeitsaufwand und durch den Datenbankausfall entstehen möglicherweise finanzielle Ausfälle. Diese Nachteile werden häufig nicht durch den Funktionsumfang einer RDB ausgeglichen, da Web-Anwendungen oftmals nur die CRUD¹⁹-Funktionalität der Datenbank benötigen.

Aufgrund der aufgeführten Nachteile wurde für TwoSpot keine RDB eingesetzt. Eine aufstrebende Alternative bilden NoSQL²⁰-Datenbanken {Edlich 2/20/2010 #107}, wie z. B. dokumentenbasierte, spaltenbasierte, graphbasierte oder key-value Datenbanken. Für TwoSpot wurden unter dem Aspekt der horizontalen Skalierbarkeit eine Reihe vielversprechender Produkte betrachtet: Apache Cassandra {Apache Software Foundation #67}, Apache HBase {Apache Software Foundation #66}, Hypertable {Hypertable #65}, MongoDB {10gen #80}.

Für eine genauere Analyse dieser Lösungen muss zunächst das, von Eric Brewer auf der PODC 2000 vorgestellte CAP-Theorem {Brewer 16.7.2000 #84} eingeführt werden, das von Gilbert und Lynch {Gilbert 2002 #108} nachgewiesen wurde. Das CAP-Theorem besagt, dass ein verteiltes System nicht gleichzeitig konsistent (Consistency), verfügbar (Available) und tolerant gegenüber Netzwerk-Partitionen (Partition tolerance) sein kann. Lediglich zwei dieser Eigenschaften lassen sich gleichzeitig erfüllen. Allerdings müssen nicht zwei Eigenschaften exklusiv gewählt werden, auch Kombinationen sind möglich. Zum Beispiel kann ein System auch konsistent, teilweise verfügbar und teilweise tolerant gegenüber Netzwerk-Partitionen sein. Bei den analysierten Technologien handelt es sich um verteilte, skalierbare und hochverfügbare Speichersysteme. Nach dem CAP-Theorem müssen sie also bestimmten Einschränkungen unterliegen.

4.2.1. MongoDB

MongoDB {10gen #80} wird von 10gen entwickelt und stellt eine dokumentenorientierte Datenbank dar. Die Datenbank wurde in C++ implementiert, lässt sich aber über Language-Bindings mit verschiedenen Sprachen nutzen. Der Zugriff erfolgt über eine

¹⁹ Create Read Update Delete

²⁰ Not only SQL

proprietäre API, eine Integration mit offenen Standards wie JDO oder JPA existiert noch nicht.

Zum Speichern von Dokumenten wird kein Schema benötigt, da dieses implizit über das Dokument angegeben wird. Dokumente werden in einer JSON²¹-ähnlichen Datenstruktur definiert. Der Aufbau des Dokuments und damit das Schema kann sich mit jedem Dokument verändern, ohne eine Migration existierender Dokumente durchführen zu müssen. Auf gespeicherte Dokumente kann am einfachsten über einen Iterator zugegriffen werden. Darüber hinaus steht eine einfache Query-Sprache zur Verfügung, die allerdings nicht auf SQL beruht. Ihr Funktionsumfang umfasst die üblichen Vergleichsoperatoren kleiner, größer, gleich, ungleich und einige komplexere Operatoren, um z. B. den Inhalt von Arrays zu vergleichen. Zur Beschleunigung der Queries kommen ähnliche zu RDBs Indizes zum Einsatz. Diese müssen allerdings explizit definiert werden.

Zur ausfallsicheren Datenspeicherung wird eine Master/Slave-Replikation unterstützt. Ähnlich zu RDBs müssen dabei alle Schreibzugriffe über einen Master-Knoten abgewickelt werden. Folglich können Schreibzugriffe nicht skalieren. Da jeder MongoDB-Knoten als Master agieren kann, wird automatisch ein Knoten als Master bestimmt. Fällt dieser aus, wird ein anderer Knoten ausgewählt. Somit ergibt sich kein SPOF.

Um eine horizontale Skalierbarkeit zu erreichen, enthält MongoDB einen sog. Auto-Sharding-Mechanismus. Dabei werden die Dokumente automatisch über alle verfügbaren Server verteilt und repliziert. Um die Leistungsfähigkeit des Systems zu erhöhen, können inkrementell neue Server hinzugefügt werden.

²¹ Java Script Object Notation

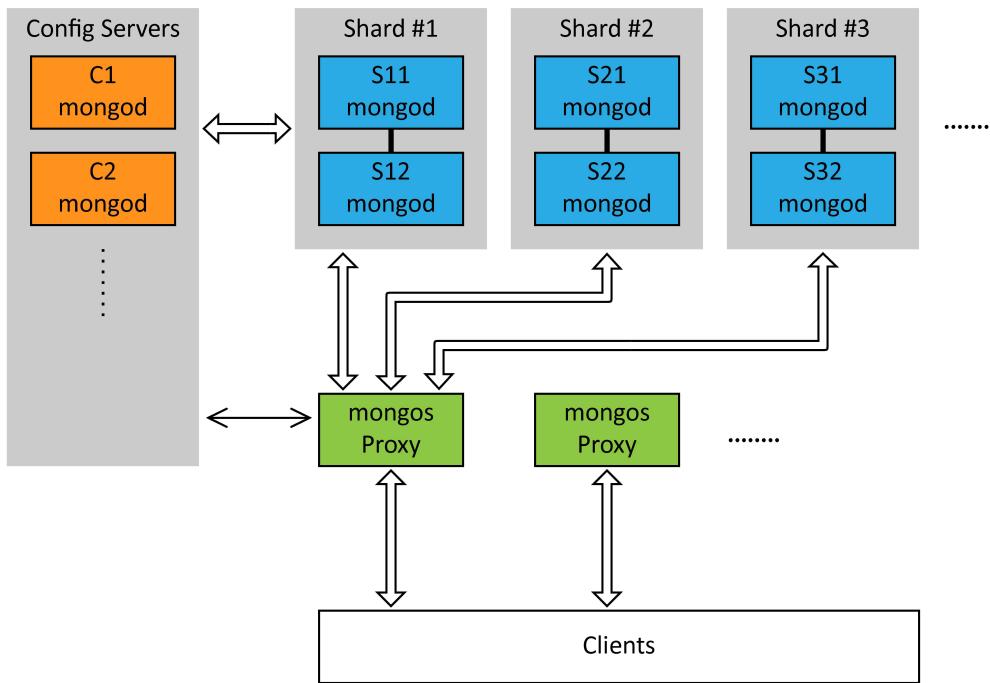


Abb. 11 MongoDB Server-Architektur beim Einsatz der Auto-Sharding-Funktion

Ein Shard besteht aus einem oder zwei Servern. Normalerweise werden zwei Server eingesetzt, womit sich die Daten innerhalb eines Shards mit einem Replikationsfaktor von zwei replizieren lassen. Die Dokumente werden vom Client innerhalb einer Sharded Collection gespeichert. Die Collection wird in mehrere Chunks aufgeteilt, die von den Shards verwaltet werden. Dazu wird die Collection allerdings nicht in gleichmäßige Stücke zerteilt. Stattdessen werden die Elemente nach einem zufälligen Muster ausgewählt und zu den Chunks hinzugefügt.

Zum Speichern eines neuen Dokuments kommuniziert der Client mit einem zufällig gewählten Proxy-Server. Um eine horizontale Skalierbarkeit zu ermöglichen, lassen sich beliebig viele Proxy-SERVER einsetzen. Der Proxy-Server ermittelt die Adresse des Shards und Chunks, in dem das Dokument gespeichert werden soll. Dazu ruft er einen Config-Server auf, der alle Metadaten der Shards und Chunks verwaltet. Aufgrund der horizontalen Skalierbarkeit lassen sich auch beliebig viele Config-SERVER einsetzen, die ihre Daten untereinander replizieren. Anschließend wird der Client-Aufruf an den Shard weitergeleitet. Dieser schreibt das Dokument in den Chunk. Sobald ein Chunk ein Speichervolumen von 50 MB erreicht, wird er zerteilt. Die entstandenen Teile werden den Shards, mit der geringsten Belastung zugewiesen. Aktuell erlaubt MongoDB lediglich den Einsatz von 20 Shards. Zukünftig sollen aber bis zu 1000 Shards möglich sein.

MongoDB stellt eine sehr interessante Datenbank für die Entwicklung von Web-Anwendungen dar. Besonders die einfache dokumentenorientierte Speicherung und API stechen positiv hervor. Der Auto-Sharding-Mechanismus befand sich zum Zeitpunkt der Evaluation noch in einem Alpha-Stadium, weshalb diesbezüglich keine Aus-

sage getroffen werden kann. Beim Einsatz einer Master/Slave-Replikation lassen sich Schreibzugriffe nicht horizontal skalieren. Ein weiterer Nachteil der MongoDB ist die fehlende Anbindung an standardisierte Schnittstellen wie JDO oder JPA. Da sich die horizontale Skalierung noch im Alpha-Stadium befand und die Master/Slave-Replikation zumindest für Schreibzugriffe nicht skaliert, wird die MongoDB nicht für TwoSpot eingesetzt.

4.2.2. Bigtable, Hypertable und HBase

Hypertable und HBase wurden beide nach dem Vorbild der Google Bigtable-Datenbank {Fay 2006 #82} implementiert. Ziel ist die Bereitstellung einer hochverfügbaren, konsistenten sowie skalierbaren {Fay 2006 #82} Datenbank auf Basis des Google File Systems {Ghemawat #110}. Somit bevorzugen Bigtable und damit HBase und Hypertable die CA-Eigenschaften des CAP-Theorems.

Die Hypertable-Datenbank wurde nicht weiter analysiert, da sie für einen produktiven Einsatz noch nicht ausgereift erschien und im Vergleich zu HBase eine kleinere Community aufweist. Stattdessen wird nachfolgend die HBase-Datenbank genauer betrachtet.

Das Datenmodell von HBase ist identisch zur Google Bigtable-Datenbank. Es lässt sich kurz als schwach besetzte, verteilte, persistente, mehrdimensionale und sortierte Map-Datenstruktur beschreiben {Apache Software Foundation #66: 1}. Der Map-Index besteht aus einem Row-Key, Column key und einem Zeitstempel. Ein gespeichertes Objekt ist ein nicht weiter interpretierter Byte-Array. Die Datenstruktur lässt sich auch als Tabelle darstellen (vgl. Abb. 12). Der Row-Key ist ein frei definierbarer, aber innerhalb der Tabelle eindeutiger Byte-Array. Die Zeilen sind in einer aufsteigenden Reihenfolge nach dem Row-Key sortiert. Eine Tabelle kann beliebig viele Spalten, gruppiert in Column families enthalten. In einer Zelle wird ein beliebig langer Byte-Array gespeichert. Allerdings wird nicht nur der aktuelle Zelleninhalt gespeichert, sondern auch vorherige Versionen des Inhalts. Adressiert werden sie über einen Zeitstempel, der sich entweder vorgeben lässt oder von der Datenbank generiert wird.

Row-Key	Column family: User			
	Column: Name		Column: Birth	
Alex	t32	Alex		
	t2	Aley		
	t0	Axel	t0	23.02.1985
Bob			t1	14.10.1978
	t0	Bob	t0	14.10.1976

Abb. 12 Eine HBase-Tabelle mit einer Column family, mehreren Spalten und Zellen-Versionen

Abhängig zu ihrer Größe (Speicherplatz) wird eine Tabelle horizontal anhand des Row-Keys in mehrere Abschnitte unterteilt. Ein Abschnitt wird dabei als Region bezeichnet. Ihre Adressierung erfolgt über den Tabellennamen und den letzten gespeicherten Row-Key. Aufgrund der aufsteigend sortierten Speicherung anhand des Row-Keys, können neue Daten direkt in die Regionen eingefügt werden (vgl. Abb. 13). Sobald eine Region über eine bestimmte Größe anwächst (z. B. 65 MB), wird sie zweigeteilt.

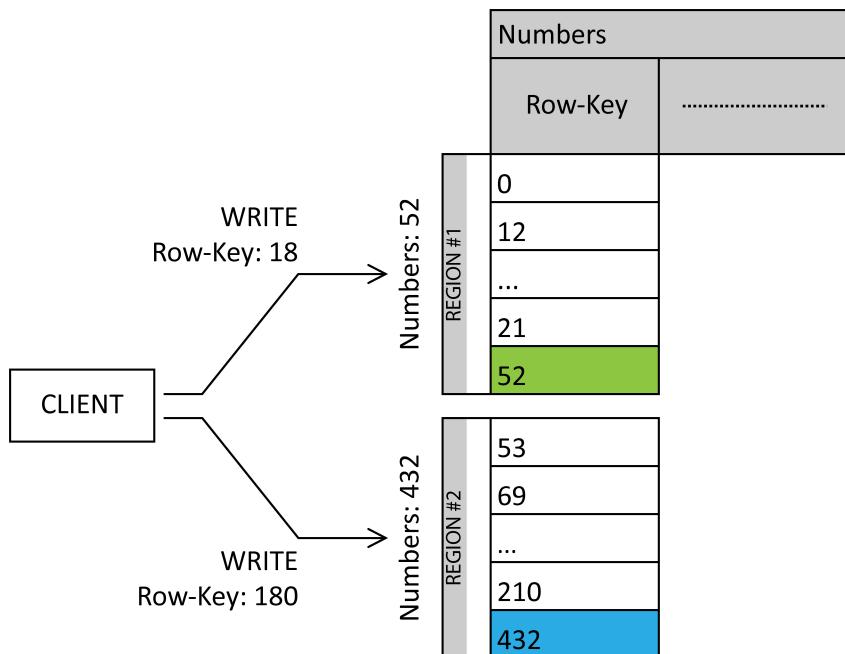


Abb. 13 Eine Tabelle wird in mehrere Regionen aufgeteilt. Schreibzugriffe erfolgen direkt in den Regionen.

Jede Region wird von genau einem Region-Server verwaltet. Dieser wickelt alle Schreib- und Lesezugriffe auf ihre Daten ab. Die HBase-Clients kommunizieren dazu direkt mit dem Region-Server. Sie müssen daher die Adresse des entsprechenden Region-Servers bzw. der Region ermitteln. Dies geschieht über einen dreistufigen Lookup-Mechanismus. Zunächst ermitteln sie über den ZooKeeper, ein zentraler Konfigurations- und Synchronisations-Dienst, die Adresse einer ROOT-Region. Diese verweist auf eine Menge von Regionen der META-Tabelle. Die Einträge der META-Tabelle zeigen abschließend auf die User-Regionen. Das sind die, in HBase gespeicherten Regionen einer Tabelle.

Die Einträge der ROOT- und META-Tabelle setzen sich aus einem Namen und einem Row-Key zusammen. Der Name ist gleich dem Tabellennamen der User-Region. Der Row-Key eines Eintrags in den META-Regionen ist gleich dem letzten Row-Key der referenzierten User-Region. Eine META-Region kann auf mehrere User-Regionen derselben Tabelle verweisen. Aus diesen Einträgen wird nun der Eintrag mit dem größten Row-Key ermittelt. Dieser Row-Key wird im Eintrag der ROOT-Region verwendet, um mit dem Tabellennamen auf die META-Region zu verweisen (vgl. Abb. 14).

Da sich die ROOT-Region nicht in mehrere Regionen zerspalten lässt, kann die Adresse einer User-Region immer in drei Lesezugriffen ermittelt werden.

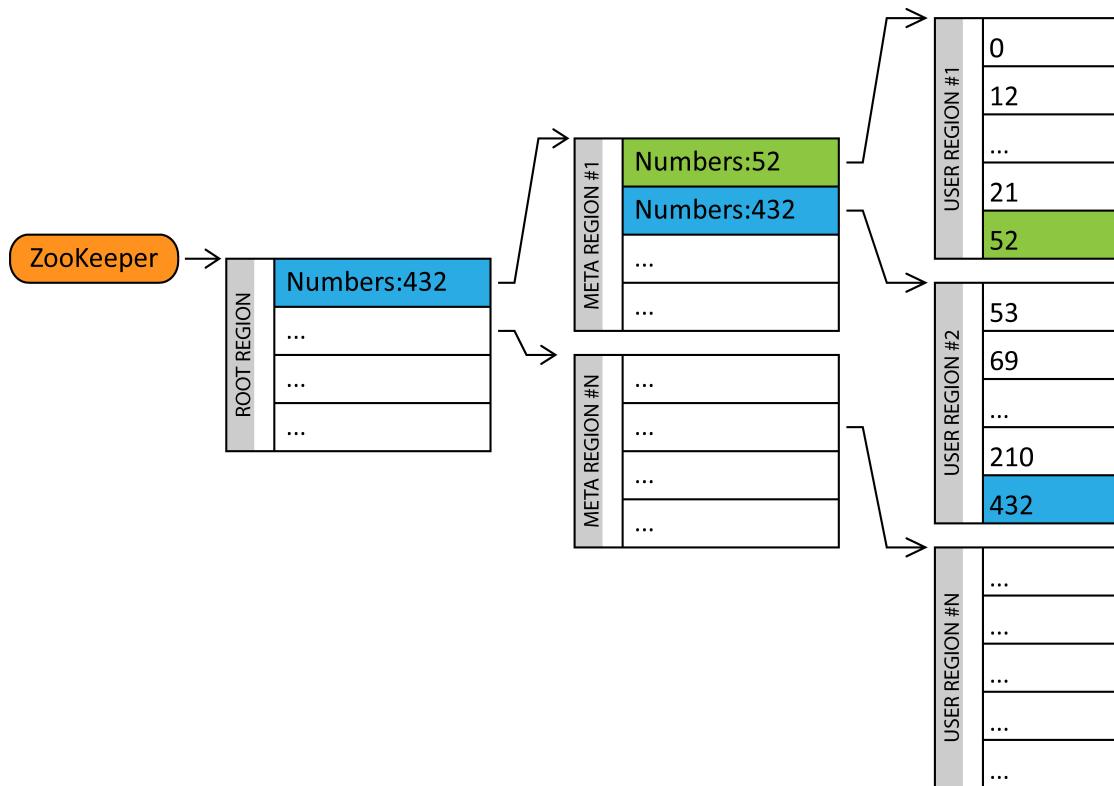


Abb. 14 Dreistufiger Lookup der Adresse einer User-Region bzw. des Region-Servers

Um die ROOT- und META-Regionen nicht zu überlasten und die Performance zu erhöhen, enthält jeder Client einen Cache. Er speichert alle bekannten Informationen aus den beiden Tabellen.

Die Grundlage von HBase bildet das Hadoop Distributed File System (HDFS) {Apache Software Foundation #81}, das im Abschnitt Hadoop Distributed File System auf S. 52 genauer beschrieben wird. HBase speichert alle Daten im HDFS und profitiert somit von dessen Replikationsmechanismus und der hohen Datenverfügbarkeit. Physikalisch wird die Tabelle anhand der Column families gespeichert. Die Zellen werden dabei kompakt in einer Datei abgelegt, wobei leere Zellen keinen Speicherplatz benötigen. Aufgrund dieses Speicherformats werden Bigtable, HBase und Hypertable auch als spaltenorientierte Datenbanken bezeichnet.

Je größer eine HBase-Tabelle wird, desto mehr Regionen entstehen. Jede Region kann durch einen eigenen Server verwaltet werden. Er übernimmt dabei alle Lese- und Schreibzugriffe. Folglich lassen sich die Datenbankgröße, die Lese- und Schreibzugriffe horizontal durch das Hinzufügen zusätzlicher Server skalieren. Da eine Region aber immer nur von einem Server verwaltet wird, lässt sich ihre Belastung nicht auf mehrere Server verteilen. Auch richtet sich ihre Größe nach dem benötigten Speicherplatz und

nicht nach ihrer Belastung. Stark beanspruchte Regionen werden damit nicht zerteilt, um die Belastung auf mehrere Server zu verteilen. Da jede Region und folglich jede Tabellen-Zeile und -Zelle aber nur von einem einzigen Server verwaltet wird, kann HBase Datenkonsistenz (Consistency) garantieren. Auch Transaction-Locks auf Zeilenebene sind daher möglich.

Beim Ausfall eines Region-Servers sind die verwalteten Regionen nicht mehr ansprechbar. Die Daten sind aber im hochverfügbaren HDFS gespeichert. Daher lassen sich die ausgefallenen Regionen einfach einem anderen Region-Server zuweisen. Damit ist HBase in Kombination mit dem HDFS hochverfügbar (Available). Die Verteilung der Regionen erfolgt allerdings über einen zentralen Master-Prozess, der einen SPOF darstellt.

HBase ist jedoch nicht tolerant gegenüber Netzwerk-Partitionen (Partition tolerance). Wird z. B. die Menge der Server in zwei Teile zerteilt, kann eine Teilmenge keine Regionen mehr bereitstellen. Nur die Server, die weiterhin mit dem Master kommunizieren können, sind weiterhin aktiv.

Die Kommunikation zwischen den HBase-Servern und den Clients erfolgt entweder über eine Java RMI-Schnittstelle, oder über eine Apache Thrift-API. Somit kann über verschiedene Programmiersprachen auf die HBase-Datenbank zugegriffen werden. Allerdings scheint die Thrift-Schnittstelle im Vergleich zur RMI-Schnittstelle wesentlich ineffizienter zu arbeiten {Ryan 2.0 Performance of HBase 2/5/2010 #111}.

4.2.3. Cassandra

Die Cassandra-Datenbank {Apache Software Foundation #67} wurde ursprünglich von Facebook {Willkommen bei Facebook #106} entwickelt, dann aber als Open-Source-Projekt veröffentlicht. Die Architektur ist eine Mischung aus Google Bigtable und der Amazon SimpleDB. Im Vergleich zu HBase werden die AP-Eigenschaften des CAP-Theorems umgesetzt. Allerdings lässt sich durch die Anpassung der Clients und der Replikationsmechanismen Konsistenz auf Kosten der Verfügbarkeit eintauschen {ArchitectureOverview #193}.

Das Datenmodell von Cassandra entspricht weitgehend dem zuvor beschriebenen HBase- und Bigtable-Datenmodell (vgl. S. 45). Ein zusätzliches Feature sind die Super Columns zur Gruppierung von Column families. Aufgrund der geringen Relevanz wird an dieser Stelle aber nicht näher auf die Super Columns eingegangen.

Cassandra ist eine vollständige Neuentwicklung und setzt z. B. kein bestehendes verteiltes Dateisystem voraus. Dabei wurde u. A. Wert auf eine möglichst einfache Administrierbarkeit gelegt. Der Betrieb von Cassandra erfordert daher lediglich einen einzelnen Prozess, der auf jedem Server ausgeführt wird.

Genau wie in HBase wird auch in Cassandra jede Tabellenzeile über einen eindeutigen, vom Client vorgegebenen Row-Key identifiziert. Die Verteilung der Daten über alle verfügbaren Knoten erfolgt mithilfe einer konsistenten Hashfunktion. Sie bildet den Row-Key auf einen endlichen und ringförmigen Namensraum ab. Wird ein Knoten das erste Mal gestartet, erhält er einen Token bzw. Platz innerhalb dieses Namensraums. Dieser Token zerteilt den Namensraum in zwei Teile. Falls mehrere Knoten existieren, wird auch der Namensraum öfters zerteilt (vgl. Abb. 15).

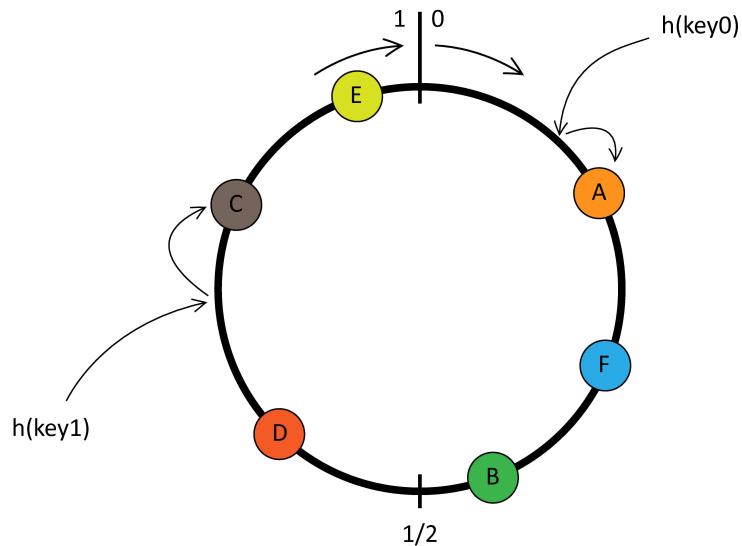


Abb. 15 Ringförmiger Namensraum, der durch mehrere Token zerteilt wird. Jeder Row-Key wird über eine konsistente Hashfunktion auf den Namensraum abgebildet.

Ein Knoten verwaltet nun den gesamten Namensraum vom Token des vorhergehenden Knotens bis zu seinem eigenen Token (im Uhrzeigersinn). Beim Speichern einer neuen Tabellenzeile wird zunächst die Hashfunktion auf den Row-Key angewendet. Damit wird der Row-Key auf einen Punkt im ringförmigen Namensraum abgebildet (vgl. Abb. 15). Der Algorithmus durchläuft nun alle bekannten Knoten im Uhrzeigersinn, bis er einen Knoten erreicht, dessen Token größer als das Ergebnis der Hashfunktion ist. Dieser Knoten ist nun für die Verwaltung der Tabellenzeile zuständig.

Beim Hinzufügen eines neuen Knotens wird diesem nicht einfach ein zufälliges Token zugewiesen. Stattdessen wird mithilfe der Lastdaten der meistgenutzte Teil des Namensraums ermittelt. Der neue Token wird nun so gewählt, dass er diesen Bereich zerteilt. Folglich reduzieren neue Knoten gezielt die Belastung der bestehenden Knoten. Die konsistente Hashfunktion garantiert außerdem, dass sich die Zuweisung zwischen Row-Key und Token auch beim Vergrößern des Namensraums und beim Hinzufügen weiterer Knoten nicht verändert. Damit bleiben bestehende Abbildungen zwischen Knoten und Tabellenzeilen bestehen.

Im Unterschied zu HBase basiert Cassandra auf keinem verteilten Dateisystem, sondern auf einem gewöhnlichen lokalen Dateisystem. Damit kann Cassandra nicht von

den Replikationsmechanismen eines verteilten Dateisystems profitieren. Um dennoch Ausfallsicherheit zu gewährleisten, repliziert jeder Knoten seine Daten auf einen oder mehrere andere Knoten. Die Anzahl der Replikate lässt sich über einen Replikationsfaktor festlegen. Cassandra implementiert verschiedene Algorithmen zur Bestimmung der Zielknoten für die Replikation. Im einfachsten Fall wird einfach der nächste Knoten im Namensraum verwendet (im Uhrzeigersinn). Komplexere Algorithmen berücksichtigen auch Server-Racks und Rechenzentren.

Cassandra verteilt alle Tabellenzeilen und damit alle Daten einer Tabelle gleichmäßig über die verfügbaren Knoten. Durch das Hinzufügen weiterer Knoten können damit die Datenbankgröße, die Speicher- und Lesezugriffe horizontal skalieren. Clients greifen über einen beliebigen Knoten auf den Datenbestand zu. Falls dieser den Aufruf nicht bearbeiten kann, leitet er ihn an einen passenden Knoten weiter. Dieser Ansatz vereinfacht den Client, führt aber zu einem höheren Netzwerkverkehr zwischen den Knoten.

Da Cassandra nur teilweise konsistent ist (eventually consistent), lassen sich Lesezugriffe durch jedes Datenreplikat bedienen. Folglich kann ein Lesezugriff auch auf veraltete Daten zugreifen. Dies ist allerdings nicht möglich, wenn Cassandra Konsistenz gewährleisten muss. Dann müssen die Lesezugriffe immer über einen zentralen Knoten ausgeführt werden, der sich damit zu einem Engpass entwickeln kann.

Durch die Hashfunktion wird die Tabelle auf Zeilenebene zerteilt. Dieser Ansatz ist allerdings für Table-Scans, die ein großes zusammenhängendes Tabellensegment auslesen, suboptimal. Für zufällige Lesezugriffe ist der Ansatz hingegen sehr gut geeignet.

Die Kommunikation zwischen Cassandra und den Clients erfolgt über eine Thrift-Schnittstelle. Somit lässt sich die Cassandra-Datenbank weitgehend unabhängig zur Programmiersprache nutzen.

4.2.4. Vergleich

MongoDB, HBase und Cassandra sind für den Einsatz in horizontal skalierbaren Umgebungen äußerst interessante Ansätze. MongoDB bietet den größten Funktionsumfang, was das Abfragen von Daten betrifft. Allerdings befindet sich der, für horizontal skalierbare Systeme unbedingt notwendige Auto-Sharding-Mechanismus noch in einer Alpha-Phase. Auch die Replikationsmechanismen stehen denen von Cassandra und dem HDFS nach. Darüber hinaus lässt sich MongoDB lediglich über eine proprietäre Schnittstelle ansprechen. Daher wird MongoDB in TwoSpot nicht eingesetzt.

HBase und Cassandra sind beides spaltenorientierte Datenbanken und bieten aus Perspektive der API einen sehr ähnlichen, aber im Vergleich zu MongoDB geringen Funktionsumfang. Aufgrund der unterschiedlichen Zielsetzungen bezüglich des CAP-Theorems und der Architektur weisen beide Systeme bestimmte Vor- und Nachteile auf.

Bezüglich des Speicher-Backends unterscheiden sich die beiden Datenbanken signifikant. HBase setzt zur ausfallsicheren und dauerhaften Datenspeicherung auf das verteilte Dateisystem HDFS. Cassandra verwendet hingegen das gewöhnliche Dateisystem und implementiert selbst ausgefeilte Mechanismen zur Replikation des Datenbestandes. Beide Replikationsmechanismen sind für den Betrieb innerhalb eines Rechenzentrums geeignet und berücksichtigen Server-Racks als physikalische Netzwerkeinheit (Rack-Aware). Cassandra ist darüber hinaus für den rechenzentren-überspannenden Betrieb ausgelegt und unterstützt somit die Replikation zwischen Rechenzentren. Allerdings wird dabei eine schnelle Netzwerkverbindung vorausgesetzt {Lakshman #194: 3}. Für HBase ist eine rechenzentren-überspannende Replikation erst für die Version 0.21.0 geplant {[#HBASE-1295] Multi data center replication #135}. Der vorgestellte Algorithmus wählt einen anderen Ansatz als Cassandra und eignet sich auch für vergleichsweise langsame Netzwerkverbindungen {[#HBASE-1295] Multi data center replication #135}.

Rechenzentren können z. B. aufgrund von Naturkatastrophen, Stromausfällen, Kühlproblemen oder Infrastruktur-Störungen ausfallen. Bezüglich TwoSpot besteht allerdings kein dringender Bedarf für eine rechenzentren-überspannende Replikation. Falls zukünftig eine solche Replikation notwendig wird, sind die Rechenzentren wahrscheinlich über die Infrastruktur des öffentlichen Internets verbunden. Damit stellt die HBase-Replikation die geeignetere Lösung dar.

Cassandra enthält im Gegensatz zum HBase-Master keinen SPOF. Aufgrund der erhöhten Komplexität eines solchen Systems erhöht sich aber auch die Wahrscheinlichkeit von Fehlverhalten in bestimmten Randsituationen. Es besteht durchaus auch die Möglichkeit, dass ein Fehler das gesamte System zum Erliegen bringt. Somit sind die beiden Systeme bezüglich der SPOF-Eigenschaft gleichwertig.

Für den Zugriff auf den HBase-Datenbestand existiert bereits ein JDO-Plugin, für Cassandra hingegen nicht. Dieser Aspekt ist besonders für TwoSpot von Interesse, da der Vendor Lock-in durch offene Standards reduziert werden soll (vgl. S. 37).

Cassandra verteilt die Daten zeilenweise auf die verfügbaren Knoten. HBase zerteilt die Tabellen hingegen in größere Segmente. Damit lässt sich die Verteilung der Zeilen gezielter über den Row-Key steuern. Der Ansatz von Cassandra eignet sich am besten zur Skalierung zufälliger Lesezugriffe unter bedingter Datenkonsistenz. Für Table-Scans, die ein großes zusammenhängendes Segment auslesen, ist HBase hingegen besser geeignet. TwoSpot erfordert eine hohe Datenkonsistenz, womit sich die Vorteile von Cassandra nicht nutzen lassen. Außerdem werden Table-Scans in verschiedenen Anwendungsszenarien benötigt (vgl. S. 71, Datenspeicher). Daher ist die Datenorganisation von HBase besser für TwoSpot geeignet.

HBase ist für die Einbindung in Map-Reduce-Frameworks optimiert. Die Architektur von Cassandra wurde hingegen nicht in Hinblick auf den Einsatz in einer Map-Reduce-Umgebung entwickelt. Die mittelfristige Planung von TwoSpot sieht die Verarbeitung der Monitoring-Daten über Map-Reduce vor. Allerdings dürften beide Systeme für diesen Anwendungsfall ausreichend sein.

Cassandra benötigt pro Server und Knoten nur einen Prozess. HBase benötigt hingegen einen zentralen Master-Prozess und einen Storage-Prozess auf jedem Speicher-knoten. Zusätzlich muss das HDFS berücksichtigt werden. Dieses benötigt ebenfalls pro Speicherknoten einen Speicher-Prozess und mehrere zentrale Master-Prozesse. Somit ist der gesamte Administrationsaufwand von HBase vergleichsweise hoch.

Abschließend findet die HBase-Datenbank und nicht Cassandra in TwoSpot Verwen-dung. Hauptgründe sind die höhere Stabilität, die Datenorganisation, das existierende JDO-Plugin und die vergleichsweise einfache Architektur. Darüber hinaus benötigt TwoSpot auch ein verteiltes Dateisystem, womit sich der Einsatz des HDFS in Kombi-nation mit HBase anbietet. Da in jedem Fall ein weiterer Prozess für ein verteiltes Da-teisystem benötigt wird, kommt der Ein-Prozess-Vorteil von Cassandra nicht zum Tra-gen.

4.3. Verteilte Dateisysteme

Die zuvor beschriebenen Datenbanken sind nicht zur Speicherung großer Datenmen-gen wie z. B. Audio-, Video- oder Bilddaten konzipiert. Aus diesem Grund wird für die TwoSpot-Plattform ein weiteres System zur Speicherung großer Datenmengen benö-tigt. Dieses muss zwei Anforderungen erfüllen: Verfügbarkeit, auch beim Ausfall einzel-ner Hardware-Komponenten und die Speicherung sehr großer Datenmengen im mehr-fachen Terabyte-Bereich.

4.3.1. Hadoop Distributed File System

Das Hadoop Distributed File System (HDFS) {Apache Software Foundation #81} wird im Rahmen des Apache Hadoop-Projekts entwickelt und orientiert sich am Google File System (GFS) {Ghemawat #110}. Bei seiner Entwicklung wurden folgende Annahmen getroffen {Ghemawat 2003 #110: 30}:

- Aufgrund der hohen Anzahl an Servern und dem Einsatz billiger Server-Hardware treten Hardware-Fehler häufig auf und müssen vom Dateisystem be-rücksichtigt werden.
- Im Dateisystem werden vergleichsweise wenige, aber sehr große Dateien mit einer Größe ab 100 MB gespeichert. Es muss daher nicht für die Verwaltung vie-ler kleiner Dateien optimiert oder ausgelegt sein.
- Einmal geschriebene Dateien werden sehr oft gelesen aber niemals verändert. Viel öfter werden neue Daten an bestehende Dateien angehängt.

- An eine Datei hängen sehr viele Clients parallel neue Daten an.
- Ein hoher Lesedurchsatz ist wichtiger als eine geringe Latenz.

Die Architektur des GFS spiegelt sich im HDFS wider und setzt sich aus einem einzelnen NameNode und vielen Chunk-Servern zusammen (vgl. Abb. 16). Die Dateien werden in Blöcke mit einer festen Größe zerteilt. Ein Block wird über einen eindeutigen Block-Handle identifiziert. Die Chunk-Server speichern die Blöcke im normalen Dateisystem ab. Zur Gewährleistung von Verfügbarkeit wird jeder Block auf mehrere Chunk-Server repliziert. Dabei werden Server-Racks als physikalische Netzwerkgrenze berücksichtigt.

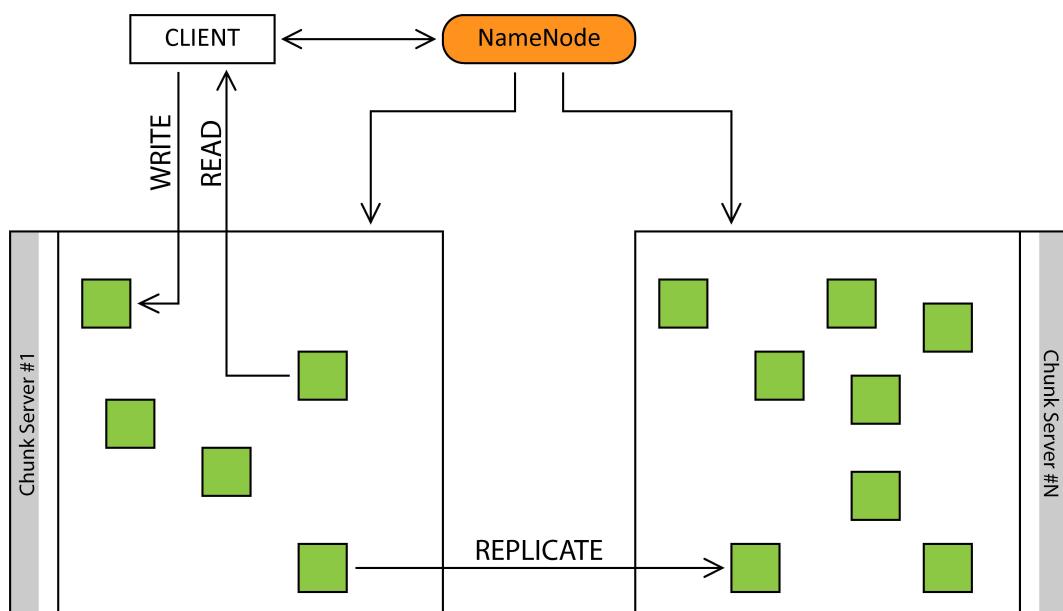


Abb. 16 Aufbau des HDFS-Dateisystems mit zwei Chunk-Servern

Der NameNode ist zur Verwaltung der Metadaten zuständig. Das sind die Datei- und die Verzeichnisstruktur. Alle Operationen bezüglich der Metadaten müssen über den NameNode innerhalb einer Transaktion ausgeführt werden. Zu einer Datei speichert er die Block-Handles, aus denen sie sich zusammensetzt. Zu den Block-Handles werden alle Adressen der Chunk-Server gespeichert, die ein Replikat des Blocks enthalten. Da lediglich ein NameNode existiert, stellt er einen SPOF dar.

Die Clients kommunizieren mit dem NameNode nur beim Zugriff auf die Metadaten. Alle datenintensiven Operationen wickeln die Clients direkt über die Chunk-Server ab. Bei einer Lese-Operation ermittelt der Client z. B. zunächst die Adresse der Chunk-Server, die den entsprechenden Block speichern. Diese Information erhält er über den Master. Anschließend wählt er einen Chunk-Server aus und liest über ihn die Daten aus.

Möchte der Client eine neue Datei erstellen oder Daten an eine bestehende Datei anhängen, teilt er dies zunächst dem Master mit. Dieser ermittelt alle Chunk-Server, die ein Replikat des entsprechenden Blocks enthalten. Ein Replikat wird daraufhin als Primär-Replikat bestimmt. Alle anderen werden als Sekundär-Replikate bezeichnet. Die Daten müssen nun an alle Chunk-Server mit einem entsprechenden Replikat übertragen werden. Dazu sendet sie der Client an den nächstgelegenen Chunk-Server, ohne Berücksichtigung seiner Rolle. Dieser leitet die Daten wiederum an den nächstgelegenen Chunk-Server weiter. Der Vorgang wird fortgesetzt, bis alle Chunk-Server die neuen Daten erhalten und ihren Empfang bestätigt haben (vgl. Abb. 17). Nun teilt der Client dem Chunk-Server mit dem Primär-Replikat mit, dass die Daten erfolgreich übermittelt wurden. Dieser ruft dann alle Sekundär-Replikate auf und weist sie an, die Daten dauerhaft zu speichern. Er selbst speichert die Daten ebenfalls. Der Speicherungsvorgang ist erfolgreich, wenn alle Chunk-Server die Daten erfolgreich speichern konnten.

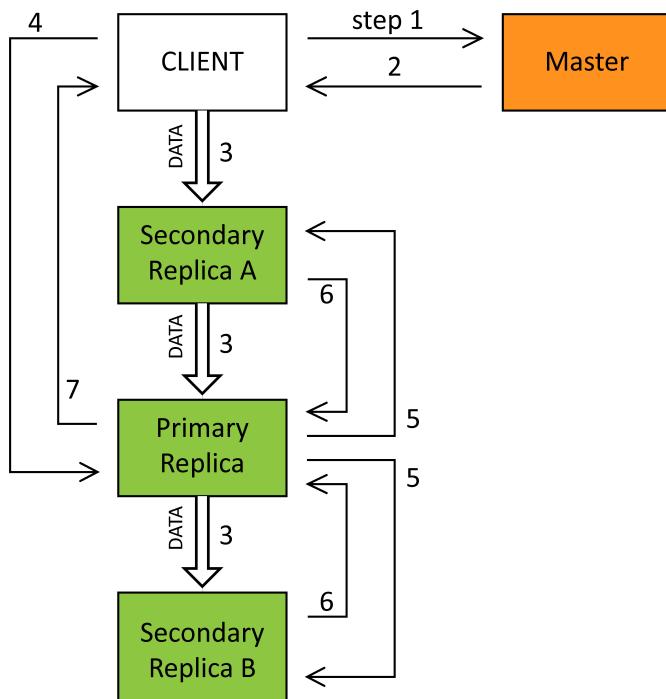


Abb. 17 Ablauf beim Schreiben einer Datei in das HDFS

Damit mehrere Clients gleichzeitig Daten an eine Datei anhängen können, werden die Schreibzugriffe vom Primär-Replikat sequenziell nummeriert. Die Daten werden auf allen Replikaten in dieser Reihenfolge geschrieben. HDFS unterstützt zum aktuellen Zeitpunkt keine Dateiänderungen, lediglich das Anhängen an bestehende Dateien und das Anlegen neuer Dateien ist möglich.

Aufgrund verschiedener Fehlerszenarien enthält das GFS eine Reihe weiterer Mechanismen zur Gewährleistung von Datenkonsistenz, Lastverteilung und Parallelität. Diese Mechanismen finden sich auch im HDFS wieder. Eine detaillierte Beschreibung würde

allerdings den Rahmen der Arbeit sprengen, weshalb an dieser Stelle auf das Google File System-Paper verwiesen wird {Ghemawat #110}.

Zusammengefasst ist das HDFS zur ausfallsicheren Speicherung sehr großer Datens Mengen konzipiert. Über die Replikation von Dateiblöcken wird darüber hinaus eine horizontale Skalierung der Lesezugriffe erreicht. Schreibzugriffe lassen sich ebenfalls über alle Speicherknoten verteilen und damit skalieren. Aufgrund der Replikation ist ihre Ausführung im Vergleich zu Lesezugriffen aber aufwendiger.

4.3.2. Cloudstore

Cloudstore {WELCOME TO CLOUDSTORE 5/21/2009 #191} ist eine Weiterentwicklung des Kosmos File Systems (KFS) und ist ebenfalls nach dem Konzept des GFS implementiert. Es ist somit auch sehr ähnlich zum HDFS und daher auch zu HBase und Hypertable kompatibel. Die Architektur ist ähnlich zu der von HBase und Bigtable und besteht aus einem NameNode zur Verwaltung der Metadaten und mehreren Chunk-Servern zum Speichern der Daten. Im Unterschied zum HDFS ist Cloudstore allerdings in C++ implementiert. Für die Clients existieren aber auch Java- und Python-Bibliotheken.

An dieser Stelle wird der Funktionsumfang von Cloudstore nicht näher beschrieben, da er eine große Ähnlichkeit zum HDFS aufweist. Ebenso ähneln sich die Vor- und Nachteile von Cloudstore.

4.3.3. MogileFS

MogileFS wird von DANGA Interactive {Danga Interactive #77} entwickelt und in mehreren großen Projekten erfolgreich eingesetzt (Last.fm {Last.fm #98}, Digg {How Digg works | Digg #113}, Wikispaces {Wikispaces #114}).

Im Gegensatz zum HDFS wurde MogileFS nicht zur Verwaltung einiger großer Dateien, sondern vieler kleiner Dateien konzipiert. Weitere Zielsetzungen sind: Skalierbarkeit, Fehlertoleranz, kein SPOF und eine hohe Verfügbarkeit durch Replikation. Die Architektur setzt sich aus mehreren Komponenten zusammen.

Auf jedem Storage-Knoten wird ein einfacher HTTP- bzw. DAV²²-Server ausgeführt. Für alle Speicher-, Lösch- und Lese-Operationen kommt daher das HTTP-Protokoll zum Einsatz. Über das DAV-Protokoll werden hingegen Performance- und Last-Daten abgefragt. Die Dateiverwaltung übernehmen Tracker-Knoten, die den mogilefsd-Prozess und eine Reihe von Kind-Prozessen ausführen. Die Clients kommunizieren immer direkt mit dem mogilefsd-Prozess. Dieser leitet die Aufrufe dann an einen entsprechenden Kind-Prozess weiter.

²² Distributed Authoring and Versioning

Der Kind-Prozess Replication ist für die Replikation neuer und veränderter Dateien zuständig. Sobald ein Client eine entsprechende Operation ausführt, wird der Replication-Prozess auf dem Tracker informiert. Er übernimmt anschließend die Replikation der entsprechenden Dateien. Beim Ausfall eines Storage-Knotens müssen die verlorenen Replikate nochmals repliziert werden, um den vorgegebenen Replikationsfaktor aufrecht zu erhalten. Zum Erkennen von Ausfällen überwacht der Reaper-Prozess alle Storage-Knoten und informiert den Replication-Prozess bei Ausfällen.

Beim Löschen einer Datei müssen auch ihre Replikate gelöscht werden. Somit lassen sich Lösch-Operationen nicht direkt vom Client ausführen. Daher erstellt der Client einen Löschen-Job und weist ihn einem Deletion-Prozess zu. Dieser verarbeitet alle Jobs in chronologischer Reihenfolge.

Die Verwaltung der Metadaten wird vom mogilefsd-Prozess und einer RDB übernommen. Die Datenbank speichert den Dateinamen und die Adressen aller Storage-Knoten, die Replikate der Datei enthalten. Dateien lassen sich damit nur in einem flachen Namensraum organisieren. Verzeichnisse werden nicht direkt unterstützt, lassen sich aber über den Dateinamen simulieren.

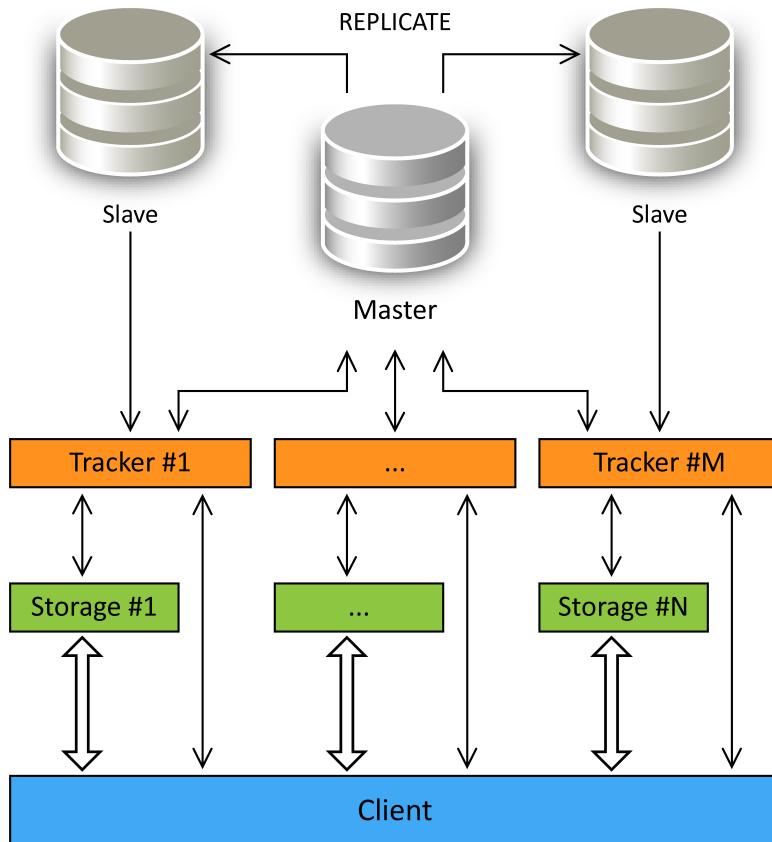


Abb. 18 Ausfallsichere Architektur des MogileFS-Dateisystems

Der Client kommuniziert direkt mit den Tracker-Knoten. Allerdings werden dabei nur Metadaten ausgetauscht. Für datenintensive Operationen kommunizieren die Clients

direkt mit den Storage-Knoten (vgl. Abb. 18). Beim Speichern einer Datei werden z. B. folgende Schritte durchlaufen.

- 1) Der Client stellt eine *create_open*-Request an einen Tracker. Dieser gibt eine Liste mit Adressen von Storage-Knoten zurück.
- 2) Der Client wählt einen Storage-Knoten aus und sendet ihm die Datei zu.
- 3) Nach einem erfolgreichen Speichervorgang ruft der Client einen Tracker mit einer *create_close*-Request auf und teilt ihm den Dateinamen und die Adresse des Storage-Knotens mit.
- 4) Der Tracker erstellt daraufhin einen Job im Replication-Prozess, der die Datei auf mehrere Storage-Knoten repliziert.

Lesezugriffe laufen nach einem ähnlichen Muster ab. Dabei erhält der Client vom Tracker eine Liste mit Storage-Knoten, die Replikate der entsprechenden Datei enthalten. Die Liste ist nach der Auslastung sortiert, d. h. Storage-Knoten mit einer geringen Belastung befinden sich am Anfang. Der Client wählt dann einen Storage-Knoten aus und liest von diesem die Datei.

Zusammengefasst weist MogileFS eine sehr einfache Architektur auf. Die gesetzten Ziele werden vollständig erreicht. Allerdings stellt MogileFS eher einen skalierbaren Dienst zur ausfallsicheren Speicherung von Daten, als ein Dateisystem dar. Da die Metadaten in einer relationalen Datenbank verwaltet werden, ergeben sich eventuell Einschränkungen bei einer hohen Schreibbelastung. Darüber hinaus ist MogileFS aufgrund der dateibasierten Replikation nur für vergleichsweise kleine Dateien geeignet.

4.3.4. Vergleich

TwoSpot benötigt ein verteiltes Dateisystem zur Speicherung von Dateien mit einer Größe von wenigen Kilobyte bis zu ca. 100 MB. MogileFS wurde speziell für diesen Anwendungsfall konzipiert. Allerdings erscheint die dateibasierte Replikation in Kombination mit sehr kleinen Dateien impraktikabel. Wenn das Dateisystem z. B. 1 TB an Dateien mit einer Größe von 15 KB speichern muss (ca. 71.582.788 Dateien), ergibt sich ein hoher Replikationsaufwand. Bei einem typischen Replikationsfaktor von 3 müssen 214.748.364 Dateien verwaltet werden. Da jede Datei in einem eigenen Kopiervorgang repliziert wird, entsteht ein hoher Protokoll-Overhead. Darüber hinaus berücksichtigt der Replikationsmechanismus keine Server-Racks als physikalische Netzwerkgrenze. Ein weiterer Nachteil besteht in den fehlenden Mechanismen zur Gewährleistung von Datenintegrität. MogileFS bietet z. B. keine Prüfsummen, um Datenfehler zu erkennen.

Cloudstore und HDFS sind hingegen auf die Speicherung großer Dateien ausgelegt. Die Replikationsmechanismen arbeiten auf Basis größerer Dateiblöcke, die sich sehr effizient replizieren lassen. Darüber hinaus berücksichtigt HDFS Server-Racks als phy-

sikalische Netzwerkgrenze. Um die Datenintegrität zu gewährleisten, können die HDFS-Clients die Prüfsumme der Dateiblöcke berücksichtigen. Ein Nachteil bezüglich TwoSpot ist, dass eine Zwischenschicht benötigt wird, die kleine Dateien innerhalb großer HDFS-Dateien verwaltet.

Bezüglich TwoSpot weist das HDFS zwei Vorteile auf: 1) Die Replikationsmechanismen sind im Vergleich zu MogileFS wesentlich effizienter; 2) Die HBase-Datenbank basiert bereits auf dem HDFS. Da MogileFS darüber hinaus Schwächen bei der Verwaltung sehr kleiner Dateien aufweist, wird das HDFS-Dateisystem für TwoSpot eingesetzt.

5. TwoSpot

Das Fundament der TwoSpot-Plattform bilden vier Komponenten:

- 1) Der AppServer stellt das Execution Environment dar.
- 2) Der Controller verwaltet alle AppServer-Prozesse auf einem Server.
- 3) Das Frontend agiert als Reverse-Proxy zwischen dem Browser und den Controllern und damit AppServern.
- 4) Der Master sammelt Management-Daten und übernimmt einen Teil des Load-Balancings.

Zusätzlich werden ein zentraler Synchronisations- und Konfigurationsdienst, ein skalierbares Dateisystem und eine skalierbare Datenbank benötigt. Die Abhängigkeiten zwischen den Komponenten sind in Abb. 19 dargestellt.

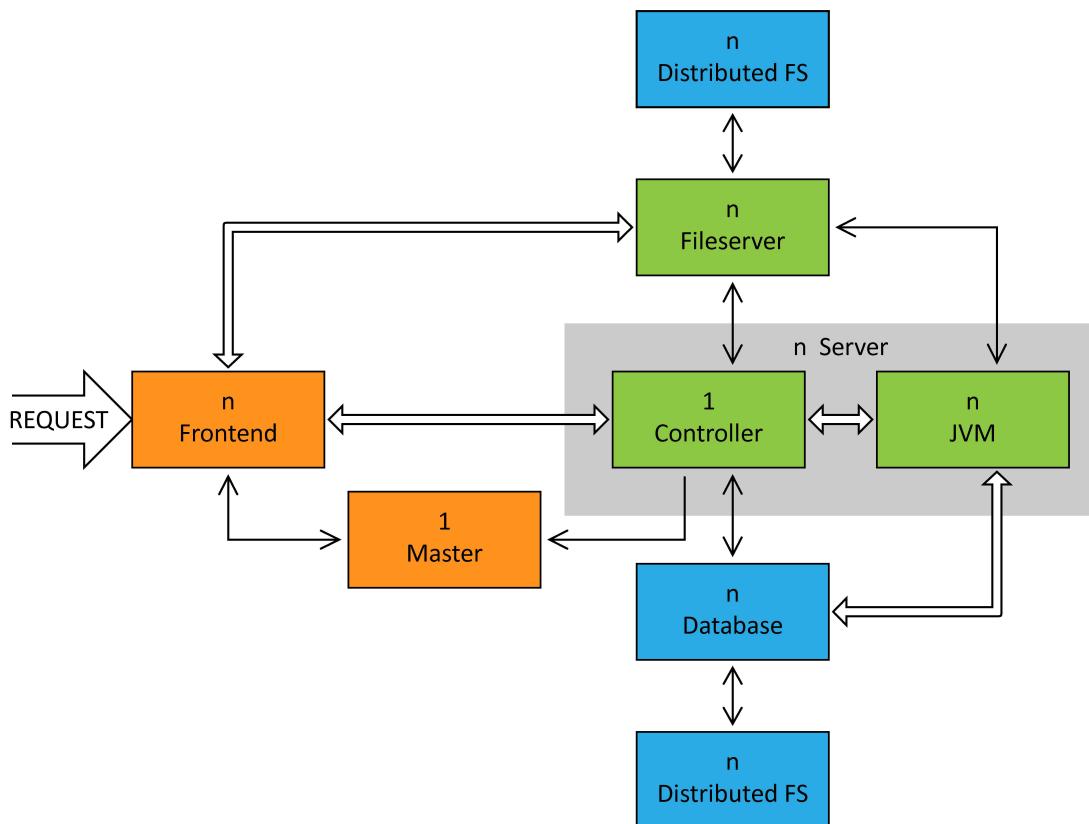


Abb. 19 Kommunikation zwischen den Komponenten der TwoSpot-Plattform

Jeder TwoSpot-Anwendung wird eine eindeutige ApplId zur Identifikation zugewiesen. Sie setzt sich aus Buchstaben und Ziffern zusammen und ist zwischen 5 und max. 20 Zeichen lang. Die ApplId wird bei der Kommunikation mit einer TwoSpot-Anwendung in der URL codiert. Dafür kommt der DNS Wildcard-Mechanismus zum Einsatz, womit die ApplId vor dem Domain-Namen platziert wird. Lautet die URL der TwoSpot-Plattform beispielsweise: <http://twospot.com>, dann ist die Anwendung „portal“ über die URL <http://portal.twospot.com> erreichbar.

5.1. ZooKeeper

Der ZooKeeper-Dienst {Apache Software Foundation #61} ist ein zentraler Synchronisations- und Konfigurationsdienst, der ursprünglich für die Apache HBase-Datenbank entwickelt wurde. Er verwaltet eine zentrale Verzeichnisstruktur aus Verzeichnissen und Datenknoten. Jeder Client kann diese Verzeichnisstruktur transaktionsgesteuert verändern.

Ein Verzeichnis- oder Datenknoten kann entweder als transienter oder persistenter Knoten angelegt werden. Persistente Knoten bleiben bis zu ihrer expliziten Löschung in der Verzeichnisstruktur gespeichert. Die Lebensdauer transienter Knoten ist hingegen an die Client-Sitzung gekoppelt. Ein Client baut beim Starten eine Sitzung mit dem ZooKeeper auf und schließt sie erst wieder, wenn er beendet wird. Beim Beenden der Sitzung werden auch alle, an die Sitzung gebundenen, transienten Knoten aus der Verzeichnisstruktur entfernt. Dies gilt auch im Fall eines Client-Absturzes, wenn die Sitzung nicht ordnungsgemäß beendet wird. Über die Datenknoten lassen sich kleine Datenmengen wie z. B. Konfigurationseinstellungen zentral im ZooKeeper speichern.

Die Knoten ermöglichen aber auch die Synchronisation zwischen mehreren Clients auf verschiedenen Servern. Dazu kann ein Client für jeden beliebigen Knoten einen Watcher installieren. Das ist ein Callback-Mechanismus, der bei allen Änderungen des Knotens aufgerufen wird. Änderungen sind z. B., wenn sich der Inhalt eines Datenknotens verändert, der Knoten gelöscht oder ein Kind-Knoten hinzugefügt oder entfernt wird. Ein Watcher kann somit z. B. zur Synchronisation von Producer-Consumer-Queues eingesetzt werden. Ein weiterer Anwendungsfall besteht im Warten, bis eine bestimmte Komponente gestartet wurde und einen transienten Knoten in der Verzeichnisstruktur angelegt hat.

5.2. AppServer

Der AppServer stellt das Execution Environment für TwoSpot-Anwendungen dar. Seine Implementierung basiert auf einer modifizierten Variante des Jetty-Servers (vgl. S. 38, Anwendungsserver).

Der AppServer-Prozess wird vom Controller gestartet. Dabei erhält er drei Kommandozeilen-Argumente: 1) Die AppId der auszuführenden Anwendung; 2) Netzwerk-Port unter dem der AppServer auf eingehende HTTP-Requests wartet; 3) Ein mehrstelliger Sicherheitstoken zur Authentifikation gegenüber dem Controller.

Im ersten Schritt lädt der Bootstrapping-Prozess das Anwendungs-Archiv vom zentralen File-Server herunter. Der Aufbau des Archivs ist fast identisch zu WAR²³-Archiven,

²³ Web Application Archive

enthält aber zusätzlich noch eine YAML²⁴-Datei {The Official YAML Web Site 10/1/2009 #148}, die als Anwendungs-Deskriptor bezeichnet wird. Dieser enthält eine Reihe von Eigenschaften zur Konfiguration des AppServers. Darüber hinaus definiert er die Programmiersprache der Anwendung und legt damit den Software-Stack fest.

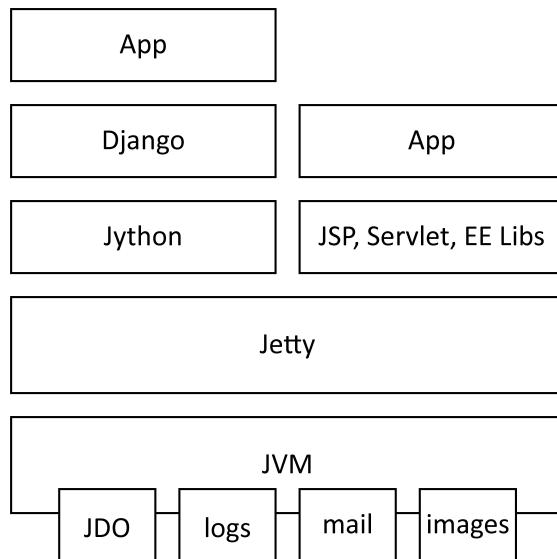


Abb. 20 Der AppServer Software-Stack unterstützt verschiedene Programmiersprachen auf Basis der Java VM

Der AppServer Software-Stack (vgl. Abb. 20) ist für den Betrieb verschiedener Programmiersprachen ausgelegt. Voraussetzung ist allerdings, dass sich die Programmiersprache auf Basis der Java VM betreiben lässt. Dies sind entweder Sprachen, deren Interpreter direkt in Java implementiert ist oder deren Compiler Java class-Dateien erstellt, die sich direkt von der Java VM ausführen lassen. Prominente Beispiele sind z. B. JavaScript (Rhino; {Happy Cog Studios - <http://www.happycog.com> #124}), Python (Jython; {Python Software Foundation #63}) und Ruby (JRuby; {JRuby.org 2/23/2010 #128}).

Die Dateistruktur des WAR-Archivs mit dem Anwendungs-Deskriptor wird in jedem Fall, unabhängig von der Programmiersprache eingesetzt. Nachdem der AppServer den allgemeinen Teil vom Anwendungs-Deskriptor verarbeitet hat, startet er den entsprechenden Software-Stack. Da jeder Software-Stack den Anwendungs-Deskriptor erneut verarbeitet, kann dieser zusätzlich zu den allgemeinen Eigenschaften, sprachabhängige Konfigurationseinstellungen enthalten.

Die aktuelle Realisierung des Software-Stacks für Java und Python basiert auf dem Jetty-Server. Dieser lässt sich aber durch jeden beliebigen Anwendungsserver wie z. B.

²⁴ Yet Another Markup Language

Tomcat oder Resin ersetzen. Allerdings erfordert dies eine alternative Implementierung der entsprechenden Schritte im Bootstrapping-Prozess.

Der Java Software-Stack basiert vollständig auf dem Servlet Context des Jetty-Servers und unterstützt somit den Betrieb gewöhnlicher Java Web-Anwendungen. Zusätzlich zu Java wurde mithilfe der Jython-Bibliothek ein Software-Stack für Python implementiert. Dabei werden die Python-Quellcodes in class-Dateien übersetzt, womit eine hohe Ausführungsgeschwindigkeit erreicht wird. Um bestehende Python Web-Frameworks wie etwa Django {Django | The Web framework #150} zu betreiben, wurde eine WSGI²⁵-Schnittstelle {PEP 333 -- Python Web 3/5/2010 #149} auf Basis von Jetty implementiert. Der Python-Stack erwartet im Anwendungs-Deskriptor eine zusätzliche Konfigurationseinstellung, die auf ein Python-Skript innerhalb der Anwendung verweist. Dieses Skript wird für jede eingehende Request ausgeführt, womit es den Einstiegs-punkt in die Python Web-Anwendung bildet. Normalerweise greift es auf die zuvor erwähnte WSGI-Schnittstelle zurück und bindet damit ein Web-Framework ein. Um einen multithreaded Betrieb zu ermöglichen, werden alle Requests innerhalb einer eigenen Instanz des Jython-Interpreters ausgeführt.

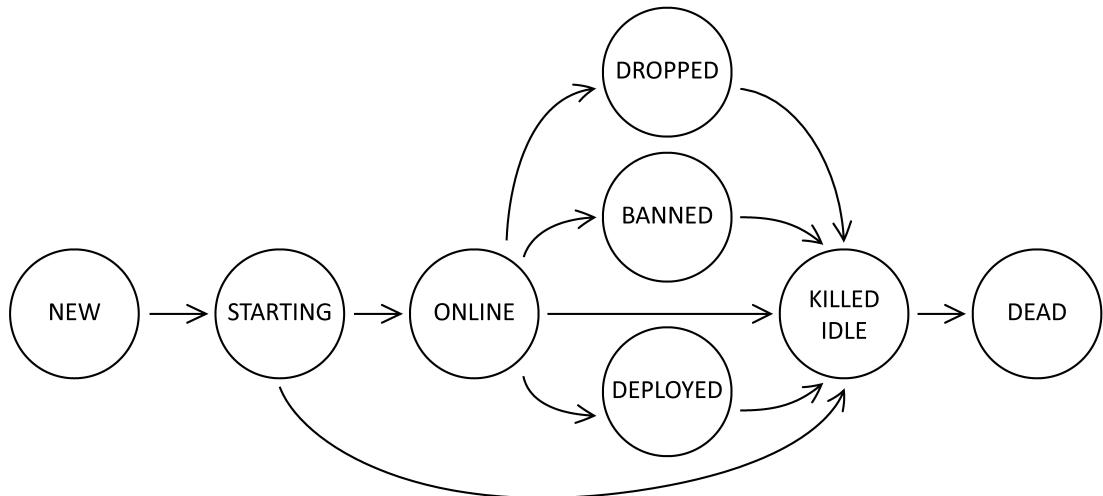
Alle Komponenten der TwoSpot-Plattform sind konsequent auf eine horizontale Skalierung ausgelegt. Aus diesem Grund startet die Plattform anstelle weniger leistungsstarker viele kleine AppServer. Damit verteilt sich die Anwendungs-Belastung auf mehrere verschiedene Server. Einzelne Hotspots wirken sich somit nicht so stark auf die Gesamtperformance einer Anwendung aus. Auch bei Serverausfällen sind kleinere App-Server von Vorteil. Sie verarbeiten vergleichsweise wenige Requests, womit sich auch die Auswirkungen eines Absturzes reduzieren.

In der Regel werden mehrere AppServer mit verschiedenen Anwendungen auf einem Server und unter demselben Betriebssystem ausgeführt. Daher ist der AppServer für die Isolation der Anwendung verantwortlich. Zu diesem Zweck implementiert er eine Reihe von Sicherheitsmechanismen, die im Abschnitt: Sicherheitsaspekte genauer ausgeführt werden.

5.3. Controller

Auf jedem Server wird genau ein Controller ausgeführt. Seine Kernaufgabe ist die Verwaltung aller AppServer-Prozesse und die Sammlung von Management-Daten. Dazu implementiert er, ähnlich zum Frontend, einen HTTP-Proxy und leitet alle eingehenden HTTP-Requests an einen entsprechenden AppServer weiter. Zu jeder eingehenden Request analysiert er die URL und ermittelt somit die AppId zur Identifikation der Anwendung (vgl. S. 59). Falls für die entsprechende Anwendung noch kein AppServer ausgeführt wird, startet der Controller einen neuen AppServer.

²⁵ Web Server Gateway Interface

**Abb. 21 Lifecycle zur Verwaltung eines AppServers**

Jeder AppServer-Prozess wird vom Controller über einen Lifecycle verwaltet (vgl. Abb. 21). Der Lifecycle beginnt mit dem Zustand NEW und geht nach dem Start des AppServer-Prozesses in den Zustand STARTING über. Nun wartet der Controller auf ein Signal vom AppServer. Zur Kommunikation werden die Standard Input- und Output-Streams verwendet. Sobald der AppServer ONLINE meldet, wird auch der Lifecycle-Zustand auf ONLINE gesetzt.

Da der Startvorgang eines AppServers mehrere Sekunden in Anspruch nimmt, lässt sich die startauslösende Request nicht direkt zustellen. Stattdessen muss diese warten, bis der AppServer in den Zustand ONLINE übergeht. Um ein aktives Blockieren des Request-Handlers und damit des Threads zu vermeiden, kommt der Continuation-Mechanismus von Jetty zum Einsatz. Falls während des Startvorgangs weitere Requests eintreffen, werden diese ebenfalls mithilfe des Continuation-Mechanismus pausiert. Sobald der Lifecycle in den Zustand ONLINE übergeht, reaktiviert er alle Continuations und führt damit die pausierten Requests erneut aus.

Zum Starten und Beenden der AppServer wird eine Job-Queue eingesetzt. Die Abarbeitung der Queue erfolgt in einem eigenen Thread nach dem Producer-Consumer-Prinzip. Dieser Mechanismus in Kombination mit dem zuvor beschriebenen Continuation-Mechanismus stellt sicher, dass die request-verarbeitenden Threads während dem Start eines AppServers nicht blockieren. Falls beim Starten eines AppServers ein Fehler auftritt oder der Prozess terminiert, wird der Lifecycle in den Zustand KILLED überführt. Bei dieser Zustands-Transition wird automatisch ein Kill-Job in der Job-Queue registriert. Dieser beendet den Prozess und führt alle Aufräumarbeiten im Controller durch. Dabei werden auch die Continuations der pausierten Requests terminiert. Anschließend aktualisiert er den Lifecycle-Zustand auf DEAD. Der Controller entfernt regelmäßig alle Lifecycle im Zustand DEAD.

Während eine Anwendung ausgeführt wird, kann eine neue Anwendungsversion deployt werden. Im Anschluss an einen erfolgreichen Deployment-Vorgang wird der Life-cycle aller entsprechenden AppServer auf DEPLOYED gesetzt. In diesem Zustand leitet der Controller keine neuen Requests an den AppServer weiter. Bereits vermittelte Requests werden aber noch vollständig bearbeitet. Anschließend wird der AppServer auf den Zustand KILLED gesetzt, womit ihn die Job-Queue automatisch beendet. Der Controller kann keine neuen Requests an einen AppServer im Zustand DEPLOYED zustellen. Daher startet er einen neuen AppServer, der automatisch mit der neuen Anwendungsversion betrieben wird. Somit werden kurzzeitig zwei AppServer mit derselben Anwendung aber unterschiedlichen Versionen ausgeführt. Dieser Mechanismus erlaubt ein Anwendungs-Deployment unter Last und ohne Ausfallzeit.

Da TwoSpot die horizontale Skalierbarkeit umsetzt, kann eine Anwendung in mehreren AppServern gestartet werden. Die Anzahl der benötigten AppServer wird automatisch mithilfe verschiedener Skalierungs-Mechanismen geregelt (vgl. S. 78, Horizontale Skalierung). Um überschüssige AppServer zu beenden, enthält der Lifecycle zwei weitere Zustände: DROPPED und BANNED. In beiden Zuständen nimmt der Controller keine neuen Requests für die entsprechende Anwendung an und weist sie mit dem HTTP Fehler-Code 803 ab. Dabei handelt es sich um einen plattformspezifischen Fehler-Code, der vom Frontend erkannt und nicht an den Browser weitergereicht wird. Alle bereits angenommenen Requests leitet der Controller allerdings ordnungsgemäß an den entsprechenden AppServer weiter und wartet, bis sie verarbeitet wurden. Abschließend wird der AppServer über die Job-Queue beendet. Auch danach werden eingehende Requests für eine bestimmte Zeit, die abhängig vom Zustand DROPPED oder BANNED ist, mit dem HTTP Fehler-Code 803 abgewiesen.

Eine weitere Aufgabe des Controllers besteht in der Erfassung von Management-Daten. Der erste Ansatz basierte auf dem OperatingSystem JMX-Bean, um grundlegende Daten bezüglich der CPU-, Speicherauslastung und Anzahl der CPU-Kerne zu ermitteln. Allerdings konnte die CPU-Auslastung nur unter Linux-Systemen ausgelesen werden. Beim Deployment in einer IaaS-Plattform hat sich zudem gezeigt, dass der JMX-Bean unter Multi-Core-Systemen keine ausreichenden Daten liefert. Aus diesem Grund wird statt dem JMX-Bean die Sigar-Bibliothek {SIGAR API System Information Gatherer 2/27/2010 #129} eingesetzt. Mit dieser ermittelt der Controller nun die Daten: CPU-Auslastung, Speicherauslastung, Anzahl der CPU-Kerne und die CPU-Zyklen des Controller-Prozesses. Zusätzlich erfasst er über jeden AppServer-Prozess eine Reihe von Management-Daten: durchschnittliche Requests pro Sekunde, CPU-Zyklen und die Auslastung der request-verarbeitenden Threads. Die erfassten Management-Daten sendet der Controller in regelmäßigen Zeitabständen zum Master.

Der Controller implementiert auch eine Reihe von Plattform-Diensten (UserService, Logging, Storage, Deployment), die er über Java RMI veröffentlicht. Verwendet werden

die Dienste ausschließlich von den AppServern. Ein logischer Ansatz wäre daher die Implementierung direkt im AppServer. Dies würde allerdings die Startzeit und den Speicherverbrauch unnötig erhöhen. Darüber hinaus verbessert dieser Ansatz die Sicherheit, da die Plattform-Dienste physikalisch getrennt vom Anwendungscode ausgeführt werden.

5.4. Master

Der Master ist eine vergleichsweise kleine Komponente, die allerdings zwei sehr wichtige Aufgaben übernimmt: Erstens sammelt er die Monitoring-Daten aller Controller und AppServer. Zweitens ist er für die Last-Verteilung und die Anwendungs-Skalierung zuständig.

Beim Starten registriert sich der Master im ZooKeeper, indem er einen neuen transienten Datenknoten anlegt. In diesem Master-Knoten speichert er seine IP-Adresse. Falls der Knoten bereits existiert, wartet der Master über einen Watcher, bis der Knoten gelöscht wird. Folglich lassen sich mehrere Master-Prozesse starten, die als Backup-Master agieren. Sobald der aktive Master ausfällt, wird sein transienter Master-Knoten automatisch vom ZooKeeper gelöscht. Dieses Ereignis erkennen die Backup-Master und aktivieren sich. Jeder versucht daraufhin selbst einen neuen Master-Knoten zu erstellen. Dem schnellsten Backup-Master gelingt dies, womit er der neue aktive Master wird.

Die Controller überwachen ebenfalls den Master-Knoten. Sobald er verfügbar ist oder sich verändert, lesen sie die aktuelle IP-Adresse aus. Anschließend senden sie ihre Monitoring-Daten und die ihrer AppServer an den Master. Zur Datenübertragung wird das UDP-Protokoll in Kombination mit Google ProtoBuf {Google #70} eingesetzt. Das UDP-Protokoll implementiert keine Sicherungsmechanismen, womit Datenpakete verloren gehen können. Da die Controller aber regelmäßig die aktuellen Monitoring-Daten zum Master übertragen, ist der Verlust eines UDP-Pakets hinnehmbar. Daher wurden keine Sicherungsmechanismen auf Anwendungsebene implementiert. Aufgrund der fehlenden Sicherungsmechanismen können die Controller auch beim Ausfall des Masters weiterhin Daten senden. Diese Daten gehen verloren, bis ein Backup-Master aktiv ist und seine IP-Adresse von den Controllern ausgelesen wurde.

In einem ersten Implementierungs-Ansatz wurde Java RMI zur Übertragung der Management-Daten eingesetzt. Dabei führte der Master regelmäßig eine Ping-Methode in jedem Controller aus (vgl. AppScale, S. 31). Die Management-Daten wurden als Rückgabewert zum Master übertragen. Dieser Ansatz wurde allerdings aufgrund einer Reihe von Nachteilen verworfen. Erstens ist das RMI-Protokoll nicht fehlertolerant. Daher wurde eine erhebliche Menge an Quellcode für das Exception-Handling benötigt, was wiederum die Komplexität des Quellcodes erhöhte. Zweites ist das RMI-Protokoll verbindungsorientiert. Dadurch musste zu jedem Controller eine eigene RMI-Verbindung

verwaltet werden. Drittens ist das RMI-Protokoll nicht sprachunabhängig. Dies ist nachteilig, da eine Implementierung des AppServers in verschiedenen Programmiersprachen möglich sein soll.

Die JMS²⁶-Technologie ist ein weiterer Ansatz zur Übermittlung der Management-Daten. Aufgrund der Notwendigkeit eines Message Brokers wird sie aber nicht eingesetzt. Da die JMS-Spezifikation sehr umfangreich und leistungsfähig ist, sind Message Broker äußerst komplexe Server-Anwendungen. Für die Übertragung der Monitoring-Daten würde nur ein Bruchteil der bereitgestellten JMS-Funktionalität benötigt. Ein komplexer Message Broker würde daher ohne einen erheblichen Vorteil die Komplexität und damit Fehleranfälligkeit von TwoSpot erhöhen.

Im Vergleich zu JMS und RMI ist die Lösung mit UDP und ProtoBuf äußerst trivial. Trotzdem erfüllt sie die Anforderungen von TwoSpot besser als die aufgeführten Alternativen.

Der Master verwendet die Management-Daten, um Entscheidungen bezüglich des Load-Balancing zu treffen. Immer wenn das Frontend eine Controller-Adresse zu einer AppId erfragt, analysiert der Master alle verfügbaren Daten. Mithilfe eines Load-Balancing-Algorithmus (vgl. S. 78, Horizontale Skalierung) wählt er dann eine oder mehrere Controller-Adressen aus und gibt sie an das Frontend zurück.

Da der Master bereits die Management-Daten aller Controller und AppServer empfängt, wird er zusätzlich zum Monitoring der TwoSpot-Plattform eingesetzt. Dazu gibt er den Plattform-Status in regelmäßigen Zeitabständen über den Logging-Mechanismus aus. Eine Darstellung über ein Web-Interface oder eine JMX-Schnittstelle wurde noch nicht implementiert.

5.5. Frontend

Das Frontend ist im Wesentlichen ein Reverse-Proxy, der alle eingehenden HTTP-Requests an einen geeigneten Controller weiterleitet. Dazu extrahiert es aus jeder Request die AppId durch eine Analyse der URL (vgl. S. 59). Das Frontend verwaltet einen Cache, der jede AppId auf eine oder mehrere Controller-Adressen abbildet (vgl. Abb. 22). Mithilfe der extrahierten AppId wird nun ein Cache-Lookup durchgeführt. Falls mehrere Controller-Adressen ermittelt werden konnten, wählt das Frontend mithilfe eines einfachen Load-Balancing-Algorithmus (vgl. S. 78 Horizontale Skalierung) eine Adresse aus. Abschließend leitet es die Request an den ausgewählten Controller weiter.

²⁶ Java Message Service

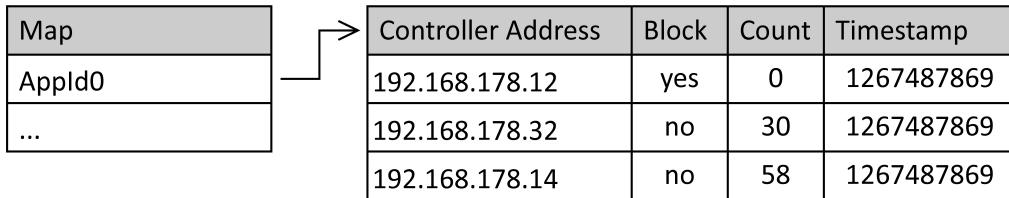


Abb. 22 Aufbau des Controller-Cache im Frontend

Falls der Cache-Lookup kein Ergebnis erbrachte, kontaktiert das Frontend den Master. Dieser ermittelt mithilfe eines Load-Balancing-Algorithmus (vgl. S. 78, Horizontale Skalierung) eine oder mehrere Controller-Adressen und gibt sie an das Frontend zurück. Dieses fügt die Adressen zum Cache hinzu. Alle Cache-Einträge verfallen nach dem Ablauf einer vorgegebenen Lebensdauer. Damit kann der Master die Zuweisung zwischen Applids und Controller-Adressen zu einem späteren Zeitpunkt verändern. Tritt bei der Kommunikation mit einem Controller ein Fehler auf, wird die Adresse ebenfalls aus dem Cache entfernt.

Das Frontend und der Controller verwenden eine eigene Reverse-Proxy-Implementierung auf Basis des Jetty-Servers. Der Frontend-Proxy nimmt zunächst alle eingehenden HTTP-Requests an. Er klonst sie in einer zweiten Request und ersetzt die Adresse durch die Adresse vom Ziel-Controller. Anschließend sendet er die zweite Request über die Jetty HTTP-Client-Bibliothek an den Controller. Die Bibliothek arbeitet asynchron. Damit die erste Request nicht aktiv auf die Client-Response warten muss, wird eine Continuation eingesetzt. Diese pausiert die erste Request. Der Client leitet alle Response-Daten der zweiten Request direkt an den Output-Stream der ersten Request und damit den Browser weiter. Sobald die Response vollständig ist, schließt der Client die pausierte Continuation der ersten Request ab. Der Proxy arbeitet damit vollständig asynchron. Durch den Continuation-Mechanismus wird eine unnötige Blockierung der request-verarbeitenden Threads vermieden.

Auch das Frontend wurde in Hinblick auf eine horizontale Skalierung entworfen. Beim Einsatz mehrerer Frontends muss die Belastung gleichmäßig aufgeteilt werden. Dazu existieren verschiedene Verfahren. Ein beliebter und sehr kostengünstiger Ansatz ist die Lastverteilung über DNS {RFC 1794 #146}{Kopparapu #177: 67}. Dabei bildet der DNS-Server einen Domain-Namen auf eine Liste von IP-Adressen und Frontends ab. Bei jeder Namensauflösung wird die Liste entsprechend eines Round-Robin-Verfahrens rotiert. Die Clients verwenden immer den ersten Listeneintrag, womit sich alle Clients auf die IP-Adressen der Frontends verteilen. Der Ansatz weist allerdings einen wesentlichen Nachteil auf: Clients cachen die Liste, aber berücksichtigen die TTL²⁷-Angaben vom DNS nicht {Kopparapu #177: 67}. Falls bei einem Belastungsanstieg weitere Frontend-Server gestartet werden, greifen die Clients damit nicht auf die

²⁷ Time To Live

aktualisierte Liste zurück. Auch beim Ausfall eines Frontends oder beim Downscaling verbleiben die ungültigen IP-Adressen im Client-Cache. Allerdings greifen sie dann auf eine andere IP-Adresse aus der Liste zurück. Ein weiterer Nachteil ist, dass der DNS-Server über keine Informationen bezüglich der Serverauslastung verfügt. Somit lässt sich die Belastung lediglich über einen lastunabhängigen Round-Robin-Ansatz verteilen {Kopparapu #177: 67}.

Eine Alternative stellen Load-Balancer auf ISO/OSI-Ebene 4 dar {Elson #147: 174}. Sie existieren entweder als Software (z. B. Linux Virtual Server {Zhang 11/8/2008 #178}) oder als teure Hardwarekomponenten. Bei diesem Ansatz durchlaufen alle Requests den zentralen Load-Balancer. Er verteilt sie dann gleichmäßig auf die Frontend-Server. Beim Hinzufügen oder Ausfall von Servern kann der Load-Balancer, im Unterschied zu DNS, zeitnah reagieren. Nachteilig ist hingegen, dass der gesamte Datenverkehr über den Load-Balancer abgewickelt wird. Dies kann besonders bei datenintensiven Anwendungen problematisch sein {Elson #147: 176}. Weiterhin ist die gesamte Plattform nicht mehr erreichbar, wenn der zentrale Load-Balancer ausfällt {Elson #147: 177}.

HTTP Redirection ist ein weiterer Load-Balancing-Ansatz {Kopparapu #177: 68}{Elson #147: 174}, der allerdings nur für das HTTP-Protokoll geeignet ist. Den ersten HTTP-Request richtet der Browser immer an den Load-Balancer. Dieser leitet ihn permanent über einen HTTP-Redirect auf einen Frontend-Server um. Alle folgenden Requests richtet der Browser damit direkt an diesen Frontend-Server. Der Load-Balancer kann ebenfalls zeitnah auf neue Server oder Serverausfälle reagieren. Außerdem stellt er keinen Engpass bei der Übertragung großer Datenmengen dar. Bei einem Ausfall können alle bereits umgeleiteten Browser weiterhin mit der Plattform kommunizieren. Ein Nachteil der Redirection ist, dass die Browser fest auf ein Frontend abgebildet werden. Da das Frontend aber einen Reverse-Proxy darstellt, verteilt es die Belastung wiederum gleichmäßig auf die Controller.

5.6. Sicherheitsaspekte

Wie bereits beschrieben, werden i. d. R. mehrere AppServer gleichzeitig auf einem Server und unter einem Betriebssystem ausgeführt. Daher ist der AppServer als Execution Environment für die Isolation der Anwendung verantwortlich. Die Sicherheitsmechanismen befinden sich allerdings nicht nur im AppServer, sondern in mehreren Plattform-Komponenten.

Grundsätzlich werden alle Anwendungscodes als potenzielle Gefahr eingestuft. Da eine Isolation über den Java Class-Loader nicht ausreichend ist {Grzegorz Czajkowski #144: 354}, wird jede Anwendung in einem eigenen AppServer und damit einem eigenen Prozess betrieben. Trotzdem können sich die Anwendungen gegenseitig beeinträchtigen. Zum Beispiel in dem sie den Prozess einer fremden Anwendung beenden. Der AppServer enthält weitere Sicherheitsmechanismen, um derartige Szenarien aus-

zuschließen. Diese sind allerdings abhängig vom Software-Stack und der Implementierung des AppServers selbst. Der AppServer mit den Software-Stacks für Python und Java greift auf die Sicherheitsmechanismen der Java VM zurück. Dazu implementiert er einen eigenen Sicherheitsmanager, der dem Anwendungscode nur eine sehr eingeschränkte Menge an Rechten zuweist. Folglich können die Anwendungen lediglich Dateien aus dem Anwendungsverzeichnis lesen, aber nicht schreiben. Zugriffe auf die Netzwerk-Funktionalität, die Thread-Verwaltung oder Systemfunktionen sind ebenfalls nicht möglich.

Der Sicherheitsmanager verhindert damit, dass sich Anwendungen direkt gegenseitig beeinflussen. Weiterhin besteht allerdings die Möglichkeit von Anwendungsfehlern oder z. B. DoS²⁸-Angriffen. Führt z. B. eine Anwendung eine CPU-intensive Aufgabe über einen längeren Zeitraum aus, kann dies die Leistungsfähigkeit aller anderen Anwendungen auf dem Server beeinträchtigen. Daher überwacht der AppServer die Verarbeitungsdauer jeder Request. Überschreitet sie einen festgelegten Wert von z. B. 30 Sek., wird der gesamte AppServer-Prozess beendet und damit auch alle Requests. Der Controller interpretiert dies als Absturz des AppServers und beantwortet alle terminierten Requests mit dem HTTP Fehler-Code 504 (Gateway timeout). Dieses Verhalten ist extrem aber notwendig, da Java keine Möglichkeit zur Terminierung einzelner Threads bietet {Overview Java Platform SE 6 1/6/2010 #86}. Da die TwoSpot-Plattform konsequent auf eine horizontale Skalierung ausgelegt ist, kommen statt einiger weniger leistungsfähiger AppServer, viele kleine AppServer zum Einsatz. Folglich betrifft der Absturz eines AppServers, abhängig zur Anwendungsgröße, nur eine relativ kleine Menge an Requests. Daher nimmt die Stabilität der Plattform einen höheren Stellenwert ein, als die störungsfreie Ausführung einer einzelnen Anwendungsinstanz.

Ein weiterer Sicherheitsmechanismus betrifft die Übertragung großer Datenmengen. Die Netzwerkkapazität eines Servers muss zwischen allen Anwendungen aufgeteilt werden. Um Engpässen vorzubeugen, überwacht der Controller die übertragene Datenmenge jeder Request. Überschreitet sie einen gewissen Grenzwert, bricht er die Übertragung ab.

Der Controller veröffentlicht mithilfe von Java RMI eine Reihe von Plattform-Diensten. Aufgrund der Einschränkungen des Sicherheitsmanagers können die Anwendungen aber nicht direkt auf die Schnittstelle zugreifen. Daher implementiert der AppServer spezielle Proxies, die den Zugriff auf die Dienste ermöglichen. Die Proxies sind in der Protection Domain des AppServers implementiert. Folglich erhalten sie die notwendigen Rechte, um auf die RMI-Dienste vom Controller zuzugreifen. Ihre Methoden werden aber grundsätzlich aus dem Anwendungscode heraus aufgerufen und erhalten damit lediglich die Rechte der Anwendung. Diese gestatten keine RMI-Aufrufe. Aus

²⁸ Denial of service

diesem Grund erfolgt der endgültige RMI-Aufruf innerhalb eines privilegierten Blocks. Dieser befindet sich in der Protection Domain vom AppServer und erhält damit die notwendigen Rechte zur Ausführung des RMI-Aufrufs (vgl. Abb. 23).

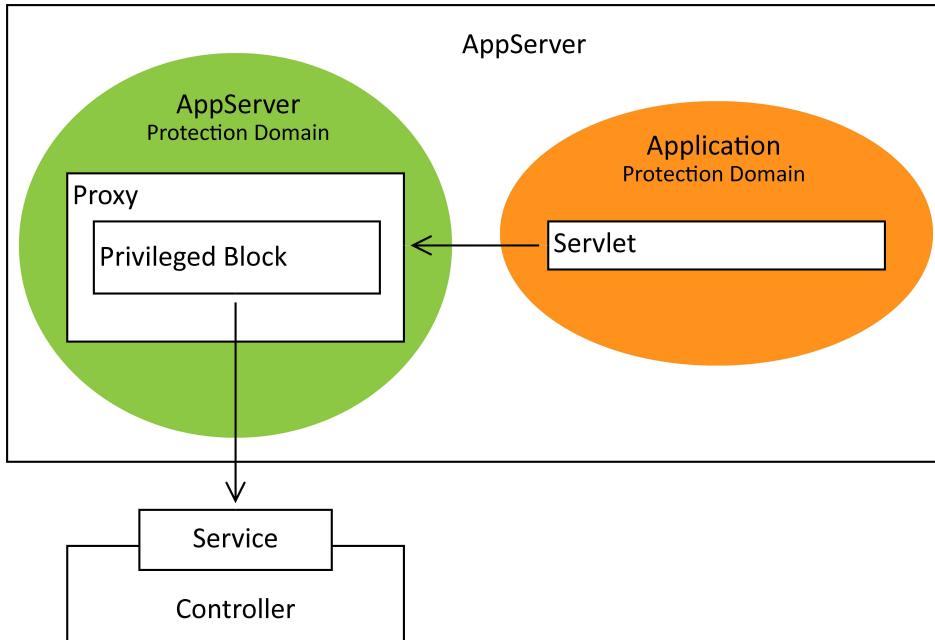


Abb. 23 Anwendung greift über einen Proxy auf einen Plattform-Dienst zu

Der Controller stellt weiterhin auch privilegierte Dienste zur Verwaltung und Administration der TwoSpot-Plattform bereit. Diese dürfen aber nur von bestimmten, vordefinierten Anwendungen genutzt werden. Dazu weist der Controller jedem AppServer ein Token, ähnlich einer HTTP Session-ID zu. Das Token wird beim Start des AppServers mit einem Parameter übertragen. Alle Methoden der privilegierten Dienste erwarten dieses Token als ersten Parameter. Der Controller kann dann mithilfe des Tokens die Autorisierung des Aufrufs überprüfen.

Die HBase-Datenbank implementiert selbst keinerlei Sicherheitsmechanismen. Aufgrund der Skalierbarkeit und Performance werden die Daten aller Anwendungen gemeinsam in einigen wenigen großen Tabellen gespeichert (vgl. S. 71, Datenspeicher). Dieser Ansatz verschärft die Sicherheitsproblematik zusätzlich. Aufgrund verschiedener Gründe (vgl. S. 71) wurde ein Storage-Dienst implementiert, der zwischen der HBase-Datenbank und der JDO-Implementierung agiert. Der Controller wickelt alle Zugriffe auf diesen Storage-Dienst ab. Die AppServer müssen dabei ebenfalls den zuvor erwähnten Token übertragen. Diesem kann der Controller eine AppId zuweisen, die abschließend zur Kommunikation mit dem Storage-Dienst eingesetzt wird. Folglich stellt der Token sicher, dass Anwendungen tatsächlich unter ihrer eigenen AppId auf die Storage zugreifen und keine fremden Daten auslesen.

5.7. Datenspeicher

Zur dauerhaften Speicherung von Daten kommt die HBase-Datenbank zum Einsatz (vgl. S. 50, Vergleich). Sie wird von der TwoSpot-Plattform selbst und den Anwendungen verwendet. Außer der Datenbank existiert momentan keine weitere Möglichkeit zur dauerhaften Datenspeicherung.

Im ersten Schritt wurde HBase über das Data Nucleus-Framework {DataNucleus #68} mit dem JDO HBase-Plugin verwendet. Der Ansatz hat sich durch einige Gründe als impraktikabel erwiesen. Erstens erfolgt der Zugriff auf die HBase-Datenbank über eine RMI-Verbindung. Aufgrund des Sicherheitsmanagers dürfen die Anwendungen aber auf keine Netzwerkdienste zugreifen. Außerdem lässt sich die Anzahl der Datenbank-Verbindungen nur sehr umständlich durch die Plattform überwachen.

Zweitens stellt HBase keine Query-Sprache für die einfache und effiziente Abfrage von Daten zur Verfügung. Das Plugin lädt daher den gesamten Datenbestand in den Arbeitsspeicher. Dort analysiert es die Daten anhand des Queries. Dieses Vorgehen ist aufgrund der sehr hohen CPU- und Speicherbelastung nicht praktikabel.

Drittens legt das Plugin zu jeder persistenten Klasse eine neue HBase-Tabelle an. Weder HBase noch Google Bigtable sind für diesen Anwendungsfall konzipiert. Das Hauptproblem besteht dabei in der Zuweisung von Tabellen zu Region-Servern. Wenn eine Tabelle über einen längeren Zeitraum nicht genutzt wird, verfällt ihre Zuweisung zu einem Region-Server. Sie wird damit nicht weiter verwaltet und ist inaktiv. Falls eine Anwendung auf eine inaktive Tabelle zugreift, muss diese erst wieder einem Region-Server zugewiesen werden. Dieser Vorgang nimmt einige Sekunden in Anspruch, währenddessen die Anwendung warten muss. Aufgrund der langen Wartezeit und der damit entstehenden Fehler (Timeouts) ist dieses Verhalten für TwoSpot nicht akzeptabel.

Um die aufgeführten Probleme zu umgehen, wurde ein neues JDO-Plugin und eine neue Speicherschicht entwickelt, die sich am Google App Engine Datastore {Google Inc. #69} orientiert. Ähnliche Konzepte werden aber auch auf Basis gewöhnlicher relationaler Datenbanken eingesetzt {How FriendFeed uses MySQL #205}. Die neue Speicherschicht wird nachfolgend einfach als Storage bezeichnet (vgl. Abb. 24).

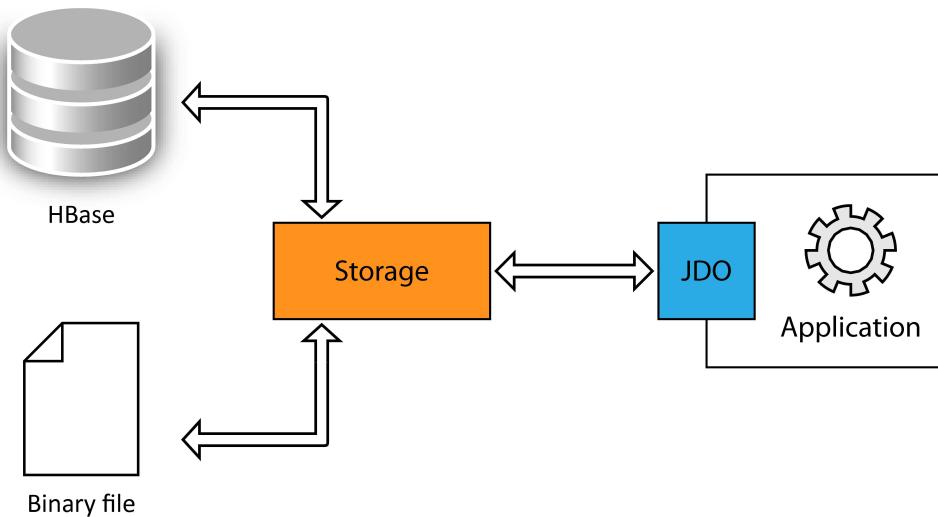


Abb. 24 Die Anwendung greift über eine JDO-Schnittstelle auf die Storage zu. Das Speicher-Backend der Storage ist austauschbar

Die Tabellenstruktur der Storage unterscheidet sich maßgeblich von der des Data Nucleus-Plugins. Anstelle einer Tabelle für jede persistente Klasse wird eine einzige große Tabelle mit dem Namen Objects verwendet. Sie besteht lediglich aus zwei Spalten. Die erste Spalte ist der Row-Key zur Identifikation einer Tabellenzeile. In der zweiten Spalte werden die Objekte in serialisierter Form als Byte-Array gespeichert. Die Eigenschaften der Objekte werden also nicht in jeweils eigenen Tabellenspalten abgelegt.

Zum Speichern eines Objekts muss es zunächst serialisiert werden. In einem ersten Versuch kamen die Serialisierungs-Mechanismen von Java zum Einsatz (ObjectInputStream und ObjectOutputStream). Diese erwiesen sich allerdings als ungeeignet, da im Deserialisierungs-Prozess die Java-Klasse des serialisierten Objekts benötigt wird. Innerhalb der Storage ist diese allerdings nicht verfügbar und lässt sich aus Sicherheitsgründen auch nicht zusammen mit dem serialisierten Objekt ablegen. Zudem würden die class-Dateien unnötig viel Speicherplatz beanspruchen. Stattdessen wird die Google ProtoBuf-Bibliothek verwendet. Ein weiterer Vorteil von ProtoBuf ist die kompakte Speicherform und die vergleichsweise hohe Geschwindigkeit bei der Serialisierung und Deserialisierung {Benchmarking #93}.

ProtoBuf besteht aus zwei elementaren Komponenten: Erstens aus Funktionen zur Serialisierung von Daten eines bestimmten Typs. Zweitens aus einem Compiler, der IDL-Dateien in Java-Quellcodes übersetzt. Innerhalb der IDL-Dateien lassen sich Datenstrukturen definieren. Der Compiler übersetzt diese IDL-Datenstrukturen in kompatible Java-Datenstrukturen und eine Hilfsklasse. Mit dieser lassen sich die Java-Datenstrukturen komfortabel serialisieren und deserialisieren. Für die Storage werden die IDL-Dateien und der Compiler aber nicht verwendet. Stattdessen greift sie auf die JDO-Annotationen zur Definition des Datenschemas zurück.

Die Serialisierung findet im JDO-Plugin statt. Beim Speichern oder Aktualisieren eines Objekts wird eine Entity-Message angelegt. Alle Message-Datenstrukturen können sich mithilfe von ProtoBuf selbst in einen Byte-Array serialisieren und aus einem bestehenden Byte-Array materialisieren. Das JDO-Plugin liest zunächst die Metadaten der persistenten Klasse aus. Diese sind über die JDO-Annotationen definiert und enthalten Informationen zu jedem persistenten Feld.

Nun wird zu jedem Feld eine eigene Index-Message erstellt. Sie speichert die Nummer, den Namen und den Datentyp des Feldes. Die Storage unterstützt bislang nur skalare Datentypen. Alle verfügbaren Datentypen sind in einem Enum definiert und werden über eine Nummer identifiziert. Nachdem alle Klassen-Felder in Index-Messages überführt wurden, werden sie in Listenform zur Entity-Message hinzugefügt.

Nun speichert das JDO-Plugin das Objekt selbst bzw. die Feld-Daten. Zu jedem Feld wird wiederum eine eigene Property-Message angelegt. In dieser wird die Nummer, der Datentyp und letztendlich der Feldwert gespeichert. Felder deren Wert gleich NULL ist, werden allerdings ignoriert. Die erstellten Messages werden dann wieder in Listenform zur Entity-Message hinzugefügt. Da Felder ohne Feldwert nicht gespeichert werden, kann sich die Anzahl der Index-Messages von den Property-Messages unterscheiden.

Abschließend wird die Entity-Message in einen Byte-Array serialisiert. Zunächst werden die Index-Messages verarbeitet. Diese können sich jeweils selbst serialisieren. Anschließend wird der vollqualifizierte Klassename der persistenten Klasse als String abgelegt. Abschließend werden alle Property-Messages gespeichert, die sich wiederum selbst serialisieren. Der gesamte Aufbau einer Entity-Message ist in Abb. 25 dargestellt.

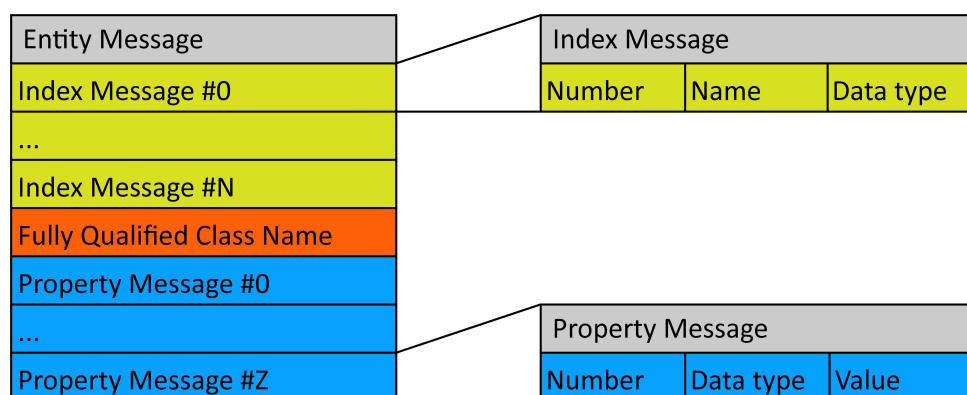


Abb. 25 Aufbau einer ProtoBuf-Message zum Speichern eines persistenten Objekts

HBase verwendet den Row-Key zur Identifikation einer Tabellenzeile und zur Sortierung der Tabelle in einer aufsteigenden Reihenfolge. Dieses Konzept erweist sich beim Speichern der serialisierten Objekte allerdings als problematisch. Jedes Objekt erfordert einen eindeutigen Row-Key. Dieser lässt sich z. B. ähnlich zu einer RDB über eine

Sequenz bzw. eine, in der Datenbank gespeicherte Zählervariable erstellen. Die Sequenz wird für jedes neue Objekt inkrementiert, womit der neue Sequenzwert gleich dem neuen Row-Key ist. Durch die aufsteigende Tabellensortierung wird damit aber jedes neue Objekt am Tabellenende eingefügt. Folglich belasten Einfüge-Operationen auch nur den Region-Server, der die letzte Tabellen-Region verwaltet (vgl. Abb. 26). Besser wäre es hingegen, wenn die Einfüge-Operation alle Region-Server gleichmäßig belastet.

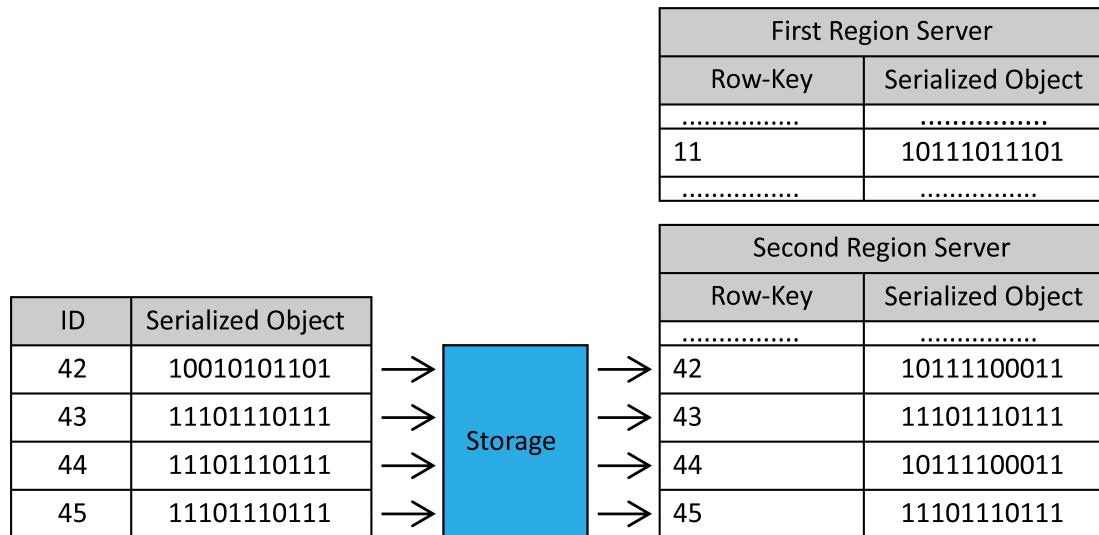


Abb. 26 Sequenz zum Generieren der Row-Keys

Dazu wird zusätzlich zur Sequenz ein Zufallsgenerator verwendet. Jeder Row-Key setzt sich damit aus einem Zufallswert und der Sequenz zusammen. Damit werden neue Objekte an zufälligen Positionen innerhalb der Tabelle eingefügt. Folglich verteilen sich auch die Einfüge-Operationen und damit die Belastung auf alle Region-Servers, womit die Einfüge-Operationen horizontal skalieren (vgl. Abb. 27).

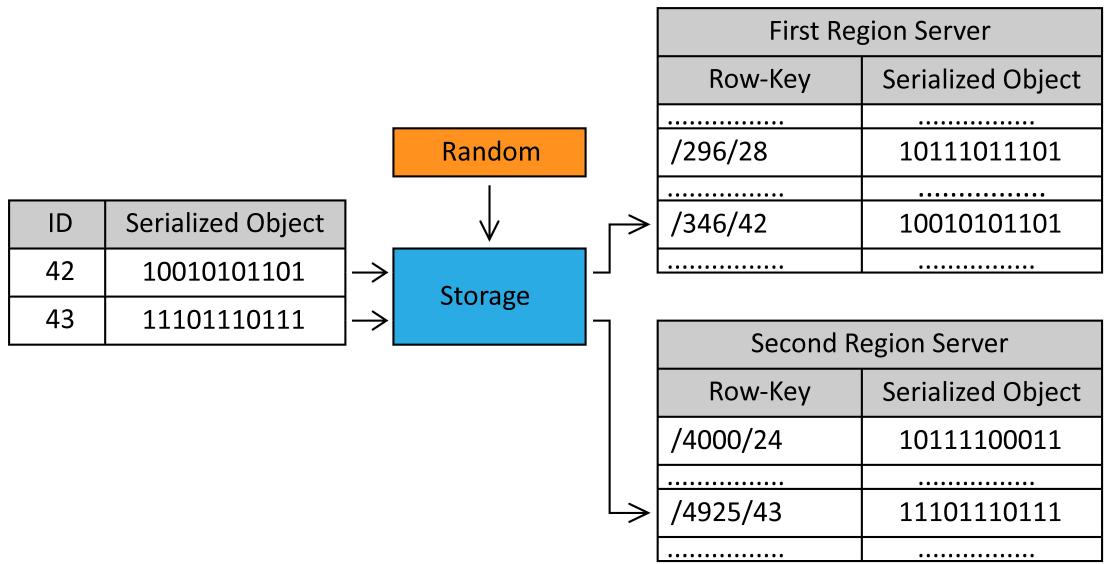


Abb. 27 Gleichmäßige Verteilung der Objekte über alle Region-SERVER mithilfe einer Zufallskomponente im Row-Key

Ein Problem bleibt aber weiterhin bestehen: Jede Einfüge-Operation muss einmal auf die Sequenz zugreifen, die ebenfalls in der Datenbank gespeichert ist. Damit ergibt sich ein weiterer Engpass. Die Situation verschärft sich zusätzlich, da jeder Zugriff innerhalb einer Transaktion erfolgen muss. Dem wirken zwei Mechanismen entgegen: Erstens inkrementiert die Storage die Sequenz nicht nur um eins, sondern um z. B. 10. Somit können die folgenden 9 Objekte ebenfalls mit einem Row-Key versorgt werden, ohne auf die Sequenz zuzugreifen.

Zweitens ist die Sequenz als Sharded Counter implementiert. Eine Sequenz ist über einen Long-Wert mit 64 Bit realisiert. Damit lassen sich max. $X = 2^{64-1}$ positive Werte (incl. 0) abbilden. Dieser Raum wird nun in z. B. 10 Teile zerschnitten, womit sich 10 Teilsequenzen ergeben. Jede Teilsequenz umfasst damit $Y = 2^{64-1} * 10^{-1}$ Werte. Die einzelnen Teilsequenzen werden chronologisch, mit 0 beginnend nummeriert. Nun wird jede Teilsequenz durch eine eigene Zählervariable in der Datenbank gespeichert. Beim Erstellen eines neuen Row-Keys wird nun zunächst über einen gleichverteilten Zufallsgenerator eine der 10 Teilsequenzen ausgewählt und inkrementiert. Der neue Sequenzwert S ergibt sich dann aus:

$$S = Y * \text{Nummer der Teilsequenz} + \text{Zählerstand}$$

Durch das Zerschneiden einer Sequenz in mehrere Teile, lässt sich die Belastung auf mehrere Zähler verteilen (vgl. Abb. 28).

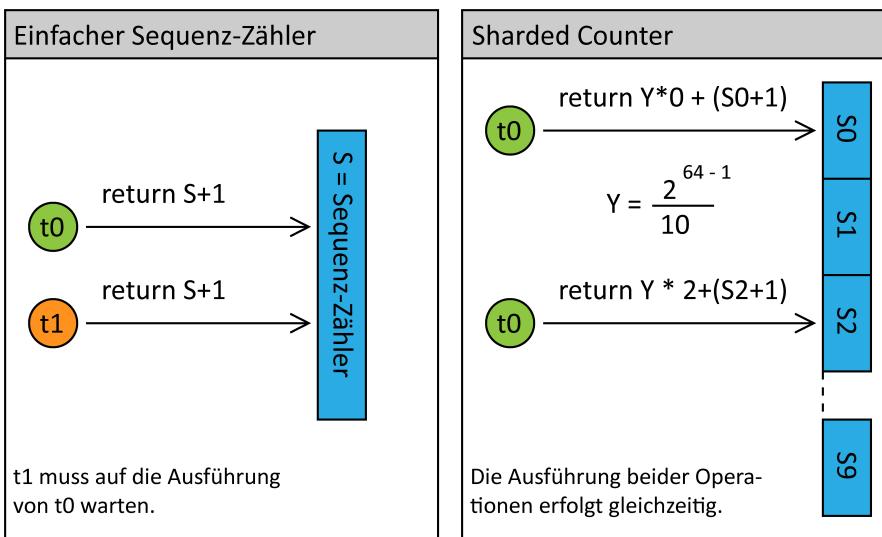


Abb. 28 Unterschied zwischen einer normalen Sequenz und einem Sharded Counter

HBase implementiert selbst keine Query-Sprache, in die sich JDO-Queries übersetzen lassen. Folglich müssen alle Daten zunächst in den Speicher geladen und über den JDO-Query verarbeitet werden. Dieses Vorgehen ist allerdings zeit-, speicher- und CPU-intensiv. Um einfache und häufig verwendete JDO-Queries trotzdem effizient ausführen zu können, kommen Index-Tabellen zum Einsatz. Ihre Funktionsweise ist sehr ähnlich zum Google App Engine Datastore {Barrett 12/15/2009 #78} und wird an dieser Stelle lediglich am Beispiel der IndexByProperty-Tabelle ausführlich erläutert. Das Konzept lässt sich aber in gleicher Form auf verschiedene Index-Tabellen übertragen.

Die IndexByProperty-Tabelle besteht aus den beiden Spalten: Row-Key und Key. Die Werte der Key-Spalte referenzieren auf Row-Keys der Objects-Tabelle und damit auf persistente Objekte. Beim Speichern eines neuen Objekts wird, entsprechend dem zuvor beschriebenen Verfahren, der Row-Key für die Objects-Tabelle erstellt. Dieser wird nachfolgend als Objekt-ID bezeichnet. Anschließend wird das Objekt deserialisiert. Für jedes Objekt-Feld wird dann ein neuer Eintrag in der IndexByProperty-Tabelle angelegt. Die Row-Keys dieser Einträge sind von besonderem Interesse. Sie setzen sich aus den folgenden Komponenten zusammen:

- 1) AppId
- 2) Object-Kind (Klassenname ohne das Paket)
- 3) Feldname
- 4) Feldwert
- 5) Objekt-ID

Dieser Row-Key ist eindeutig, da bereits die Objekt-ID das Objekt eindeutig identifiziert und der Feldname innerhalb des Objekts eindeutig ist. Für jedes gespeicherte Objekt existiert somit genau ein Eintrag in der Objects-Tabelle und eventuell mehrere Einträge in der IndexByProperty-Tabelle (vgl. Abb. 29).

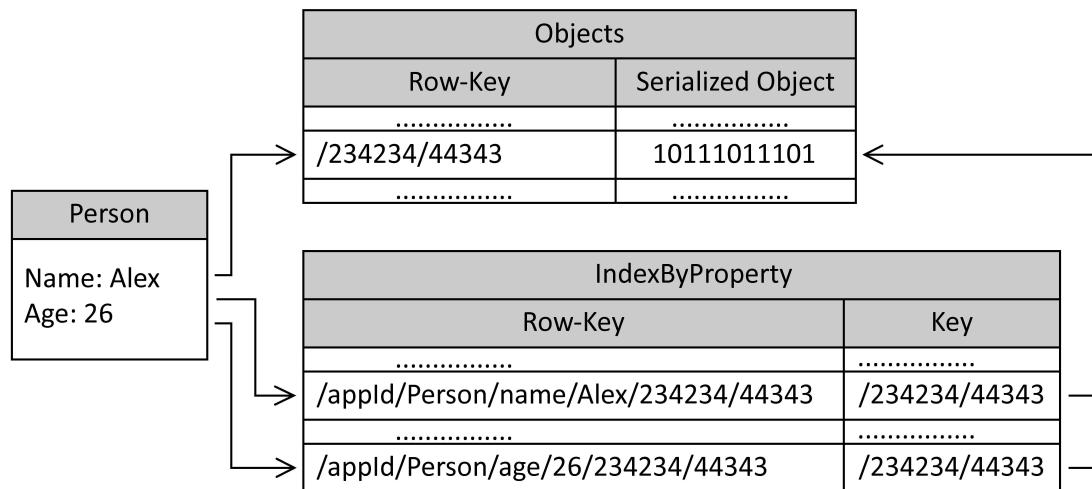


Abb. 29 Einfügen eines Objekts in die Objects- und IndexByProperty-Tabellen

Aufgrund der aufsteigenden Sortierung befinden sich alle Index-Zeilen mit denselben Werten für AppId, Object-Kind, Feldname und Feldwert in einem zusammenhängenden Block, auch wenn sich die Objekt-ID unterscheidet (vgl. Abb. 30).

IndexByProperty	
Row-Key	Key
.....
/appId/Person/alter/12/234234/44343	/234234/44343
/appId/Person/alter/16/234534/42233	/234534/42233
/appId/Person/alter/16/256656/12332	/256656/12332
/appId/Person/alter/23/568234/33443	/568234/33443
/appId/Person/alter/28/123831/34334	/123831/34334
/appId/Person/name/Alf/341244/44	/341244/44
/appId/Person/name/Tom/234323/67	/234323/67
.....

Abb. 30 Interner Aufbau der IndexByProperty-Tabelle

Diese Eigenschaft lässt sich nun für einfache JDO-Queries ausnutzen. Im einfachsten Fall kann eine effiziente Suche nach Objekten mit einem bestimmten Feldwert realisiert werden. Dazu wird ein neuer Row-Key für einen Table-Scan erstellt, der nachfolgend als Search-Key bezeichnet wird. Er setzt sich ebenfalls aus AppId, Object-Kind, Feldname und dem gesuchten Feldwert zusammen, enthält allerdings keine Objekt-ID. Diese Angaben lassen sich aus einem JDO-Query ermitteln. Wird der Search-Key nun in Kombination mit einem Table-Scan ausgeführt, beginnt der Scan direkt vor dem eben erwähnten zusammenhängenden Block. Der Table-Scan wird so lange ausgeführt, bis sich der gesuchte Feldwert verändert. Abschließend lassen sich anhand der ermittelten Index-Zeilen die entsprechenden Objekte effizient mithilfe eines Table-Gets aus der Objects-Tabelle auslesen (vgl. Abb. 31).

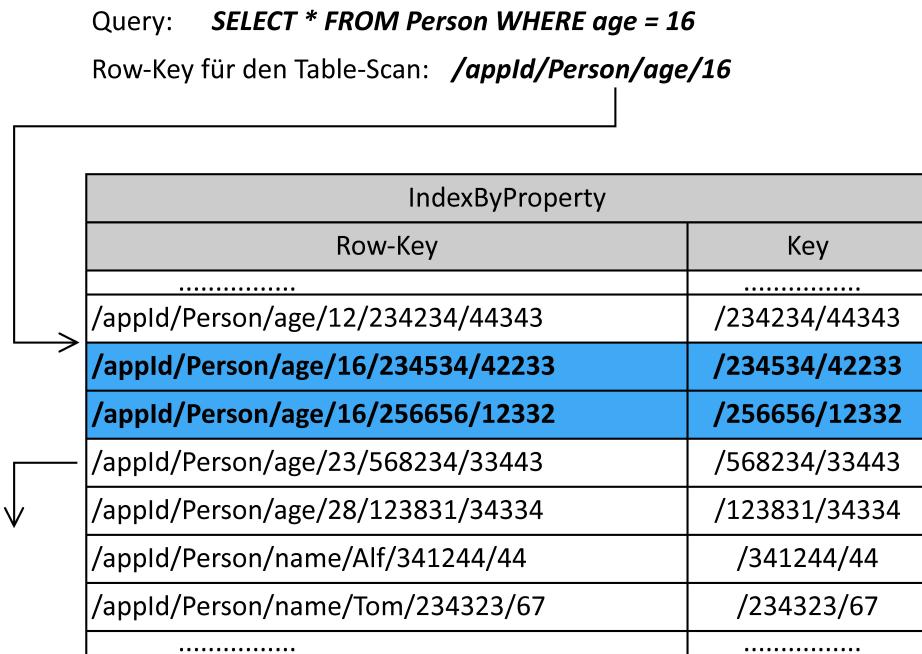


Abb. 31 Ein Query mit einem Gleich-Operator wird in einen Table-Scan übersetzt

Nach demselben Konzept lassen sich JDO-Queries mit einem Größer-, Kleiner-Operator oder Kombinationen aus diesen ausführen. Für den Größer-Operator wird der Table-Scan z. B. erst gestoppt, wenn sich der gescannte Feldname ändert. Bei einem Kleiner-Operator wird im Search-Key kein Feldwert angegeben und der Table-Scan wird gestoppt, sobald der gescannte Feldwert gleich dem gesuchten Wert ist.

5.8. Horizontale Skalierung

Die TwoSpot-Plattform wurde auf eine horizontale Skalierung ausgelegt, womit eine Anwendung parallel auf mehreren Servern ausgeführt werden kann. Die Anzahl der Anwendungsinstanzen ist dabei abhängig zur Belastung und wird automatisch durch die Plattform gesteuert. Steigt die Belastung an, werden zusätzliche Instanzen gestartet. Diese Aufgabe übernimmt der Master. Das Frontend kontaktiert den Master regelmäßig, um ein Mapping zwischen einer AppId und Controller-Adressen zu ermitteln. Bei jedem Aufruf analysiert der Master alle verfügbaren Management-Daten und erstellt mithilfe eines Load-Balancing-Algorithmus eine Controller-Liste zur AppId. Das Frontend verteilt dann alle entsprechenden Requests gleichmäßig über die Controller in dieser Liste.

Im ersten Schritt ermittelt der Load-Balancing-Algorithmus alle Controller, unter denen die Anwendung bereits ausgeführt wird. Sie werden direkt zur Ergebnisliste hinzugefügt. Falls die Liste weiterhin leer ist oder die Hälfte der ermittelten Controller eine hohe Auslastung der request-verarbeitenden Threads aufweist, werden die restlichen Controller bewertet. Die Bewertung wird auf Basis der folgenden Management-Daten gebildet:

- 1) Anzahl der ausgeführten Anwendungen.
- 2) CPU-Auslastung.
- 3) Speicherauslastung.
- 4) Auslastung der request-verarbeitenden Threads.
- 5) Durchschnittliche Requests pro Sekunde.

Für jeden verbleibenden Controller werden diese Management-Daten mit vorgegebenen Gewichten multipliziert. Die aufaddierten Ergebnisse ergeben dann die Bewertung des Controllers. Der Controller mit der besten Bewertung (mit dem kleinsten Wert) wird nun zur Ergebnisliste hinzugefügt. Abschließend wird die gesamte Liste an das Frontend zurückgegeben.

Das Frontend nimmt alle erhaltenen Controller-Adressen in den lokalen Cache auf. Damit lassen sich für eingehende Requests effizient die Controller-Adressen ermitteln. Das Mapping zwischen AppId und Controller-Adressen ist allerdings nicht statisch. Änderungen treten z. B. auf, wenn sich die Belastung einer Anwendung verändert. Aus diesem Grund verfallen alle Cache-Einträge eine Minute nachdem sie hinzugefügt wurden. Somit ist das Frontend gezwungen, regelmäßig das aktuelle Mapping vom Master zu laden. Die Cache-Einträge verfallen auch, falls ein Kommunikationsproblem mit dem entsprechenden Controller auftritt.

Eingehende Requests verteilt das Frontend gleichmäßig auf alle, zur AppId bekannten Controller-Adressen. Dazu verwaltet es für jeden Controller einen Request-Zähler. Beim Zustellen einer Request wird der Zähler inkrementiert und beim Abschließen einer Response dekrementiert. Eingehende Requests werden immer an den Controller mit dem kleinsten Zählerstand und damit der vermutlich geringsten Belastung weitergeleitet.

Sobald sich die Belastung auf eine Anwendung reduziert, muss die Plattform überschüssige AppServer beenden. Dieser Mechanismus befindet sich im Controller, in Form mehrerer Balancing-Processors. Diese implementieren jeweils einen Algorithmus, der nach überflüssigen AppServern sucht und diese beendet. Im Rahmen der Arbeit wurden drei Balancing-Processors implementiert. Der erste ermittelt alle AppServer, die seit einer vorgegebenen Zeit keine Request mehr verarbeitet haben und damit im Leerlauf betrieben werden. Wenn ein AppServer gefunden wird, setzt der Balancing-Processor seinen Lifecycle-Zustand auf KILLED. Damit wird er automatisch durch die Job-Queue beendet.

Der zweite Balancing-Processor sucht AppServer mit einer geringen Belastung. Diese liegt vor, wenn durchschnittlich weniger als 30% der request-verarbeitenden Threads genutzt werden. Die gefundenen AppServer werden allerdings nicht direkt beendet. Stattdessen prüft der Balancing-Processor, ob die Plattform mehrere Instanzen der entsprechenden Anwendung ausführt. Für diese Synchronisationsaufgabe wird der

ZooKeeper eingesetzt. Beim Registrieren einer Anwendung wird automatisch ein Anwendungsverzeichnis in der Verzeichnisstruktur des ZooKeepers erstellt. Sobald ein Controller eine Anwendung startet, erstellt er einen eindeutigen Datenknoten im Anwendungsverzeichnis. Sein Name ist gleich der GUID²⁹ des Controllers (vgl. Abb. 32). Folglich ist die Anzahl der Anwendungsinstanzen gleich der Anzahl der Datenknoten im Anwendungsverzeichnis. Falls mehr als eine Anwendungsinstanz existiert, beendet der Balancing-Processor den AppServer über den DROPPED-Zustand.

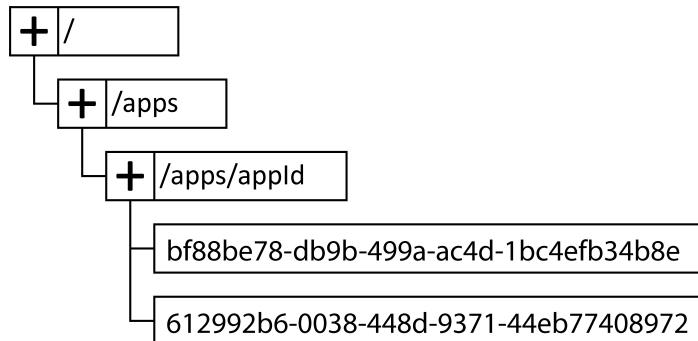


Abb. 32 Verzeichnisstruktur im ZooKeeper mit einem Anwendungsverzeichnis

Der letzte Balancing-Processor ist nur aktiv, wenn der Controller überlastet ist. Das ist der Fall, wenn der Controller nicht über genug freie request-verarbeitende Threads verfügt. In diesem Zustand wird der AppServer mit der geringsten Belastung bzw. der geringsten CPU-Auslastung ermittelt und über den BLOCKED-Zustand beendet. Dieser Vorgang wird fortgesetzt, bis nur noch ein einzelner AppServer verbleibt oder der Controller nicht mehr überlastet ist. Den stark belasteten AppServern werden damit alle Ressourcen zur Verfügung gestellt.

Die beiden Zustände DROPPED und BLOCKED verhalten sich ähnlich zu KILLED, womit der AppServer automatisch über die Job-Queue beendet wird. Im Unterschied zu KILLED, führen die beiden Zustände aber zu einer Anwendungssperre. Für die Dauer der Sperre wird die Anwendung vom Controller nicht neu gestartet. Alle für die Anwendung eingehenden Requests werden mit dem internen HTTP Fehler-Code 803 abgewiesen (vgl. S. 64). Das Frontend erkennt diesen Fehler und markiert den Controller-Eintrag im Cache entsprechend. Anschließend wandelt es den Fehler in den HTTP-Status 301 (Redirect) um und sendet die Response an den Client-Browser. Dieser sendet die Request daraufhin erneut zum Frontend, das sie dann an einen anderen Controller weiterleitet. Dieser Mechanismus verhindert den direkten Neustart einer Anwendung, nachdem sie ein Balancing-Processor beendet hat.

Für eine vordefinierte Zeit leitet das Frontend keine Requests an markierte Controller weiter. Auch nicht, wenn das Mapping zwischen ApplId und der Controller-Adresse

²⁹ Global Unique Identifier

durch den Master erneuert wird. Dieses Verhalten ist erforderlich, da der Master über keine Informationen bezüglich der Anwendungssperren verfügt.

Zusammengefasst befinden sich die Mechanismen zur Skalierung sowohl im Controller als auch im Frontend und Master. Der Master ist für den Start und die Verteilung neuer AppServer auf die verfügbaren Server zuständig. Im Gegensatz dazu ist der Controller für das Beenden von AppServern verantwortlich. Beide Interessen sind gegenläufig, was zu einer automatischen Justierung der ausgeführten Anwendungsinstanzen führt.

Potenziell können bei den aufgeführten Skalierungsmechanismen verschiedene Schwingungs-Szenarien auftreten. Als Gegenmaßnahme werden die AppServer nur beendet, falls sie ungenutzt sind. Nach dem Beenden einer Anwendung kann sie der Controller erst nach Ablauf einer bestimmten Wartezeit neu starten. Nach dem Start kann sie wiederum für eine bestimmte Zeit nicht beendet werden. Die Skalierungsmechanismen sind außerdem in der Annahme entworfen, dass immer eine ausreichende Menge an Controllern und Servern zur Verfügung steht.

5.9. Benutzersitzungen

In einer skalierbaren Architektur ist die Verwaltung von Benutzersitzungen (Session-Handling) besonders kritisch. Normalerweise verwaltet ein Anwendungsserver alle Sitzungsdaten lokal im Arbeitsspeicher {Zend Technologies 2005 #187: 4}. In einer verteilten Architektur lässt sich dieser Ansatz allerdings nur in Kombination mit Sticky-Sessions einsetzen {Zend Technologies 2005 #187: 5}. Dabei kommt ein Load-Balancer zum Einsatz, der alle Anfragen einer Benutzersitzung an denselben Anwendungsserver und damit dieselbe Anwendungsinstanz weiterleitet. Dazu ist allerdings ein leistungsfähiger Load-Balancer erforderlich. Ein Nachteil dieser Lösung ist auch, dass ein dynamischer Lastausgleich zwischen den Servern nicht möglich ist. Nach dem Zuweisen einer Sitzung an einen Server, lässt sich die Sitzung nicht mehr auf einen anderen Server verschieben {Zend Technologies 2005 #187: 5}.

Alternativ können die Sitzungsdaten zentral in einem Speichersystem (z. B. einer Datenbank oder einem verteilten Dateisystem) abgelegt werden {Zend Technologies 2005 #187: 5}. Jede Request liest die Sitzungsdaten aus dem Speichersystem aus und aktualisiert sie bei Änderungen. Dies führt zu einer sehr starken Belastung des zentralen Speichersystems, was die Skalierbarkeit wiederum einschränkt {Zend Technologies 2005 #187: 5}.

Die In-Memory-Replication der Sitzungsdaten ist ein weiterer Ansatz {Penchikala 11/24/2004 #189}. Jeder Anwendungsserver verwaltet dabei die Sitzungsdaten lokal im Arbeitsspeicher. Nach dem Verarbeiten einer Request prüft er, ob sich die Sitzungsdaten verändert haben. In diesem Fall sendet er die Änderungen inkrementell über einen Multicast-Mechanismus an alle anderen Server. Da die Daten über Multicasts verteilt

werden, skaliert der Ansatz linear bis zu einer bestimmten Anzahl an Servern. Alternativ können die Sitzungsdaten zentral auf dem Server verbleiben, auf dem die Sitzung erstellt wurde. Wenn ein anderer Server die Daten benötigt, ruft er sie explizit von diesem Server ab. Die Sitzungsdaten verteilen sich damit auf alle Server. Für beide Ansätze hängt die Performance und Skalierbarkeit aber maßgeblich von der Größe der Sitzungsdaten ab {Penchikala 11/24/2004 #189}. Darüber hinaus ist die Implementierung beider Ansätze äußerst komplex und daher fehleranfällig. Besonders in TwoSpot, mit einer großen Anzahl an AppServern, die ständig neu gestartet und beendet werden.

Der Fokus von TwoSpot liegt auf der horizontalen Skalierbarkeit, weshalb keine speziellen Mechanismen für die Verwaltung von Sitzungsdaten implementiert wurden. Folglich müssen sich alle Request-Handler zustandslos verhalten. Die Implementierung einer Sitzungsverwaltung bleibt dem Anwendungsentwickler überlassen. Nur er kann unter Berücksichtigung der speziellen Anwendungs-Anforderungen eine sinnvolle Entscheidung bezüglich der Sitzungsverwaltung treffen. Der TwoSpot UserService-Dienst (vgl. S. 83 Portal-Anwendung) ähnelt dem OpenID-Dienst {OpenID Foundation website 3/2/2010 #131} und lässt sich daher in Kombination mit der Storage gut zur Realisierung einer zentralen Sitzungsverwaltung verwenden. Cookies in Kombination mit Prüfsummen und einer Verschlüsselung bieten eine weitere vielversprechende Möglichkeit. Generell stellen zustandslose Request-Handler aber die effizienteste Lösung dar {Henderson 2006 #55: 20}.

5.10. Anwendungsentwicklung

Eine wesentliche Anforderung an TwoSpot (vgl. S. 1, Hintergrund und Motivation) besteht darin, die Entwicklung skalierbarer Web-Anwendungen so weit wie möglich zu vereinfachen. Aus diesem Grund wird ein Dev-Kit bereitgestellt, das die lokale Entwicklung und den Test von Anwendungen ermöglicht. Dabei kommt der AppServer mit einer speziellen Konfiguration zum Einsatz. Die Komponenten: Frontend, Master und Controller werden somit nicht benötigt. Da die Plattform-Dienste im Controller implementiert sind und somit in der Entwicklungsumgebung nicht zur Verfügung stehen, greift der DevServer auf Mock-Implementierungen aller Dienste zurück. Auch die Storage-Implementierung ist im DevServer unverändert enthalten. Anstelle der HBase-Datenbank wird jedoch eine einfache Binärdatei als Speicherbackend eingesetzt. Aus Perspektive der Anwendung ist die Entwicklungsumgebung somit identisch zur Produktivumgebung.

Das Dev-Kit enthält außerdem eine Reihe von Hilfs-Skripten, die in Python realisiert sind. Sie unterstützen den Entwickler z. B. beim Anlegen einer neuen Anwendung, da sie die Verzeichnisstruktur und den Anwendungs-Deskriptor erstellen. Auch der Dev-Server lässt sich über die Skripte komfortabel starten und verwalten. Darüber hinaus übernehmen sie auch das Anwendungs-Deployment in die Produktivumgebung (vgl. S. 84).

5.11. Portal-Anwendung

Die Portal-Anwendung ist selbst eine TwoSpot-Anwendung und wird zur Verwaltung der Plattform eingesetzt. Sie ist daher als privilegierte Anwendung registriert, womit sie auf die entsprechenden Plattform-Dienste wie die Benutzer- und Anwendungsverwaltung zugreifen kann.

Eine wichtige Funktion betrifft die Verwaltung von Benutzern und Anwendungen. Neue Benutzer können sich über ein Formular unter einem eindeutigen Namen registrieren. Anschließend erhalten sie Zugriff auf eine Administrationsoberfläche. Über diese lassen sich max. drei neue Anwendungen bzw. AppIds registrieren. Außerdem können die Log-Ausgaben bestehender Anwendungen und die gespeicherten Daten in der Storage eingesehen werden. Diese Funktionen greifen auf die privilegierten Plattform-Dienste Logging und UserService zurück.

Der Login-Mechanismus wurde generisch implementiert und lässt sich daher von allen TwoSpot-Anwendungen über die nicht privilegierten Funktionen des UserService nutzen. Die Benutzerprofile selbst sind innerhalb der Portal-Anwendung gespeichert und lassen sich daher nicht von anderen Anwendungen auslesen. Über den Login-Mechanismus erhalten die Anwendungen lediglich den Login-Namen.

Der UserService enthält eine Funktion, die den Login-Namen des Benutzers zurückgibt, falls er eingeloggt ist. Andernfalls wird der Wert NULL zurückgegeben. Wenn der Benutzer nicht eingeloggt ist, kann die Anwendung eine Login-Funktion ausführen. Die Funktion erwartet zwei URLs als Parameter. Die Erste trägt die Bezeichnung SuccessURL und gibt das Weiterleitungs-Ziel nach einem erfolgreichen Login-Vorgang an. Bei einem Abbruch oder fehlerhaften Login wird der Benutzer auf die zweite URL mit der Bezeichnung FailedURL weitergeleitet. Die Login-Funktion codiert die beiden Parameter in einer neuen URL, die auf den Login-Mechanismus der Portal-Anwendung verweist. Nun leitet die Anwendung den Benutzer auf diese URL weiter.

Sobald sich der Benutzer angemeldet hat, wird eine neue Sitzungs-ID erstellt. Sie wird in einem Cookie und zentral in der Storage gespeichert. Der Cookie ist für die gesamte Domain und somit alle TwoSpot-Anwendungen gültig. Um Session Hijacking-Angriffe zu unterbinden, ist die ID allerdings nur für die Login-auslösende Anwendung gültig. Nach einem erfolgreichen Login wird der Benutzer auf die SuccessURL und damit die Anwendung weitergeleitet. Bei einem fehlgeschlagenen Login wird der Benutzer hingegen auf die FailedURL weitergeleitet. In beiden Fällen kann die Anwendung prüfen, ob der Benutzer eingeloggt ist. Im Erfolgsfall erhält sie den Login-Namen, andernfalls den Wert NULL. Der gesamte Ablauf ist in Abb. 33 für den Fall eines erfolgreichen Logins dargestellt.

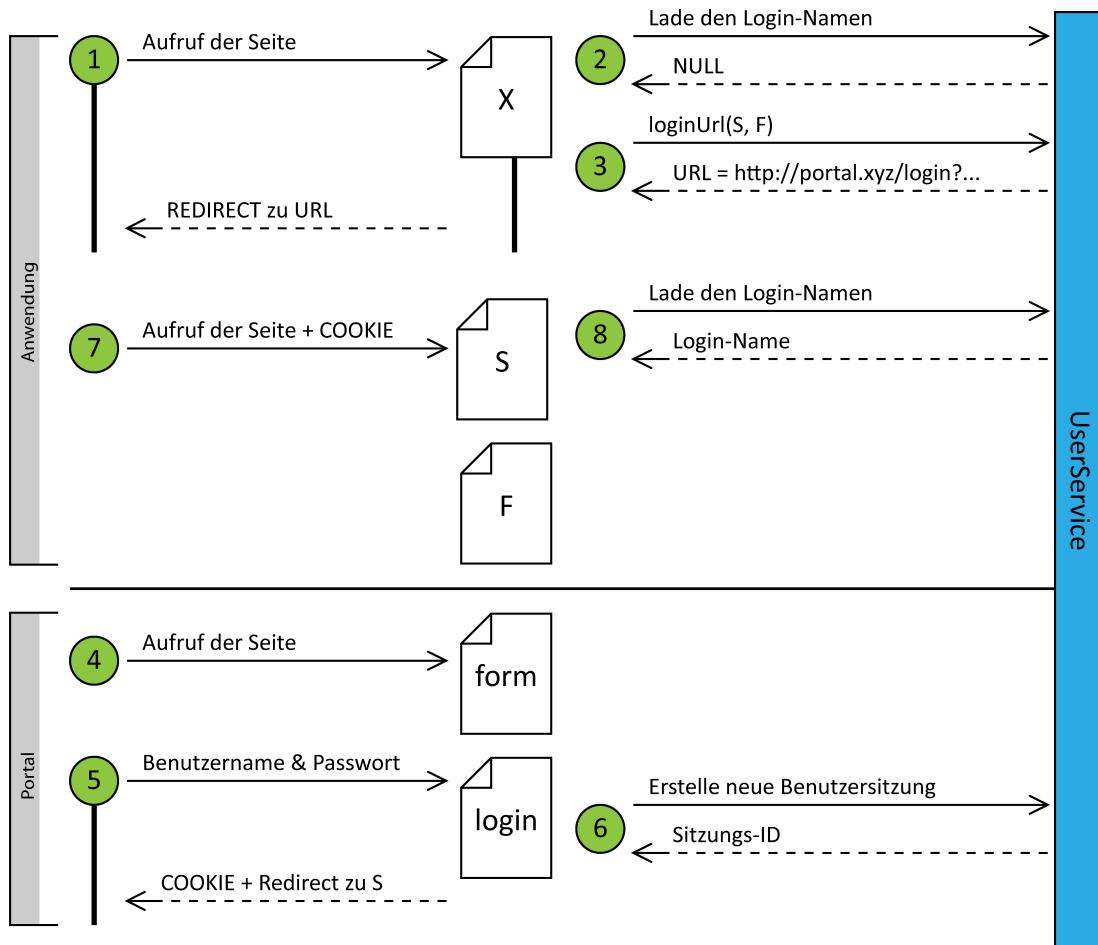


Abb. 33 Funktionsweise des UserService für einen erfolgreichen Login-Vorgang

Die Portal-Anwendung implementiert mit dem Deployment-Mechanismus eine weitere äußerst wichtige Funktion. Das Deployment einer Anwendung erfolgt grundsätzlich unter Einsatz der Hilfs-Skripte (vgl. S. 82, Anwendungsentwicklung). Beim Deployment erfragt das entsprechende Skript zunächst die Login-Daten vom Benutzer. Anschließend loggt es sich über den zuvor beschriebenen Login-Mechanismus ein und speichert den Cookie mit der Sitzungs-ID.

Nun (vgl. Abb. 34) ruft es das Deployment-Servlet auf und übergibt dabei den gespeicherten Cookie und die AppId. Das Servlet prüft zunächst, ob der Benutzer eingeloggt und im Besitz der entsprechenden Anwendungsrechte ist. Anschließend greift es auf den privilegierten Deployment-Service zurück und erstellt ein neues Upload-Token. Der Service erstellt das Token aber nicht selbst, sondern greift dabei auf den zentralen File-Server zurück (Schritt 2, 3). Nun wird das Token mit dem Response an das Skript zurückgegeben (Schritt 1, 4).

Daraufhin sendet das Skript das Anwendungs-Archiv zusammen mit dem Upload-Token an die virtuelle TwoSpot-Anwendung „deploy“. Anhand dieser AppId erkennt das Frontend den Upload und leitet die Daten direkt an den File-Server weiter. Dieser

schreibt die Daten zunächst in eine temporäre Datei, deren Name gleich dem Upload-Token ist (Schritt 5).

Nachdem das Anwendungs-Archiv vollständig übertragen wurde, ruft das Skript wieder das Deployment-Servlet auf (Schritt 6). Dabei übergibt es zusätzlich zum Cookie und der AppId noch das Upload-Token. Das Servlet prüft noch einmal, ob der Benutzer eingeloggt ist und über die notwendigen Anwendungsrechte verfügt. Anschließend bestätigt es das Deployment über den Deployment-Service, der wiederum den Upload im File-Server bestätigt (Schritt 7). Der File-Server prüft zunächst die Gültigkeit des Tokens und benennt dann die temporäre Datei unter Verwendung der AppId um. Abschließend aktualisiert der Deployment-Service das Anwendungsverzeichnis im ZooKeeper (Schritt 8). Damit werden alle Controller über den Deployment-Vorgang informiert und können die Anwendung neu starten (vgl. S. 64). Falls der File-Server das Upload-Token für ungültig erklärt, wird der gesamte Deployment-Prozess abgebrochen.

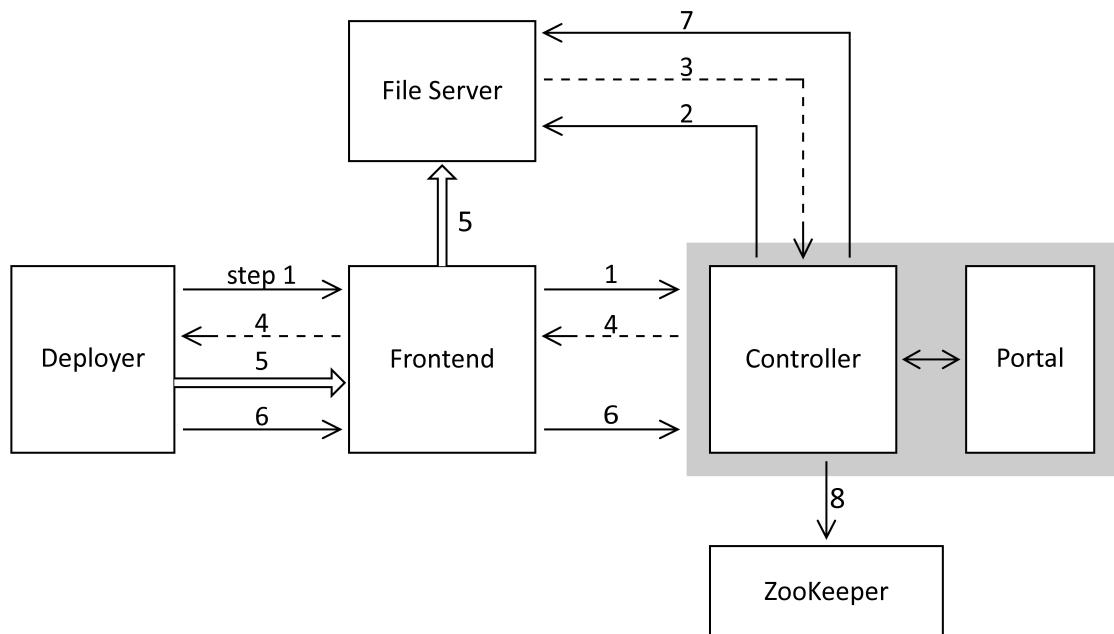


Abb. 34 Ablauf des Deployment-Vorgangs

6. Skalierungs- und Performance-Tests

Um eine Aussage bezüglich dem Verhalten von TwoSpot in größeren Deployments zu erhalten, wurde die Plattform auf 13 VMs der Rackspace-Cloud {Rackspace HOSTING #26} deployt. Davon waren 8 VMs mit 512 MB Arbeitsspeicher und einer 20 GB Festplatte ausgestattet. Jeder Server enthielt mindestens zwei 2GHz Quad Core-Prozessoren, wobei einer VM 1/16 der Gesamtleistung garantiert wurde. Die restlichen fünf VMs verfügten über 1024 MB Arbeitsspeicher, 40 GB Festplattenkapazität und konnten mindestens 1/8 der verfügbaren CPU-Leistung in Anspruch nehmen. Die einzelnen Server waren über eine Gigabit-Ethernet-Verbindung miteinander verbunden.

Bezüglich der CPU muss angemerkt werden, dass der Hypervisor CPU-Bursting unterstützt. Falls ein Server somit über genug freie CPU-Kapazität verfügt, kann diese von jeder VM gleichberechtigt genutzt werden. Unter optimalen Bedingungen kann sich damit die Leistung einer VM gegenüber der zugesicherten Leistung drastisch verbessern. Während der Tests konnte allerdings ein konstantes Leistungsniveau festgestellt werden.

HDFS und HBase wurden beide mit der Standardkonfiguration auf den fünf 1024MB-VMs betrieben. Eine VM wurde für die HDFS- und HBase-Master eingesetzt. Weiterhin wurde auf ihr der TwoSpot-Master betrieben. Für das TwoSpot-Frontend und den File-Server wurde ebenfalls eine eigene VM genutzt. Weitere fünf VMs wurden über den TwoSpot-Controller verwaltet und führten somit auch die AppServer aus. Die letzte VM stand exklusiv JMeter {Apache Software Foundation #92} zur Verfügung, das zur Belastung der TwoSpot-Anwendungen eingesetzt wurde (vgl. Abb. 35).

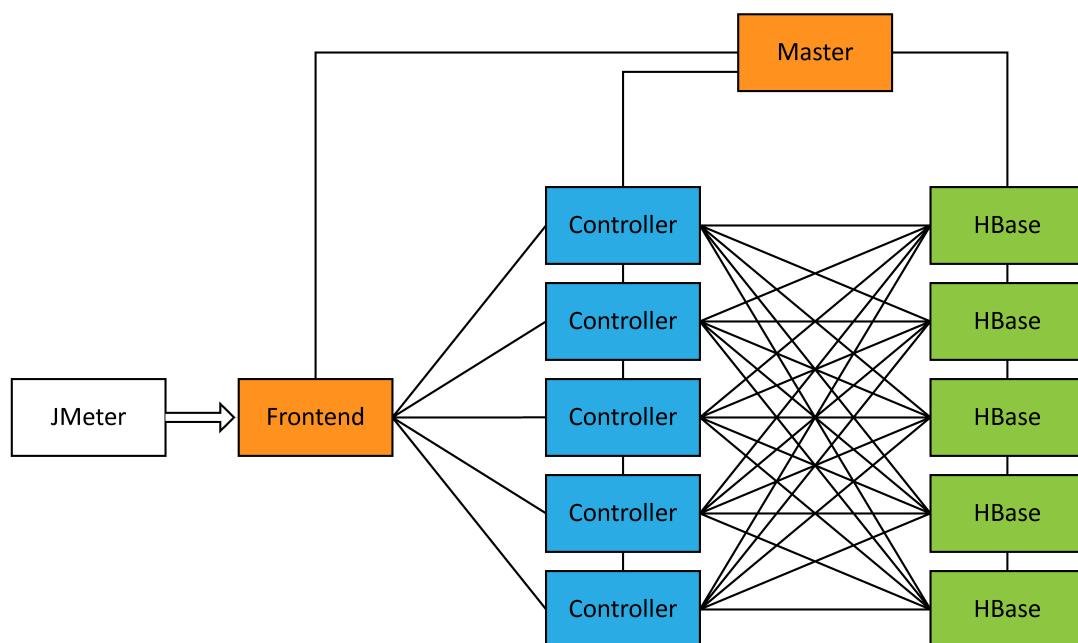


Abb. 35 Testaufbau zur Ausführung der Performance- und Skalierungstests

Um die TwoSpot-Plattform zu testen, wurden außer der Portal-Anwendung noch zwei weitere sehr einfache Java Web-Anwendungen implementiert. Beide basieren direkt auf der Servlet-API und verwenden keine zusätzlichen Frameworks oder Bibliotheken. Die GWiki-Anwendung stellt ein sehr einfaches Wiki dar. Es lassen sich neue Seiten anlegen und bestehende modifizieren. Beim Laden einer Wiki-Seite wird ihr Inhalt mithilfe regulärer Ausdrücke gefiltert. Dabei werden z. B. Wiki-Links in HTML-Links übersetzt, Zeilenumbrüche in valides HTML umgewandelt und Überschriften entsprechend eines Style-Sheets formatiert. Die dritte Testanwendung, mit dem Namen Guestbook, ist ein sehr einfaches Gästebuch. Es besteht aus einer Startseite, auf der die letzten 10 Einträge angezeigt werden. Neue Einträge lassen sich über ein einfaches HTML-Formular anlegen. Die beiden Anwendungen GWiki und Guestbook greifen lediglich auf die Storage zu, verwenden sonst aber keine Plattform-Dienste.

Da keine der Anwendungen eine merkliche Belastung der CPU simuliert, wurde zusätzlich ein Servlet zur ZIP-Kompression zufällig erstellter Byte-Arrays implementiert. Innerhalb der Portal-Anwendung wird ein 1 MB großes Array komprimiert. Da die GWiki-Anwendung das ZIP-Servlet wesentlich häufiger aufruft, wird dort nur ein 512 KB großes Array eingesetzt.

Um die Anwendungen mithilfe von JMeter zu testen, wurde im ersten Schritt zu jeder Anwendung ein Testplan erstellt. Dazu wurde eine Benutzersitzung über den JMeter-Proxy aufgezeichnet. Der entstandene Testplan wurde dann in einer Endlosschleife ausgeführt, die wiederum von mehreren Threads parallel ausgeführt wurde. Zu Testbeginn existierten lediglich drei Threads. Nach jeweils fünf Sekunden wurden drei weitere Threads gestartet. Dieser Ansatz simulierte eine stetig wachsende Anzahl von Nutzern und damit eine steigende Belastung.

Während der Testausführung wurde jede Request von JMeter in eine Log-Datei geschrieben. Nach der Testdurchführung wurde diese Log-Datei mithilfe eines Python-Skripts analysiert und zwei Metrik-Werte berechnet: 1) Die Anzahl der Requests, die innerhalb eines Fünf-Sekunden-Intervalls abgeschlossen wurden. 2) Die durchschnittliche Response-Zeit aller Requests, die innerhalb eines Fünf-Sekunden-Intervalls gestartet wurden. Jeder Test wurde insgesamt fünf Mal ausgeführt. Abschließend wurden über beide Metriken der Durchschnitt und die Standardabweichung ermittelt. Die Ergebnisse sind in Abb. 36 und Abb. 37 dargestellt.

Die Portal- und GWiki-Anwendungen verarbeiten mit einer steigenden Anzahl an Benutzern auch mehr Requests und skalieren im Rahmen der Tests näherungsweise linear. Die Guestbook-Anwendung kann hingegen nur für ca. 50 Benutzer linear skalieren. Danach erhöht sich die Anzahl der verarbeiteten Requests nicht mehr mit der steigenden Anzahl an Benutzern.

Der Grund für dieses Verhalten ist die Startseite des Gästebuchs, auf der die letzten 10 Gästebucheinträge angezeigt werden. Zum Abfragen der Einträge wird ein einfacher JDO-Query eingesetzt, der auf einen sehr kleinen Ausschnitt der IndexByProperty-Tabelle zugreift. Dieser wird nur von einem einzelnen Region-Server verwaltet, aber in jeder Request einmal abgefragt. Folglich kann die Anwendung nicht weiter skalieren, wenn dieser eine Region-Server überlastet ist. Im Fall der Guestbook-Anwendung konnte dieses Szenario sehr gut anhand der CPU-Belastung nachvollzogen werden. Problematisch sind somit die HBase-Architektur und die Storage-Implementierung. Eine Lösung dazu wäre die mehrfach redundante Speicherung der Index-Einträge, verteilt über mehrere Region-Server. Damit würde sich die Belastung auf mehrere Server verteilen. Ebenso lässt sich ein Cache zur Reduktion der Storage-Belastung einsetzen. In der Guestbook-Anwendung kann die gesamte Startseite zwischengespeichert und nur in bestimmten Zeitintervallen aktualisiert werden. Falls auf der Startseite tatsächlich die aktuellsten Gästebucheinträge benötigt werden, können neue Einträge sowohl in die Datenbank als auch in den Cache geschrieben werden. Die Startseite lässt sich dann vollständig über den Cache generieren.

Diese Situation verdeutlicht, dass eine skalierbare Architektur alleine nicht ausreichend ist, um alle Anwendungen zu skalieren. Es lassen sich auch weiterhin nicht-skalierende Anwendungen entwerfen. Allerdings wird die Implementierung tatsächlich skalierender Anwendungen erleichtert.

Bei einer genauen Betrachtung der Graphen fallen außerdem zufällige Leistungseinbrüche auf. Diese entstehen, sobald der Anwendung ein weiterer AppServer zur Verfügung gestellt wird. In diesem Fall leitet das Frontend einen Großteil der eingehenden Requests umgehend auf den neuen, noch unbelasteten AppServer weiter. Dieser muss vom Controller allerdings erst gestartet werden. Somit werden alle eingehenden Requests pausiert, bis der AppServer verfügbar ist. Das Verhalten wird besonders beim Starten vom zweiten AppServer deutlich, da das Frontend in diesem Fall die Hälfte aller Requests auf den zweiten Controller weiterleitet. Dies führt zu einem besonders

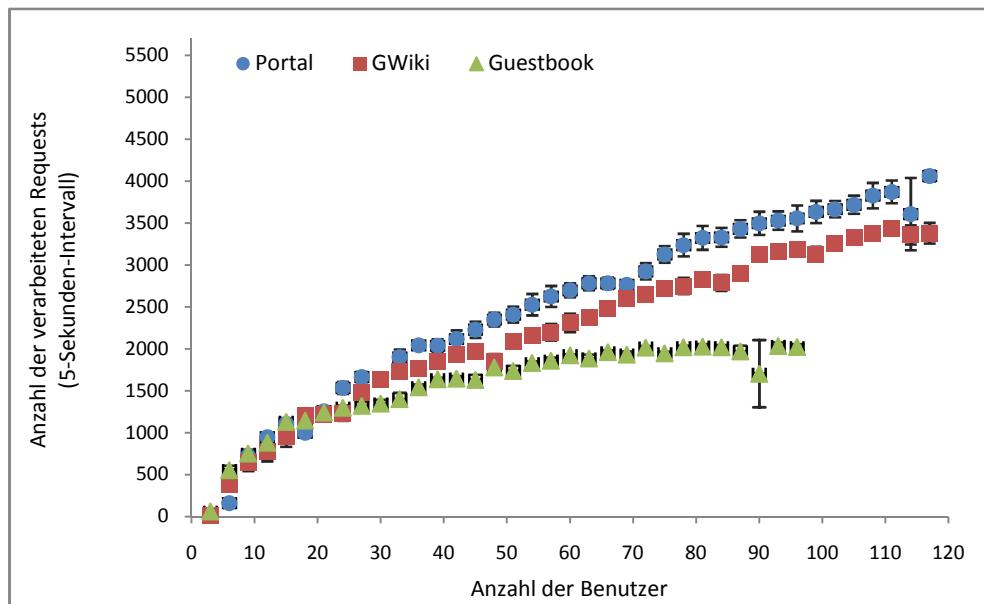


Abb. 36 Anzahl der verarbeiteten Requests unter einer steigenden Belastung

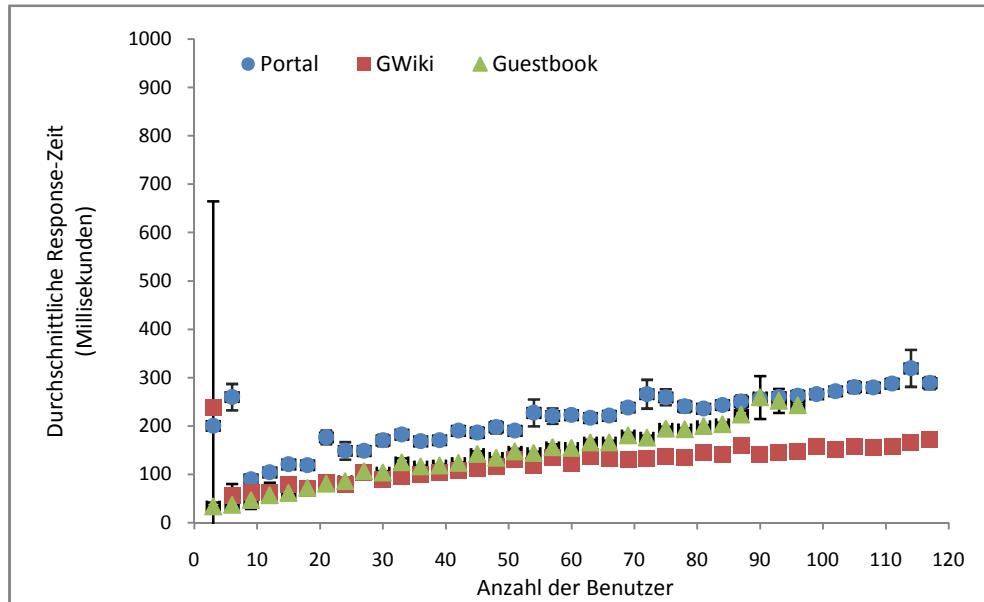


Abb. 37 Verhalten der Response-Zeit unter einer steigenden Belastung

hohen Leistungseinbruch.

Eine weitere Ursache zufälliger Leistungseinbrüche sind Region-Splits der HBase-Datenbank. Alle Anwendungen fügen kontinuierlich Daten zur Datenbank hinzu. Die Tabellen-Regionen wachsen somit ständig an. Sobald eine Region eine bestimmte Maximalgröße erreicht hat, wird sie zweigeteilt (vgl. S. 45, Bigtable, Hypertable und HBase). Anschließend werden die beiden Teile einem Region-Server zugewiesen. Vor einem Region-Split wird die entsprechende Region aber deaktiviert, womit ihre Daten nicht mehr erreichbar sind. Für die Dauer eines Region-Splits, der mehrere Sekunden in Anspruch nehmen kann, können die HBase-Clients bzw. AppServer nicht auf die Daten zugreifen. Sie warten daher, bis die Daten wieder verfügbar sind. Dieses Verhalten führt zu einem drastischen Leistungseinbruch, der aber nur selten auftritt. Die Problematik war in der Testumgebung verschärft, da lediglich eine Anwendung auf der Plattform betrieben wurde und nur fünf HBase-Server zur Verfügung standen.

Bei der Bewertung der Skalierungs- und Performance-Tests ist zu berücksichtigen, dass TwoSpot lediglich eine prototypische Implementierung darstellt und daher noch keine Performance-Optimierungen enthält. Trotzdem konnten bereits vielversprechende Ergebnisse erzielt werden.

7. Ausblick

Im Rahmen der Arbeit wurde das Fundament der TwoSpot-Plattform gelegt. Darauf aufbauend lassen sich nun weitere Optimierungen und Erweiterungen an den Algorithmen und der Architektur vornehmen. Nachfolgend sind einige Zielsetzungen für die mittelfristige Entwicklung aufgeführt.

Die Storage-Implementierung orientiert sich stark am Design des Google App Engine Datastores und kann bisher lediglich als „Proof-of-Concept“ betrachtet werden. Zusammengesetzte Storage-Queries mit mehr als einer Bedingung müssen aktuell im Arbeitsspeicher ausgeführt werden. Über zusammengesetzte Indizes, die mehr als eine Objekt-Eigenschaft erfassen, lassen sich der Speicherverbrauch und die Performance derartiger Queries noch maßgeblich verbessern.

Das Einfügen und Aktualisieren von Objekten führt zur Erstellung oder Änderung mehrerer Einträge in verschiedenen HBase-Tabellen. Diese Änderungen sind vollkommen unabhängig zueinander. Folglich lassen sie sich in mehreren Threads parallel ausführen. Ein multithreaded Ansatz verspricht daher eine maßgebliche Performanceverbesserung der Insert- und Update-Operationen. Darüber hinaus bietet die Storage bislang keine Unterstützung für Transaktionen. Ähnlich zum Ansatz im App Engine Datastore {Barrett 12/15/2009 #78} lassen sich allerdings stark eingeschränkte Transaktionen realisieren.

In der aktuellen Implementierung wird die Storage im Controller ausgeführt. Durch die Auslagerung in einen eigenen Prozess kann die Belastung auf den Controller wesentlich reduziert und auf spezielle Storage-Server verschoben werden.

Aber nicht nur die Storage-Implementierung bietet Ansatzpunkte für eine Optimierung. Wie die Tests aufgezeigt haben, besteht in den Load-Balancing-Mechanismen weiteres Optimierungspotenzial. Das Frontend berücksichtigt z. B. die Größe und Verarbeitungsdauer der Requests nicht. Darüber hinaus treten starke Performanceeinbrüche beim Starten neuer AppServer auf, da ihre Startzeit nicht berücksichtigt wird.

Auch im Master lassen sich die verfügbaren Management-Daten besser ausnutzen. Ein profilgestütztes Load-Balancing verspricht dabei erhebliche Vorteile. Dabei erfasst die Plattform zu jeder Anwendung Performance- und Lastdaten. Daraus wird dann ein Lastprofil erstellt, das sich für eine intelligente Skalierung nutzen lässt. Beispielsweise lassen sich damit typische Lastanstiege zur Mittagszeit vorhersehen. Für die Verarbeitung der dabei entstehenden Datenmengen, stehen mit dem HDFS und dem Hadoop Map-Reduce-Framework bereits leistungsfähige Mechanismen zur Verfügung.

Eine weitere wichtige Erweiterung ist die Bereitstellung eines Memcache-Clusters {Memcached #39} als Plattform-Dienst. In diesem lassen sich z. B. ganze Web-Seiten zwischenspeichern und damit äußerst effizient ausliefern. Ein weiterer Anwendungsfall

ist die Zwischenspeicherung zeitintensiver Storage-Queries, womit sich die Belastung auf die Storage reduzieren lässt.

Die TwoSpot-Architektur ist momentan für die Verarbeitung dynamischer Requests ausgelegt. Statische Requests wie z. B. Bild-, JavaScript- oder CSS-Dateien müssen ebenfalls über die AppServer ausgeliefert werden. Abhilfe kann ein Dienst zur Verarbeitung statischer Requests bzw. Ressourcen schaffen. Dazu muss das Frontend die Requests analysieren und auf diesen speziellen Dienst umleiten. Die Controller und AppServer werden somit keiner unnötigen Belastung mehr ausgesetzt.

Die TwoSpot-Architektur wurde für den Betrieb von Anwendungen in verschiedenen Programmiersprachen entworfen (vgl. S. 60, AppServer). Die ursprüngliche Planung sah vor, verschiedene Programmiersprachen auf Basis der Java VM auszuführen. Während der Implementierung des Python-Stacks wurde allerdings festgestellt, dass sich die Startgeschwindigkeit durch das Laden der zusätzlichen Sprach-Bibliotheken drastisch verlängert und sich damit nicht mehr in einem akzeptablen Rahmen bewegt. Ein Lösungsansatz besteht im Vorstarten von AppServern ohne eine Anwendung. Bei Bedarf werden sie dann mit einer Anwendung bestückt. Somit lässt sich die Zeit zum Starten des Servers einsparen. Eine andere Möglichkeit ist die Implementierung alternativer AppServer in der entsprechenden Programmiersprache. Zum Beispiel kann ein AppServer auf Basis der Python VM realisiert werden. Oftmals weisen die Implementierungen einer Programmiersprache auf Basis der Java VM Inkompatibilitäten mit der Referenzimplementierung auf. Dieser Ansatz greift auf die Referenzimplementierung zurück und ist somit vollständig kompatibel zu bestehenden Software-Bibliotheken.

8. Abschluss

Ziel dieser Arbeit war der Entwurf und die Realisierung einer neuen Cloud-Plattform, die den Anforderungen der Fojobo Web-Anwendung genügt. Dazu wurde zunächst das Umfeld des Cloud Computing beleuchtet und ein Klassifikationsschema für Cloud-Provider beschrieben. Sie lassen sich demnach in drei Kategorien einteilen: IaaS, PaaS und SaaS. Von besonderem Interesse sind dabei PaaS-Provider, da sie die Entwicklung und den Betrieb skalierbarer Web-Anwendungen maßgeblich vereinfachen. Unter Einsatz dieser Klassifikation wurde dann eine Reihe bekannter PaaS-Provider, wie die Google App Engine, Heroku, Joyent Smart und Microsoft Windows Azure evaluiert. Der Analyseschwerpunkt wurde dabei auf die Plattform-Architektur und die Mechanismen zur Datenspeicherung gelegt. Dabei hat sich gezeigt, dass alle Plattformen eine horizontale Skalierung der Anwendungen unterstützen. Allerdings unterscheiden sich die Plattformen sehr stark in ihrer Architektur. Wesentliche Unterschiede betreffen das Vorgehen zur Anwendungsisolation, wobei zwei grundlegende Ansätze unterschieden werden können. Zum einen kommt die Virtualisierungs-Technologie zum Einsatz, womit jede Anwendung exklusiv in mehreren VMs ausgeführt wird. Beim zweiten Ansatz werden mehrere Anwendungen gleichzeitig auf einem Server oder einer VM ausgeführt. Die Ausführung erfolgt in einem Execution Environment, das auch Mechanismen zur Anwendungsisolation enthält. Zur Datenspeicherung greifen einige Plattformen auf klassische relationale Datenbanken zurück, die in einer vertikalen Weise skaliert werden. Größtenteils wird allerdings ein horizontal skalierbares Speichersystem eingesetzt. Meist handelt es sich dabei um proprietäre Entwicklungen oder offene NoSQL-Speicherlösungen.

Auf Basis dieser Analyseergebnisse wurde eine IaaS-Ebene mit einem Physical Resource Set als Fundament der TwoSpot-Plattform gewählt. Dabei erfolgt die Anwendungsisolation über ein Execution Environment. Mit diesem Ansatz lassen sich die Cloud-Anwendungen äußerst ressourcen- und kostensparend betreiben. Darüber hinaus ermöglicht das Execution Environment eine effiziente Anpassung an kurzfristige Veränderungen der Anwendungsbelastung.

Im Hinblick auf die Implementierung wurde anschließend eine Reihe von Kerntechnologien für das Execution Environment, die Datenspeicherung und das verteilte Dateisystem evaluiert. Die Wahl fiel schließlich auf den Jetty-Server, der sich durch seine modulare und klare Architektur vom Apache Tomcat-Server abhebt. Als Datenbank kommt HBase mit dem HDFS-Dateisystem zum Einsatz. Die MongoDB- und Cassandra-Datenbanken stellen ebenfalls vielversprechende Lösungen dar. Im Vergleich zu Cassandra ist die Datenorganisation von HBase allerdings besser für die Bedürfnisse von TwoSpot geeignet. Darüber hinaus erschienen Cassandra und der Auto-Sharding-Mechanismus der MongoDB noch nicht für den produktiven Betrieb geeignet. Als verteiltes Dateisystem kommt das HDFS zum Einsatz. Das MogileFS wurde aufgrund der dateibasierten und damit ineffizienten Replikationsmechanismen nicht gewählt. Weitere

Kritikpunkte betrafen die Datenintegrität und die fehlende Berücksichtigung von Server-Racks in den Replikationsmechanismen.

Für den Betrieb der TwoSpot-Anwendungen wurde ein neues Execution Environment auf Basis des Jetty-Servers entworfen. Seine Architektur unterstützt den Betrieb mehrerer Programmiersprachen auf Basis der Java VM. Zur Anwendungsisolation wird auf die Sicherheitsmechanismen der Java VM zurückgegriffen, womit sich eine Sandbox-Umgebung ergibt. Darüber hinaus wird jede Anwendungsinstantz in einem eigenen Execution Environment und damit Prozess ausgeführt. Um der horizontalen Skalierbarkeit Rechnung zu tragen, lässt sich eine Anwendung auf mehreren Servern parallel ausführen. Die Belastung wird über einen Reverse-Proxy gleichmäßig auf alle Anwendungsinstanzen verteilt. Die Anzahl der Instanzen wird automatisch durch die Plattform geregelt. Dazu sammelt sie in regelmäßigen Zeitabständen Management-Daten über die Server und die Anwendungen.

Zur Reduktion des Vendor lock-ins erfolgt die persistente Datenspeicherung über eine standardisierte JDO-Schnittstelle. Aus Gründen der Sicherheit und Performance wurde aber zwischen der JDO-Schnittstelle und der HBase-Datenbank eine neue Speicherschicht namens Storage implementiert. Sie verhält sich ähnlich einer dokumentenorientierten Datenbank und benötigt somit keine Schema-Migrationen.

Im Anschluss an die Implementierung wurde die Plattform verschiedenen Tests unterzogen. Dazu wurde sie in einer produktiven IaaS-Umgebung mit 13 Servern installiert. Anschließend wurde ihre Skalierbarkeit mithilfe verschiedener Test-Anwendungen überprüft. Bereits ohne umfangreiche Optimierungen ließen sich vielversprechende Ergebnisse erzielen. Auf dieser Grundlage sollen nun weitere Technologien eingebunden und Optimierungen am bestehenden Design vorgenommen werden.

9. Abkürzungsverzeichnis

Abkürzung	Beschreibung
AC	Application Controller (AppScale)
API	Application Programming Interface
AS	Application Server (AppScale)
AWS	Amazon Web Services
BIS	Basic Infrastructure Service
CAP	Consistency, Availability, Partition Tolerance
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DAV	Distributed Authoring and Versioning
DBM	Database Master (AppScale)
DBS	Database Slave (AppScale)
DNS	Domain Name System
DOS	Denial Of Service
GFS	Google File System
GQL	Google Query Language
GUID	Global Unique Identifier
HDFS	Hadoop Distributed File System
HIS	High Infrastructure Service
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
IIS	Internet Information Services (Microsoft)
IP	Internet Protocol
JDO	Java Data Objects
JMX	Java Management Extension
JPA	Java Persistence API
JSON	JavaScript Object Notation
KFS	Kosmos File System
PRS	Physical Resource Set
RDB	Relational Database
REST	Representational State Transfer
RMI	Remote Method Invocation
SPOF	Single Point Of Failure
SQL	Structured Query Language
SSH	Secure Shell (Protocol)
TTL	Time To Live
VM	Virtual Machine
VRS	Virtual Resource Set
WAR	(Java) Web Archive
WSGI	Web Server Gateway Interface
YAML	Yet Another Markup Language

10. Abbildungsverzeichnis

Abb. 1 Architektur der Google App Engine {Levi 28.05.2009 #12: 20}	13
Abb. 2 Auslastung der App Engine-Anwendung „Open for Questions“ Quelle: {Levi 28.05.2009 #12: 51}	17
Abb. 3 Architektur der Joyent Smart-Plattform.....	18
Abb. 4 Software-Stack eines Heroku Dyno-Prozesses	22
Abb. 5 Architektur der Heroku-Plattform.....	23
Abb. 6 Aufbau des Windows Azure Table-Storage	27
Abb. 7 Architektur der AppScale-Plattform mit mehreren VMs	31
Abb. 8 Master/Slave-Betrieb einer relationalen Datenbank	40
Abb. 9 Multi-Master-Replikation einer relationalen Datenbank	40
Abb. 10 Sharding (vertikale Partitionierung) mit zwei Datenbankservern.....	41
Abb. 11 MongoDB Server-Architektur beim Einsatz der Auto-Sharding-Funktion	44
Abb. 12 Eine HBase-Tabelle mit einer Column family, mehreren Spalten und Zellen-Versionen	45
Abb. 13 Eine Tabelle wird in mehrere Regionen aufgeteilt. Schreibzugriffe erfolgen direkt in den Regionen.	46
Abb. 14 Dreistufiger Lookup der Adresse einer User-Region bzw. des Region-Servers.....	47
Abb. 15 Ringförmiger Namensraum, der durch mehrere Token zerteilt wird. Jeder Row-Key wird über eine konsistente Hashfunktion auf den Namensraum abgebildet.	49
Abb. 16 Aufbau des HDFS-Dateisystems mit zwei Chunk-Servern	53
Abb. 17 Ablauf beim Schreiben einer Datei in das HDFS	54
Abb. 18 Ausfallsichere Architektur des MogileFS-Dateisystems.....	56
Abb. 19 Kommunikation zwischen den Komponenten der TwoSpot-Plattform.....	59
Abb. 20 Der AppServer Software-Stack unterstützt verschiedene Programmiersprachen auf Basis der Java VM	61
Abb. 21 Lifecycle zur Verwaltung eines AppServers.....	63
Abb. 22 Aufbau des Controller-Cache im Frontend	67
Abb. 23 Anwendung greift über einen Proxy auf einen Plattform-Dienst zu	70
Abb. 24 Die Anwendung greift über eine JDO-Schnittstelle auf die Storage zu. Das Speicher-Backend der Storage ist austauschbar	72
Abb. 25 Aufbau einer ProtoBuf-Message zum Speichern eines persistenten Objekts	73
Abb. 26 Sequenz zum Generieren der Row-Keys	74

Abb. 27 Gleichmäßige Verteilung der Objekte über alle Region-Server mithilfe einer Zufallskomponente im Row-Key.....	75
Abb. 28 Unterschied zwischen einer normalen Sequenz und einem Sharded Counter.....	76
Abb. 29 Einfügen eines Objekts in die Objects- und IndexByProperty-Tabellen.....	77
Abb. 30 Interner Aufbau der IndexByProperty-Tabelle	77
Abb. 31 Ein Query mit einem Gleich-Operator wird in einen Table-Scan übersetzt.....	78
Abb. 32 Verzeichnisstruktur im ZooKeeper mit einem Anwendungsverzeichnis.....	80
Abb. 33 Funktionsweise des UserService für einen erfolgreichen Login-Vorgang.....	84
Abb. 34 Ablauf des Deployment-Vorgangs.....	85
Abb. 35 Testaufbau zur Ausführung der Performance- und Skalierungstests	86
Abb. 36 Anzahl der verarbeiteten Requests unter einer steigenden Belastung	89
Abb. 37 Verhalten der Response-Zeit unter einer steigenden Belastung	89

11. Literaturverzeichnis

Citavi...

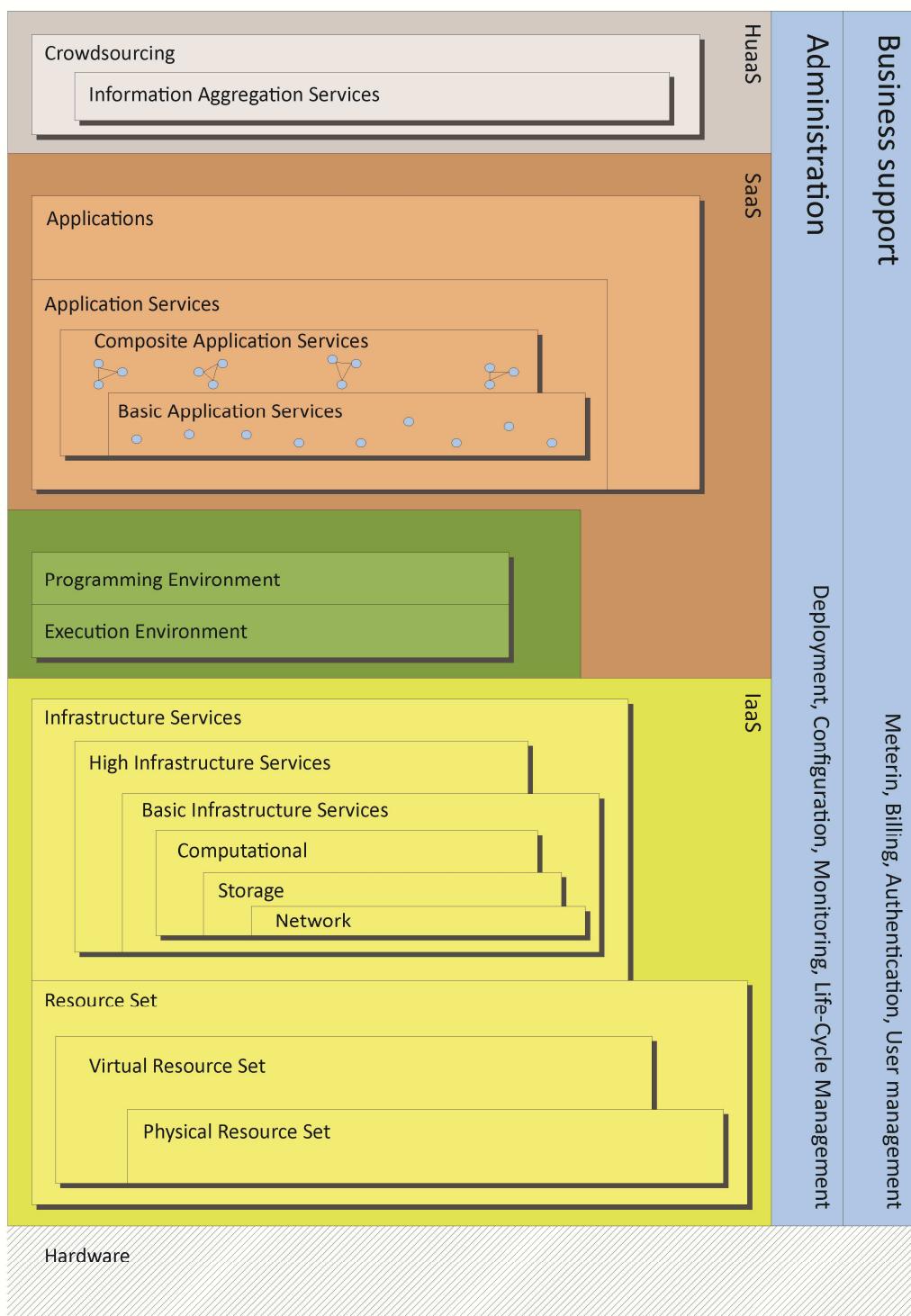
1. Anhang Autorisierung bei Flickr

Autorisierung wenn eine Benutzeranwendung über die Flickr-API auf das persönliche Benutzerprofil zugreift. (Quelle: Screenshot der Flickr Web-Seite, Der Benutzername wurde durch schwarze Balken unkenntlich gemacht, ({Flickr #102}))

The screenshot shows the Flickr login interface. At the top, there's a navigation bar with links for 'Startseite', 'Sie', 'Organisieren', 'Kontakte', 'Gruppen', 'Entdecken', and a search bar. Below the navigation is a large yellow header box containing the text: 'flickr® von YAHOO!'. To the left of the header is a small profile picture of a person holding a camera. The main content area has a black background with white text. It starts with 'Hallo [REDACTED]' where '[REDACTED]' is a black redaction box. Below this is a large yellow box containing the text: 'flickrDrive möchte eine Verknüpfung zu Ihrem Flickr Account herstellen.' Underneath this, there are two columns of text. The left column says: 'Um sicherzustellen, dass es sich hierbei um eine echte Anfrage handelt, wählen Sie bitte eine der folgenden Optionen aus.' The right column says: 'Was ist hier los? Flickr unterstützt die Entwicklung von eigenen Programmen, aber diese Programme müssen im Ihren Account freigeschalten werden.' The left column continues: 'Falls Sie zu dieser Seite gelangt sind, weil Sie unabhängig von flickrDrive, auf einem Link in einer E-Mail, oder einer IM in Twitter oder auf einer Webseite geklickt haben, klicken Sie hier.' The right column continues: 'Möchten Sie mehr erfahren? Ausführliche Informationen finden Sie auf der Seite Flickr Services.' At the bottom of the yellow box are two blue buttons: 'WEITER' on the left and 'ZURÜCK' on the right. The footer of the page contains a section titled 'flickrDrive bietet folgende Beschreibung:' followed by a short English description of the service. At the very bottom, there's a copyright notice: 'Copyright © 2013 Yahoo! Inc. Alle Rechte vorbehalten.'

2. Anhang Klassifikationsschema

Die folgende Abbildung zeigt das Klassifikationsschema von Alexander Lenk et al.
(Quelle: What's Inside the Cloud?, ({Lenk 2009 #89: 26}))



3. Anhang USB-Speicherstick

Der beiliegende Datenträger weist folgenden Inhalt auf:

- **source:** Alle im Rahmen dieser Arbeit entwickelten Programme.
- **tests:** Die im Kapitel: Skalierungs- und Performance-Tests verwendeten JMeter-Testpläne.
- **documents/papers:** Die in dieser Arbeit verwendeten Papers.
- **documents/presentations:** Die in dieser Arbeit verwendeten Präsentationen.
- **documents/twospot.xps, twospot.pdf:** Digitale Fassung dieser Arbeit.