Entwicklung einer effizienten Revalidierungsmethodik für umfangreiche Caches im Kontext von leistungsoptimierten Webapplikationen

Abschlussarbeit

von

Florian Schutz
7095337

im

Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften

Gutachter

Prof. Dr. Norbert Ritter

Dr. Wolfram Wingenrath

Universität Hamburg

Abgabe 07.03.2020

Florian Schutz

7095337

Hans-Henny-Jahnn-Weg 22, 22085 Hamburg

f@florian-schutz.de

Inhalt

Abbildungsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
AWS	Amazon Web Services
Baas	Backend-as-a-Service
CDN	Content Delivery Network
HTTP	Hyper Text Transfer Protocol
RTT	Round Trip Time
TCP	Transmission Control Protocol
TTL	Time to Live
VM	Virtual Machine

1 Einleitung

Das Internet beschränkt sich nicht nur auf die Darstellung persönlicher Profile in sozialen Netzwerken. Vielmehr kommen komplexere Webapplikationen zum Einsatz, die beispielsweise für die Verwaltung von Zahlungen oder ganzer Geschäftslogiken verwendet werden können. Unabhängig vom Einsatzgebiet von Webseiten empfinden Anwender lange Ladezeiten als störend. Mit Hilfe des Einsatzes effizienter Caching Technologien können Seitenaufrufe beschleunigt werden. Im schlechtesten Fall bewirken diese Technologien jedoch die Auslieferung veralteter Daten, wenn diese Informationen im Cache nicht zeitnah aktualisiert werden. Auch wenn die Anzeige eines alten Profilbilds im Falle eines sozialen Netzwerkes nicht optimal ist, könnte es dem Betreiber eines Online-Shops Probleme bereiten, wenn ein Artikel zu einem alten Preis verkauft wird. Betreiber von Webanwendungen jeglicher Art haben folglich zum einen das Ziel die Auslieferung ihrer Applikation zu beschleunigen, wollen andererseits aber nicht die Aktualität gefährden.

Baqend betitelt ein erfolgreiches Startup, das aus gewonnenen Erkenntnissen rund um Caching von Webanwendungen in Form von Abschlussarbeiten und Dissertationen ein Produkt kreiert hat, welches die effiziente Beschleunigung von Ladezeiten verspricht. Mit SPEED KIT sollen Betreiber mit nur minimaler Konfiguration eine optimierte Version ihrer Anwendung erhalten. Im Hintergrund übernimmt der Anwendungsserver ORESTES die Durchführung dieser Leistungssteigerung. Unter anderem ergänzt der Einsatz von SPEED KIT einen bestehenden Webserver um ein verteiltes Content Delivery Network (CDN), das strategische Knotenpunkte physisch näher zu

anfragendenen Nutzern bringt. Diese Knotenpunkte speichern ein Abbild des eigentlichen Webservers. Jedes Abbild ist eine Ausprägung eines Caches, also die Zwischenspeicherung von Daten. Er dient dazu einen ersten Aufruf der Anwendung zu beschleunigen. Auf der Seite des Browsers – auf dem Endgerät des Nutzers – verwaltet SPEED KIT einen weiteren Cache, der jeden folgenden Aufruf optimieren soll. Jedes Mal, wenn eine Ressource – etwas ein Bild – auf dem Webserver des Betreibers aktualisiert wird, muss es auch in allen Caches des verteilten Systems erneuert werden, damit die bereits verdeutlichte Anzeige alter Daten vermieden wird. Obwohl Baqend effiziente Mechanismen implementieren konnte, die die Verwaltung der Caches übernimmt, stellt es dennoch eine Herausforderung dar, den Zeitpunkt des Ablaufs des Caches festzustellen. Im Idealfall informiert der Anwendungsbetreiber den ORESTES Server über die Aktualisierung einer Ressource. Die Praxis zeigt jedoch, dass dies in der Regel nicht genutzt wird. Der einzig verbleibende Weg den Verfall einer Ressource im Cache festzustellen, ist die regelmäßige Revalidierung. Dafür wird die Ressource manuell erneut vom Server abgefragt und mit der bisherigen im Cache gespeicherten Version verglichen.

Zusammengefasst ist es für Betreiber also wichtig, ihre Informationen schnellstmöglich an Nutzer ihrer Webanwendung auszuliefern. Gleichzeitig gefährden bestehende Caching Ansätze jedoch die Aktualität dieser Daten.

Problemdefinition

Eine moderne Webanwendung, vor allem umfangreiche Online-Shops, setzen sich oftmals aus mehreren hunderttausend Ressourcen zusammen. Damit eine aus dem Cache angezeigte Anwendung aktuelle Ergebnisse ausspielt, müssen alle Ressourcen also regelmäßig revalidiert werden. Im Regelfall geschieht dieser erhebliche Arbeitsaufwand alle 24 Stunden und wird von ORESTES durchgeführt. Gerade weil Baqend durch stetigen Wachstum und der regelmäßigen Gewinnung neuer Kunden einen beachtli-

chen Erfolg beweisen konnten, werden manche Teildiziplinen von ORESTES dem großen Aufwandsvolumen nicht mehr gerecht. Die beschriebenen umfangreichen Revalidierungsaufgaben können sich je nach Webanwendung über mehrere Stunden erstrecken. In Einzelfällen beginnt eine Revalidierung sogar kurz vor dem Abschluss der vorherigen. Die vorgestellte Problematik ist vor allem für Betreiber relevant, die mit ihren Webanwendungen Umsatz generieren. Unter Umständen nehmen Online-Shops mehrmals täglich Preisänderungen vor, die entsprechend aktuell an die Nutzer ausgestrahlt werden müssen. Für diese spezifischen Anwendungsfälle reicht eine tägliche Revalidierung oft nicht aus. Es kann abgeschätzt werden, welche Ressourcen häufiger Aktualität verlieren und dementsprechend mehrmals revalidiert werden müssen. Es bleibt jedoch immer bei einer Schätzung. Nur mit regelmäßiger umfangreicher Revalidierung kann sichergestellt werden, dass eine durch SPEED KIT optimierte Webanwendung zu jeder Zeit korrekte und aktuelle Inhalte an die Nutzer ausspielt.

Zielsetzung der Arbeit

Die vorliegende Arbeit soll die Cache-Revalidierung von ORESTES fokussieren. Anknüpfend an die Problemdefinition sollen Limitationen im bisherigen Mechanismus aufgedeckt werden, die die Leistungsfähigkeit der Revalidierungsaufgabe einschränkt. Aufbauend darauf sollen mögliche Problemlösungen definiert werden, die zur Eliminierung der Schwachstellen eingesetzt werden könnten. Mit der (mindestens) prototypischen Implementierung der erreichten Verbesserung kann ein möglicher Erfolg in einer praktischen Umsetzung evaluiert werden. Die Forschungsfragen dieser Ausarbeitung lassen sich daher wie folgt formulieren:

1) *Welche Limitationen schränken die Leistung des aktuellen Cache-Revalidierungs-Mechanismus von ORESTES ein und welche Aspekte funktionieren bereits gut?*

2) *Kann ein Prototyp die erkannten Schwachstellen eliminieren?*

Methodik der Ausarbeitung

Zur Feststellung der Limitationen in der aktuellen Revalidierungslogik wird die bestehende Implementierung in ORESTES analysiert. Nach dem Verständnis der konkreten Umsetzung können Problemfelder abgeleitet werden, die auf Einschränkungen schließen lassen. Gleichzeitig werden die Aspekte der bisherigen Umsetzung herausgestellt, die bereits gut funktionieren. Gefordert ist, eine neue mögliche Implementierung vorzustellen. Diese soll alle bereits solide bearbeiteten Aspekte weiterhin erfüllen, aber gleichzeitig die abgeleiteten Problemlösungen umsetzen. Dazu lassen sich Anforderungen definieren, die im gesamten Implementierungsprozess verwendet werden. In einer anschließenden Evaluation soll bewiesen werden, wie die vorgestellte Lösung die geforderten Anforderungen erfüllt und die Cache-Revalidierung – messbar in definierten Kennzahlen – verbessern kann.

Aufbau der Arbeit

In Kapitel 2 werden Grundkenntnisse in Bezug auf Webanwendungen mit Fokus auf Datenübertragung und Ladezeitverbesserung vermittelt sowie die Möglichkeiten des SPEED KITS, diese zu verbessern. Das Ziel ist das Verständnis der Cache-Revalidierung im konkreten Kontext von Baqends eingesetzten Systemen. Kapitel 3 dient der Analyse der aktuellen Implementierung. Weiterhin werden die Anforderungen an eine neue Lösung definiert. Diese soll in den Kapiteln 4 und 5 erarbeitet werden. Im darauffolgenden Kapitel 6 soll die vorgestellte Neuentwicklung in einer Evaluation beweisen, dass alle geforderten Anforderungen korrekt umgesetzt werden. In Kapitel 7 werden die Ergebnisse in den Kontext von Baqend übertragen. Themen, die über den Umfang dieser Ausarbeitung hinaus gehen, werden im Ausblick behandelt. Kapitel 8 schließt die Bearbeitung der Thematik mit der Beantwortung der Forschungsfragen ab.

2 Grundlagen

Moderne Webanwendungen sind betriebssystemunabhängige Applikationen, die auf dem Client-Server-Modell aufbauen. Die beiden Akteure in diesem System sind dabei räumlich getrennt. Ruft ein Anwender eine Applikation über einen Internetbrowser auf, werden alle dafür benötigten Daten vom Server an den Client übertragen. Internetseiten können ebenso als Webanwendung betitelt werden wie technisch aufwendigere Programme. Dies können beispielsweise Textbearbeitungsdienste sein, die im Browser aufrufbar und nutzbar sind. Die Kommunikation in diesem verteilten System findet dabei in der Regel über das Internet statt. Als standardisiertes Regelwerk für den Inhalt dient das Hyper Text Transfer Protocol (HTTP). Dieses steuert nicht nur den Ablauf der Kommunikation, sondern stellt Entwicklern auch einen strengen Handlungsrahmen, der bei der Implementierung von Webanwendungen eingehalten werden muss. Die Kommunikation auf technischer Sicht wird mit dem Transmission Control Protocol (TCP) ermöglicht. Dies gibt beispielsweise vor, dass Daten für die Übertragung in kleine Pakete zerlegt werden müssen, damit sie in einem verteilten Netzwerk, wie dem Internet, effizienter transportiert werden können.

Aufgrund der physischen Entfernung von Endanwender und Server können Verzögerungen beim Aufruf einer Anwendung in Form von Ladezeiten entstehen. Diese Verzögerung, auch Latenz genannt, bemisst den Zeitintervall, der den Erhalt eines Ereignisses – hier die Anzeige der Anwendung – in die Zukunft verschiebt.[1] Verschiedene technische Aspekte haben einen Einfluss auf das Ausmaß dieser Verzögerung.

[1] Flach et al. (2013).

Im Folgenden werden verschiedene Kriterien, die die Ladezeit einer Anwendung beeinflussen, kurz erläutert. Nach dieser Herleitung wird beschrieben wie mit Caching die Ladezeit verbessert werden kann. Aufbauend darauf wird das von der Firma Baqend bereitgestellte SPEED KIT vorgestellt, das Betreibern die Ladezeitoptimierung vereinfachen soll.

2.1 Kriterien für Ladezeiten von Webanwendungen

Der Aufbau einer Anwendung verteilt sich in der Regel auf viele unterschiedliche Dateien. Manche Dateien beschreiben die Gestaltung, andere die Funktionalitäten und wiederum andere die medialen Inhalte. Alle Dateien müssen beim Aufruf vom Server an den Client übertragen werden. Die eigentlichen Daten werden zwar über HTTP übertragen. Bevor der Client jedoch eine solche Anfrage an den Server senden kann, muss zwischen den beiden Akteuren zuerst eine TCP Verbindung bestehen – quasi ein Tunnel für die Datenübertragung. In Abbildung 2.1 ist dieser zweistufige Prozess erkennbar. Die Distanz zwischen Client und Server muss hier pro Stufe zweimal zurückgelegt werden – je einmal für Anfrage und Antwort. Die Zeit, die für einen Umlauf aufgebracht wird, wird Paketumlaufzeit (engl. *Round Trip Time*, RTT) genannt. Es liegt nahe zu denken, dass die Bandbreite, also die Geschwindigkeit, mit der man die Daten zwischen Client und Server versendet, einen entscheidenden Einfluss auf die Höhe der RTT hat. Mogul und Padmanabhan konnte den Einfluss der RTT auf die Ladezeit einer Anwendung in einem Experiment beschreiben. In ihrem Test brauchte eine für das Web durchschnittliche große Datei – etwa 20 Kilobyte – für den Transfer über eine definierte Strecke 100 Millisekunden. Sie nahmen dabei eine Bandbreite von 1,5 Megabits pro Sekunde an. Sie wiederholten den Test mit der maximal denkbaren Bandbreite – nur durch physische Kriterien begrenzt – und konnten die Übertragungsgeschwindigkeit nur um 30 Prozent reduzieren.[2] Die Autoren leiteten daraus ab, dass die Bandbreite zwar einen Einfluss auf die Latenz hat, aber nicht der entscheidenste Faktor

[2] Vgl. Padmanabhan/Mogul (1995).

ist, vor allem weil in der Praxis nur die wenigsten Anwender an den im Test definierten Bestwert herankommen würden.

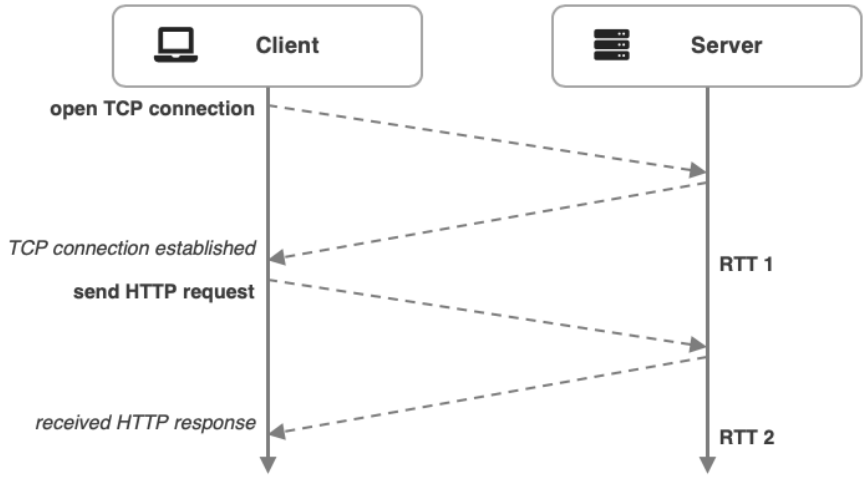

Abbildung 2.1: Ablauf HTTP Anfrage über TCP[3]

Aus den bisherigen Erkenntnissen lässt sich jedoch ableiten, dass die Anzahl der Round Trips eine Relevanz in Bezug auf die Ladegeschwindigkeit hat, zumal für eine Anwendungen mehrere Dateien zwischen Server und Client übertragen werden müssen. Laut Analyseergebnissen von 3,8 Millionen Webseiten hat das HTTP Archive ermittelt, dass pro Webseite durchschnittlich 74 Ressourcen übermittelt werden müssen.[4] Da für jede Übertragung theoretisch zwei Umläufe nötig sind, addiert sich die RTT mit zunehmender Anzahl zu übertragender Ressourcen schnell auf. Diese Schwachstelle wird in HTTP mit dem „keep-alive" Header adressiert. Ein Webbrowser kann dem Server bei einer HTTP Anfrage mitteilen, dass die TCP Verbindung nach Abschluss nicht beendet werden soll, sondern für folgende Anfragen wiederverwendet wird. Dadurch kann zumindest ein Umlauf pro Datei eliminiert werden, so lange die Verbindung bestehen bleibt.

Die Kommunikation in einem verteilten Netzwerk findet in der Regel nicht auf einem direkten Weg statt. Die übermittelten Nachrichten werden – je nach Entfernung

[3] Vgl. Flach et al. (2013).
[4] HTTP Archive (2018).

beider Akteure und Auslastung des Netzes – über mehrere Knotenpunkte geleitet. Eine Möglichkeit die Entfernung von Anwender zum angefragten Server zu verringern, ist der Einsatz von Content-Delivery-Netzwerken (CDN). Diese versuchen Spiegelungen des Servers strategisch im Internet zu verteilen. Damit fragt ein Client nun nicht mehr den eigentlichen Server an, sondern seinen nächstgelegenen Knotenpunkt im teilnehmenden CDN, der sich genauso wie der angefragte Server verhält. Damit kann zwar nicht die Anzahl der Umläufe reduziert werden, jedoch die Strecke, die für die Übermittlung zurückgelegt wird.

Letzter für diese Arbeit relevanter Faktor für die Höhe der Ladezeit ist der Rechenaufwand. Übermittelt ein Server nur statische Daten, wie Bilder, ist der Rechenaufwand marginal. Werden jedoch auf den anfragenden Nutzer spezifische Inhalte zusammengestellt, verzögert sich die Auslieferung der Daten. In einem mehrschichtigen System bezieht ein Anwendungsserver weitere benötigte Daten von externen Quellen wie Datenbanken.[5] Die Übertragung zwischen Server und Datenbank und der anschließenden Verarbeitung verzögert die Auslieferung an den Client weiter.

2.2 Caching zur Verringerung von Ladezeiten

Bisher konnte gezeigt werden, dass für den Aufruf einer Webanwendung viele verschiedene Daten zwischen Client und Server übertragen werden müssen. Für jede Datei muss die Strecke zwischen den beiden Akteuren im schlechtesten Fall zweimal zurückgelegt werden. Dieser Prozess beginnt bei jedem erneuten Aufruf von vorne. Ein sinnvoller Ansatz die Übertragung der genannten Ressourcen zu optimieren, ist das Zwischenspeichern (engl. *Caching*). Beim Caching wird versucht Daten nicht mehrmals zu übertragen, sondern sie nach dem ersten Bezug für die weitere Nutzung im Speicher zu behalten. Das Zwischenspeichern kann im bisher genannten System an verschiedenen Stellen erfolgen. Gleichzeitig sollen die gespeicherten Ressourcen allerdings nicht die Aktualität der Anwendung gefährden. Für Bereitsteller einer Webanwendung ist

[5] Offutt (2002).

die Verwendung vom Browsercache eine einfache Lösung Caching für eine Anwendung einzuführen. Für einzelne Ressourcen kann eine Ablaufzeit (engl. *Time to Live*, TTL) definiert werden. Der Client Browser speichert bei einem Aufruf die einzelnen Ressourcen so lange, bis die TTL abgelaufen ist. Erst dann versucht er eine neue Version vom Server abzufragen. Die Umsetzung erfolgt über HTTP, das vorsieht, Ressourcen mit einem „cachable" Attribut zu versehen.[6] Diese Praxis beschleunigt zwar nicht den ersten Aufruf einer Applikation – es müssen weiterhin alle Daten übertragen werden – aber jeden weiteren, so lange die TTL sinnvoll gewählt ist. Nachteilig ist jedoch, dass das Protokoll keine Möglichkeit vorsieht, die TTL vorzeitig zu umgehen. Sollte ein Betreiber eine Ressource aktualisieren, die er mit einer hohen Ablaufzeit versehen hat, muss er damit rechnen, dass viele Anwender eine alte Version bis zum Ablauf verwenden werden. Wenn ein Client eine Ressource zur Darstellung der Applikation verwenden möchte, deren Ablaufzeit überschritten ist, so fragt er die Ressource erneut beim Server ab. Es besteht allerdings theoretisch noch die Möglichkeit, dass die Ressource weiterhin unverändert ist und sie theoretisch umsonst übertragen wird. Um dies zu vermeiden, stellt HTTP das sogenannte „eTag" zur Verfügung. Mit diesem Attribut wird die Version einer Ressource definiert. Diese vom Server vergebene Zeichenkette wird zusammen mit der abgefragten Information übermittelt und ist einmalig in Bezug auf Version und Inhalt. Ändert sich eine Ressource, generiert der Server dafür auch ein neues eTag. Bei einer erneuten Abfrage übersendet der Client dann gleichzeitig mit einem „If-not-changed" Header dieses eTag. Der Server muss dann die vom Client übertragene Zeichenkette mit seiner aktuellen Version vergleichen. Weichen sie voneinander ab, übermittelt er die neue Version der Ressource und das entsprechende neue eTag. Sind beide Zeichenketten identisch – die Version des Clients ist also weiterhin aktuell – antwortet der Server, dass die Ressource nicht modifiziert wurde, übermittelt aber keine weiteren Daten. Dieser Ansatz spart zusätzlich unnötigen Datentransfer.

[6] Foundation (o. J.).

Im Gegensatz zum Browsercaching kann mit Proxy Caches bereits der erste Aufruf einer Webapplikation beschleunigt werden. Ein Beispiel hierfür sind die bereits beschriebenen Knotenpunkte eines CDN. Sie werden auch als Proxy, also Stellvertreter, betitelt und speichern eine Kopie dessen zwischen, was der eigentliche Server ausliefern würde. Aus diesem Grund werden sie auch „Surrogate Server“ genannt.[7] Auch der Server selbst kann einen Cache besitzen. Werden gewisse Abfragen regelmäßig angefragt, kann er die Ergebnisse davon unter Umständen zwischenspeichern, damit er die häufige Kommunikation zu einer externen Datenquelle verringern und schneller auf Anfragen antworten kann.

Es gilt allerdings zu beachten, dass das Ablegen von Ressourcen in einem Cache – egal ob bei Client, Proxy oder Server – nicht in jedem Fall sinnvoll ist. Ein Cache kostet unter anderem Speicherplatz und es sollten nur die Ressourcen gespeichert werden, die auch mehrmals zum Einsatz kommen könnten. Gerade die auf den Endanwender spezifisch angefertigte Anfragen sind oftmals einmalig und können in der gleichen Form nicht wiederverwendet werden. Ein Beispiel hierfür sind Empfehlungen in einem Online-Shop. Sie sind für jeden Benutzer unterschiedlich und verändern sich laufend. Diese Informationen werden in der Regel nicht im Cache behalten.

Gleichzeitigt birgt ein Cache das Risiko, dass er veraltete Daten gespeichert hat. Im Verlauf wurde bereits erwähnt, welche negativen Aspekte eine im Vorfeld definierte Ablaufzeit haben kann, wenn die Ressource vor Ablauf verändert werden sollte. Die Definition einer effizienten TTL stellt ein klassisches Maximierungsproblem dar. Sie soll so hoch sein wie möglich sein, damit eine Ressource nicht unnötig erneut abgefragt werden muss, aber gleichzeitigt so gering wie nötig, um die Aktualität zu gewährleisten.

2.3 Baqend Speed Kit

Das Startup Baqend entwickelt aufbauend auf jahrelang erworbenen wissenschaftliche Erkenntnisse, ein System zur Optimierung von Ladezeiten für bestehende und neu

[7] Vakali/Pallis (2003).

entwickelte Webanwendungen. Das sogenannte SPEED KIT verspricht Betreibern die Ladegeschwindigkeit ihrer eingesetzten Applikation in wenigen Schritten zwischen 50 und 300 Prozent zu verbessern.[8] Baqend stellt für die Anwender von SPEED KIT das Backend-as-a-Service (BaaS) ORESTES zur Verfügung, welches die Optimierung im Hintergrund durchführt. ORESTES ist datenbankunabhängig an unterschiedliche Anwendungen anbindbar.[9] Es kommen verschiedene Techniken zum Einsatz, vor allem aber wird Caching eingesetzt, um das Datenaufkommen zu minimieren.

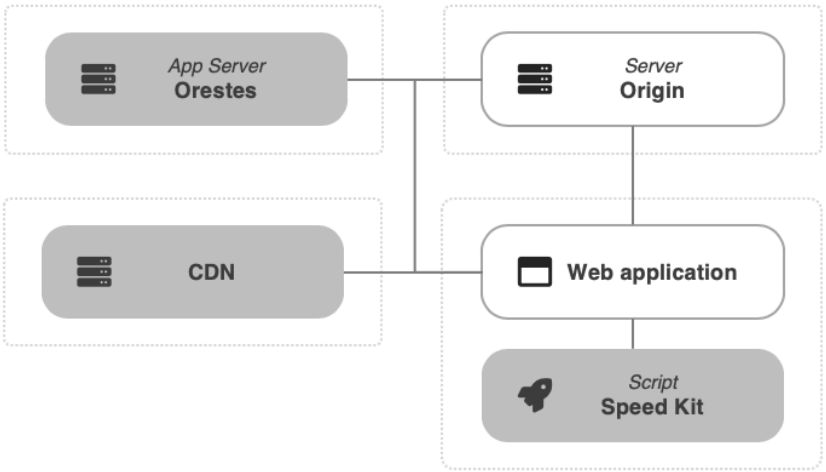

Abbildung 2.2: Baqends Systeme im Anwendungskontext[10]

Abbildung 2.2 hebt die Systeme hervor, die eine bestehende Anwendungsumgebung beim Einsatz von SPEED KIT ergänzen. Es ist zu sehen, dass Baqend ein CDN einsetzt, um die Auslieferung von Ressourcen näher an die Anwender zu bringen. Das dargestellte Skript ist die einzige Veränderung, die ein Betreiber an seiner Anwendung vornehmen muss und ist mit der Einbindung weniger Codezeilen durchführbar. Mit Hilfe dieses Skripts werden der Anwendung Funktionalitäten zur Verwaltung des Browsercaches hinzugefügt. Im Verlauf wurde beschrieben, dass eine festgesetzte TTL nicht vorzeitig unterbrochen werden kann. SPEED KIT kann mit Hilfe eines Service Workers

[8] Baqend GmbH (o. J. b).
[9] Gessert (2018), S. 92.
[10] Gessert (2018), S. 94.

jedoch jede Anfrage vom Browser abfangen – egal ob die Anfrage an den Server gerichtet ist oder an den Browsercache. Damit kann das normale Verhalten des Browsers weitestgehend überschrieben werden. Dies ermöglicht, dass Ressourcen auch vor Ablauf der TTL im Einzelfall erneut abgefragt werden, sollte dies erforderlich sein. Für diesen Zweck kommt ein „Bloom Filter" zum Einsatz, der hier auch *Cache Sketch* genannt wird.[11] Ein Bloom Filter stellt eine Datenstruktur dar, mit der ohne großen Rechenaufwand festgestellt werden kann, ob ein Objekt in dieser Struktur gespeichert ist oder nicht.[12] Falsch positive Ausgaben sind möglich, falsch negative jedoch nicht. Gibt ein Bloom Filter an, dass ein Datensatz nicht enthalten ist, so ist diese Angabe definitiv korrekt. Der *Cache Sketch* im SPEED KIT wird bei jedem Aufruf der Anwendung in aktueller Version vom Server an den anfragenden Client übermittelt. Er enthält eine Auflistung aller Ressourcen, die möglicherweise im Browsercache abgelaufen sein könnten. Jede Ressource, die der Client verwenden möchte und im Bloom Filter enthalten ist, muss vor Verwendung revalidiert werden – ungeachtet der eingestellten TTL. Dafür stellt der Client eine gewöhnliche HTTP Anfrage an den nächsten Surrogate Server im CDN. Dieser selbst beantwortet die Anfrage aus seinem eigenen Cache. In Abbildung 2.3 sind die verschiedenen Caches und der Bloom Filter erkennbar. Die Verwendung von je *expiration-based* Cache (durch Ablaufzeit) und *invalidation-based* Cache (manuelle Invalidierung) wird auch zweiseitige Cache Validierung genannt.[13]

[11] Gessert (2018).
[12] Bloom (1970).
[13] Gessert (2018), S. 124.

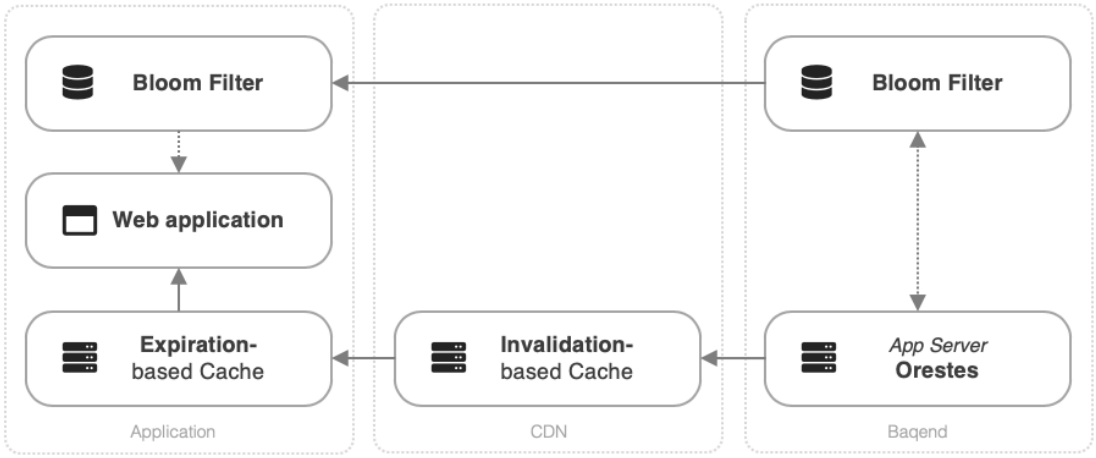

Abbildung 2.3: Bloom Filter und Caches im Speed Kit[14]

Der Cache des Clients wird also auf zwei Arten aktuell gehalten: Entweder erreicht eine Ressource die festgelegte TTL oder sie wurde schon vorher im Bloom Filter markiert. In beiden Fällen fragt der Client eine aktuelle Version über eine Anfrage ab. Der ORESTES Server verwaltet den Bloom Filter. Er muss feststellen, wann Ressourcen aktualisiert werden und dementsprechend reagieren. ORESTES verwaltet auch den Proxy Cache der Surrogate Server im CDN. Wird der Verfall einer Ressource erfasst, wird dem CDN in einer *purge* Anfrage der Auftrag übermittelt, diese Ressource im Cache zu aktualisieren. ORESTES kann auf zwei Arten feststellen, ob eine aktuellere Version einer Ressource zur Verfügung steht.[15] So können Betreiber der Anwendung ORESTES aktiv mitteilen, dass bestimmte Inhalte aktualisiert wurden. Dies findet im Regelfall jedoch wenig Einsatz. Daher bleibt nur die zweite Möglichkeit zur Feststellung der Aktualität: Es muss in regelmäßigen Abständen jede Ressource vom Anwendungsserver bezogen und mit der gespeicherten Version im Cache verglichen werden. Dieses Verfahren wird auch Cache-Revalidierung genannt. Je nach Umfang der Webanwendung werden mehrere Millionen Dateien verwaltet. Eine regelmäßige Revalidierung nimmt aus diesem Grund bis zu mehrere Stunden in Anspruch und wird daher meistens nur alle 24 Stunden und nachts durchgeführt. Dieser Mechanismus – die regelmäßige umfangreiche

[14] Gessert (2018), S. 126.
[15] Baqend GmbH (o. J. a).

Cache Revalidierung in Baqends Systemen – wird der Fokus der Arbeit sein und im Folgenden umfangreich bearbeitet.

3

Problemstellung und Konzeption

Die erste Forschungsfrage der vorliegenden Arbeit gibt vor, Schwachstellen in der Cache Revalidierungsroutine von Baqends ORESTES Applikation zu identifizieren. Aufbauend auf einer umfangreichen Beschreibung dieser Routine werden Probleme davon abgeleitet. Aus diesen Erkenntnissen lassen sich Anforderungen formulieren, die eine mögliche bessere Umsetzung erfüllen muss. Dies sind die Aspekte, die bereits in der aktuellen Implementierung ausreichend umgesetzt wurden, ergänzt um die identifizierten Schwachstellen.

Im Verlauf wurde bereits beschrieben, wie mit Hilfe von Baqends SPEED KIT eine Webanwendung um ein effizientes Caching ergänzt werden kann. ORESTES steuert dabei unter anderem, welche Ressourcen im Cache gespeichert werden dürfen und stellt gleichzeitigt deren Aktualität sicher. Für jeden Kunden von Baqend – also Bereitsteller einer Anwendung, der SPEED KIT verwendet – gibt es mindestens eine ORESTES Instanz. Die Applikation wird auf einer virtuellen Serverinstanz bei Amazon Web Services (AWS) ausgeführt. Je nach Umfang einer Anwendung wird eine einzelne ORESTES Instanz auf einem eigenen Server ausgeführt, in der Regel teilen sich jedoch mehrere Kunden einen gemeinsamen Server. Der vom Kunden eingesetzte Web- oder Anwendungsserver, der die zu optimierende Webanwendung bereitstellt, kann ohne Änderung beim alten Hoster weiter ausgeführt werden.

Für die Verwaltung der im Cache abgelegten Ressourcen verwendet ORESTES eine MongoDB Datenbank. In ihr wird für jede Ressource ein Eintrag hinterlegt, der neben der URL unter anderem auch einen Hashwert des Inhalts enthält. Dieser dient zur

Feststellung der Aktualität und wird im Folgenden noch näher erklärt. Die Ressourcen selber werden in einem AWS S3 Bucket abgelegt. Der entsprechende Verweis dorthin findet sich ebenfalls in den Datenbankeinträgen. In Abbildung 3.1 werden die einzelnen Systeme und deren Zusammenhänge verdeutlicht.

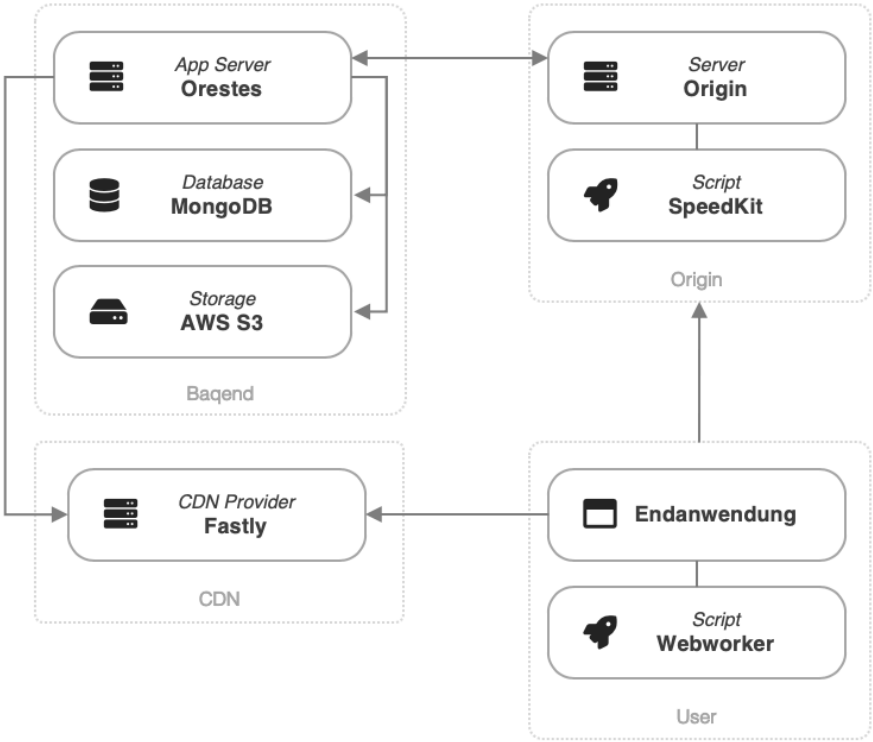

Abbildung 3.1: Systeme von Orestes im Kontext der Cache Revalidierung

Die gespeicherten Ressourcen können auf zwei Wege invalidiert werden. Der Bereitsteller einer Webanwendung kann die von Baqend angebotene *RefreshAPI* implementieren, um Ressourcen in Echtzeit im Cache zu aktualisieren oder zu löschen.[16] In der Regel werden jedoch terminierte umfassende Revalidierungsaufträge angelegt. Diese Aufträge können entweder über die Weboberfläche von ORESTES oder über eine API erstellt werden. Für die Definierung, welche Daten dort revalidiert werden sollen, können Filter ausgewählt werden. Eine Terminierung ermöglicht es umfassende Aufträge zu Zeiten auszuführen, in denen wenig gewöhnliche Auslastung zu erwarten ist. So

[16] Baqend GmbH (o. J. a).

werden die meisten Jobs nachts ausgeführt. Je nach Umfang der Webanwendung erreichen manche Revalidierungsaufträge mehrere hunderttausend bis zu über einer Millionen Dateien, die üblicherweise im 24 Stunden Takt überprüft werden.

3.1 Revalidierung vom Cache in Orestes

Wird ein Revalidierungsauftrag ausgeführt – egal ob terminiert oder durch API sofort angestoßen – wird für ihn eine einmalige Identifikationsnummer vergeben. Der aktuelle Status des Auftrags kann zu jeder Zeit unter anderem in der Weboberfläche von ORESTES eingesehen werden. Aus dem Ressourcenfilter wird eine Datenbankabfrage abgeleitet, mit der alle benötigten Datensätze ausgewählt werden. Die ID des Auftrags wird zu jedem Eintrag in der Datenbank hinzugefügt. Damit werden unter anderem doppelte gleichzeitige Revalidierungen einer Datei vermieden. Abbildung 3.2 enthält alle Schritte der Revalidierung sowie die Zusammenhänge zu externen Systemen und Datenquellen. Ein Auftrag definiert in einem zusätzlichen Attribut, ob das Löschen von Ressourcen im Cache erlaubt ist oder nicht. Entweder ist das Löschen komplett untersagt, teilweise erlaubt oder es soll alles gelöscht werden. Beim teilweisen Löschen werden nur die Ressourcen aus dem Cache entfernt, die veraltet sind. Ist das Löschen untersagt können hingegen nur Aktualisierungen vorgenommen werden.

Nachdem alle für den Auftrag benötigten Ressourcen zusammengestellt und markiert wurden, wird geprüft, ob sie in der vergangenen Periode von einem Anwender angefragt wurden oder nicht. Ist das Löschen erlaubt, werden die nicht angefragten Daten aus dem Cache entfernt. Diese Maßnahme schränkt den Umfang bzw. den Speicherbedarf des Caches ein, sodass nur häufig verwendete Dateien enthalten sind. Sollte eine Ressource gelöscht werden, so werden die entsprechenden Einträge in der Datenbank und dem S3 Bucket ebenfalls entfernt.

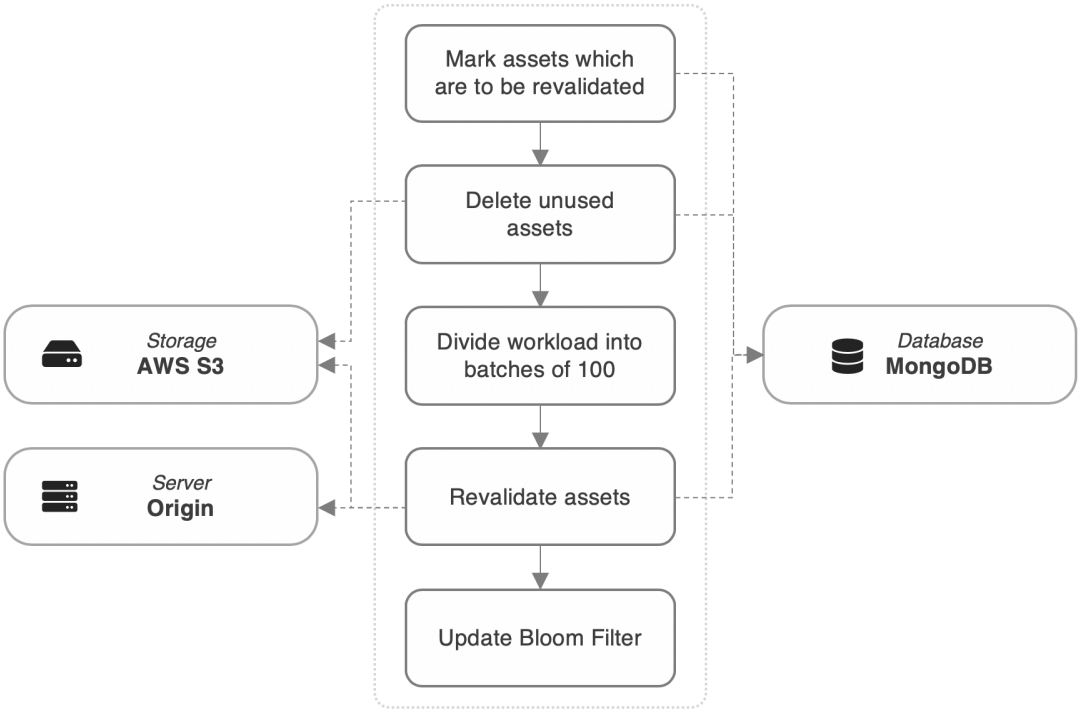

Abbildung 3.2: Ablauf der Cache Revalidierung in Orestes

Nach dem initialen Löschen bleiben noch die Dateien übrig, die revalidiert werden sollen. Die Arbeit wird in Pakete zu je 100 Ressourcen aufgeteilt. Die eigentliche Revalidierung einer Datei startet, indem der Original Webserver der Anwendung für diese Ressource angefragt wird. ORESTES führt sechs Anfragen gleichzeitigt zu einem Server durch, um ihn nicht unnötig zu belasten. Die Antwort des Origin wird mit einer gewöhnlichen Hashfunktion (CRC32) in eine Zeichenkette überführt. Der Hashwert, der dort errechnet wurde, wird mit dem Hashwert verglichen, den ORESTES bereits zu einem früheren Zeitpunkt von dieser Ressource gespeichert hat. Daraus wird die Aktualität der Ressource im Cache abgeleitet. Weichen die beiden Hashwerte voneinander ab, so ist die Ressource veraltet. Der neue Hashwert wird dann zusammen mit anderen Metadaten in der Datenbank aktualisiert. Die neue Ressource selbst wird im S3 Bucket abgelegt. Falls der Origin es unterstützt, sendet ORESTES bei den HTTP Anfragen auch ein bekanntes eTag mit. Somit kann im Fall, dass sich eine Ressource nicht verändert hat, übermittelte Daten gespart werden.

Falls der Origin mit einem 4xx Fehler antworten sollte, bedeutet dies, dass die angeforderte Ressource nicht auf diesem Server gefunden werden kann. Ist das Löschen bei diesem Auftrag erlaubt, wird die Ressource in diesem Fall, wie bereits beschrieben, aus allen Systemen entfernt. Antwortet der Webserver hingegen mit einem 5xx Fehler – einem Serverfehler – oder überschreitet die Anfrage eine festgegebene Zeit, wird dies als Serverfehler gewertet. Mit der Ressource wird nichts weiter unternommen, weil keine konkrete Antwort erhalten wurde. Häufen sich allerdings Serverfehler in einem Arbeitspaket, so wird der gesamte Revalidierungsauftrag storniert, um die Integrität des Caches nicht zu gefährden. Der Schwellenwert für den Abbruch ist bei 50 Prozent der Assets in einem Paket (also 50 Ressourcen) definiert.

Nachdem alle Ressourcen revalidiert wurden, wird der Bloom Filter aktualisiert. Alle Ressourcen, die sich geändert haben, werden nun aufgenommen. Beim nächsten Aufruf der Webanwendung durch einen Anwender wird der aktuelle Bloom Filter an den Client übertragen. Alle Ressourcen, die nun in aktuellerer Version am Server vorliegen, müssen vor einer Nutzung erneut angefragt werden. Damit jedoch die Surrogate Sever, an die sich der Browser richtet, selber die aktuellste Version jeder Ressource im eigenen Cache speichern, muss zuvor das CDN mit einer *purge* Anfrage über Änderungen unterrichtet werden.

3.2 Definition von Defiziten in der bisherigen Umsetzung

Die im Verlauf beschriebene Revalidierungsroutine bei ORESTES bietet an verschiedenen Stellen Ansatzpunkte für mögliche Optimierungen. ORESTES zeichnet sich dadurch aus, dass der Server zustandslos ist. Das kann jedoch Nachteile im beschriebenen Kontext hervorbringen. Sollte der Server während einer laufenden Revalidierung ungeplant und vorzeitig ausfallen, so ist bis auf weiteres nach einem Neustart keine Wiederaufnahme des Fortschrittes möglich. Eine aktuelle Gegenmaßnahme ist, den Status eines Auftrags zu überwachen. Dabei werden regelmäßig Aktualisierungen in einem Messagingsystem veröffentlicht. So lassen sich Rückschlüsse daraus ziehen,

zu welchem Zeitpunkt die Revalidierung abgebrochen ist. Gleichzeitig sind die Ressourcen unter Umständen noch mit der Auftragsnummer in der Datenbank markiert. Im Regelfall wird jedoch nach einem Ausfall ein etwaiger Auftrag manuell neugestartet, um sicherzustellen, dass alle Ressourcen korrekt bearbeitet werden. Da je nach Umfang eine Revalidierung allerdings mehrere Stunden, in einigen Fällen bis zu einem halben Tag dauern kann, ist ein Ausfall besonders gravierend, weil ein Fehlverhalten von ORESTES einen enormen Zeitverlust bedeutet.

Es wurde bereits beschrieben, dass der Origin Server nicht mit zu vielen gleichzeitigen Anfragen überlastet werden soll. Da selbst beim Einsatz von SPEED KIT viele Anfragen noch von diesem Server beantwortet werden, könnten zu viele gleichzeitige Anfragen bei der Revalidierung die Ladegeschwindigkeit für Anwender sogar noch verschlechtern. Dies soll auf jeden Fall vermieden werden. Jedoch hat die Anzahl der gleichzeitigen Verbindungen einen großen Einfluss auf die Bearbeitungszeit eines Revalidierungsauftrags. In einem einfachen Rechenbeispiel dauert der beschriebene Prozess bei 100.000 Ressourcen und einer durchschnittlichen Antwortzeit des Origin von 500 Millisekunden bei sechs gleichzeitigen Verbindungen 9,3 Stunden. Würde man die Anzahl der Verbindungen nur verdoppeln, könnte man diese Zeit halbieren. Dabei wird jedoch vorausgesetzt, dass der Origin bei einer Erhöhung der Verbindungen nicht langsamer antworten würde. Hieraus lässt sich ableiten, dass eine dynamische Anzahl von Verbindungen – abhängig von der Antwortzeit des Servers – die Bearbeitungszeit eines Auftrags beschleunigen kann. Heydon und Najork beschreiben mit ihrem „Domain-Based-Throttling“ für Webcrawler einen ähnlichen Mechanismus, mit dem versucht wird möglichst viele Daten von einem Server abzufragen, ohne ihn unnötig zu belasten.[17]

Es ist damit zu rechnen, dass der angefragte Webserver nicht immer durchgehend zu erreichen ist. Vor allem über den langen Zeitraum eines umfangreiches Revalidierungsprozesses kann es zu Verbindungsabbrüchen kommen oder der Server antwortet

[17] Vgl. Najork/Heydon (2002).

mit Fehlern. In diesem Fall wird die Revalidierung der Ressource übersprungen. Der Auftrag wird komplett abgebrochen, wenn 50 Prozent eines Arbeitspakets, also 50 Ressourcen nicht korrekt abgefragt werden können. Der Auftrag ist damit vorzeitig beendet und wird nicht wieder aufgenommen. Ein temporärer Ausfall vom angefragten Server führt dementsprechend aktuell dazu, dass der ganze Prozess abgebrochen wird. Gleichzeitig werden übersprungene Ressourcen auch bei einem sonst korrekt ausgeführten Auftrag nicht erneut revalidiert. Hieraus lässt sich ableiten, dass ein Auftrag im Falle von Fehlverhalten des Webservers nicht komplett abgebrochen werden sollte, sondern nur temporär ausgesetzt. Ressourcen sollten zwar übersprungen werden können, die Revalidierung sollte jedoch noch im selben Auftrag zu einem späteren Zeitpunkt wiederholt werden.

Der oft mehrstündige Prozess stellt für ORESTES zudem eine Dauerbelastung dar. Zwar erzeugt die Revalidierung keinen großen Rechenaufwand, jedoch werden dennoch Ressourcen verbraucht. Gerade wenn eine Erhöhung der gleichzeitigen Verbindungen erwogen wird, würde dies verlorene Netzwerkkapazitäten bedeuten. Wie bereits erklärt, teilen sich unter Umständen mehrere SPEED KIT Kunden einen gemeinsamen Server für ORESTES. Ein umfangreicher Revalidierungsauftrag von einem Kunden kann daher eine Belastung für die ORESTES Instanz eines anderen Kunden auf dem gleichen Server darstellen. Da es zu den Zeiten, an denen keine Revalidierung durchgeführt wird durchaus sinnvoll ist die Ressourcen zu teilen, ist es nicht ratsam dieses Konstrukt zu verändern. Würde man jedoch den Arbeitsaufwand auf andere Hardware auslagern, würde die Belastung von ORESTES verringert. Dies könnte für eine Optimierung in Erwägung gezogen werden. Die Auslagerung würde gleichzeitig einen entscheidenden Vorteil mit sich bringen: Fällt ORESTES während der Revalidierung aus, kann der Prozess in einer getrennten Applikation unabhängig weiter ausgeführt werden.

Identifiziertes Problem	Vorgeschlagene Lösung
Fehlverhalten von Orestes führt zu Abbruch des Revalidierungsauftrags	Auslagerung der Logik
Statische Anzahl Verbindungen verlängert Bearbeitungszeit	Dynamische Abfragerate
Fehlverhalten des Origin kann zum Abbruch des Revalidierungsauftrags führen	Verzögerte Wiederholung bis Fehlverhalten nicht mehr auftritt
Ressourcenverbrauch wirkt sich auf andere Kunden aus	Auslagerung der Logik

Tabelle 1: Identifizierte Probleme und vorgeschlagene Lösungen

In Tabelle 1 sind alle im Verlauf erläuterten Defizite bei der aktuellen Cache Revalidierungsroutine in ORESTES aufgelistet. Grundsätzlich lassen sich drei verschiedene Maßnahmen ableiten, mit denen man die Problemstellungen verbessern könnte. Um die Laufzeit einer Revalidierung zu verringern, könnte die Anzahl der gleichzeitig zu revalidierenden Ressourcen abhängig von der aktuellen Antwortzeit des Origin Webservers gestaltet werden. Lässt seine Reaktionsrate zusätzliche gleichzeitige Anfragen zu, könnte die Revalidierung schneller erfolgen. Ist der Webserver temporär zudem nicht erreichbar oder produziert sonstige Fehler, so sollte die Revalidierung nicht komplett abgebrochen, sondern zu einem späteren Zeitpunkt erneut versucht werden. Außerdem konnte erarbeitet werden, dass die Revalidierungslogik bei einer Auslagerung in eine eigene Applikation auf eigener Hardware zwei verschiedene Probleme adressieren kann. Zum einen schränkt ein ungeplanter Ausfall von ORESTES die Revalidierung nicht ein, zum anderen würde der Ressourcenverbrauch der Revalidierung nicht die Funktionsweise von ORESTES beeinträchtigen.

3.3 Formulierung der Anforderungen

Aus den Lösungsideen zur Verbesserung der bestehenden Revalidierungslogik wurde abgeleitet, dass die Revalidierung autonom von ORESTES stattfinden soll. Diese neue Implementierung soll zudem Anfragen dynamisch skalieren und geregelt auf Fehler

reagieren können. Im Folgenden sollen alle Anforderungen an die neue Applikation aufgelistet und erörtert werden. Diese Anforderungen stammen zum Teil aus dem jetzigen Verhalten der Methode, aus den vorangegangenen Ideen und aus enger Absprache mit Baqend. Unabhängig von der konkreten Umsetzung ist gefordert, dass eine Instanz der neuen Applikation von allen ORESTES Servern gleichzeitig genutzt werden kann. Im Umkehrschluss soll keine Anwendung konzipiert werden, die jeder ORESTES Server selbst verwalten muss. Dies setzt jedoch voraus, dass die Anwendung skalierbar ist, so dass sie auf Lastspitzen der verschiedenen ORESTES Server angemessen reagieren kann.

Da die neue Methode die gleichen Ergebnisse liefern wie die alte, die konkrete Revalidierung in Bezug auf den Hashwertvergleich soll übernommen werden. Im Fehlverhalten vom angefragten Webserver soll jedoch wie im Verlauf beschrieben entsprechend reagiert werden. Gleichzeitig müssen jedoch neue Abbruchbedingen definiert werden, sodass ein Auftrag nicht unendlich lange ausgeführt wird, sondern auf Fehler entsprechend reagieren kann. Zudem soll die Anfragerate auf die Antwortzeit angepasst werden. Der Status eines Revalidierungsauftrags soll weiterhin nachvollziehbar sein. Wünschenswert ist jedoch den genauen Fortschritt in Bezug auf die revalidierten Ressourcen überwachen zu können. Die neue Applikation soll zudem die Ressourcen in den AWS S3 Buckets abspeichern können. Die Verwaltung der Metadaten in der Datenbank soll aus verschiedenen Gründen weiterhin in ORESTES erfolgen. Sollte die neue Anwendung selbst ausfallen, soll der letzte Zustand wiederhergestellt werden, so dass die Arbeit nach einem Neustart automatisch fortgesetzt werden kann.

3.4 Implementierung eines verteilten Systems

In diesem letzten Schritt soll eine Grobplanung der Architektur der neuen Umsetzung der Revalidierungslogik vorgenommen werden. Eine der Hauptanforderungen ist die unabhängige Ausführung von Revalidierungen von ORESTES. Diese Anforderungen

werden vom Architekturprinzip von Microservices grundlegend abgedeckt. Ein Microservice ist eine kleine Applikation, die einen einzigen Zweck erfüllt. Eine solche Implementierung soll dezentralisiert funktionieren und vollkommen autonom handeln.[18] Mit Hilfe von Cloud Computing können Microservices zugleich kosteneffizient betrieben werden. Je nach aktueller Last können unterschiedlich viele Instanzen dieses Services ausgeführt werden und dementsprechende Hardware allokiert werden. Für eine konkrete Umsetzung würde ein Microservice für den geforderten Kontext ein Programm darstellen, das nur genau diese eine Aufgabe erfüllen muss. Dieses Programm erhält dann die Revalidierungsaufträge von ORESTES und gibt die Ergebnisse entsprechend zurück.

Durch die Verteilung der Aufgaben erfordert der Einsatz eines Microservices allerdings den Austausch von Daten zwischen den unterschiedlichen Systemen; hier die Revalidierungsroutine und ORESTES. Eine Schnittstelle kann über unterschiedliche Technologien selbstständig implementiert werden – oder man setzt eine Message Oriented Middleware (MOM) ein. Letztere bieten vor allem die Möglichkeit, dass ein Nachrichtenaustausch nicht von der Erreichbarkeit der Kommunikationsteilnehmer abhängig ist. Alle Nachrichten werden über die Middleware an die entsprechenden Empfänger weitergeleitet. Sollten diese temporär nicht erreichbar sein, werden die Nachrichten so lange gespeichert, bis eine Übertragung stattfinden kann.[19] Eine vielversprechende Implementierung einer solchen Middleware ist Apache Kafka. Anfangs für den Einsatz bei LinkedIn konzipiert, verspricht Kafka sehr große Datenmengen zwischen unterschiedlichen Systemen effizient zu vermitteln. Netflix verarbeitet beispielsweise täglich zwei Milliarden Events, die mit Kafka gespeichert und verwaltet werden.[20] Applikationen können bei Kafka zwei verschiedene Rollen einnehmen: Produzenten (Producer) oder Konsumenten (Consumer), je nachdem ob Nachrichten

[18] Nadareishvili et al. (2016), S. 7.
[19] Vgl. Banavar et al. (1999).
[20] Valtonen (2018).

übermittelt oder verarbeitet werden sollen. Durch das „Decoupling senders from receivers“[21] müssen beide Parteien den konkreten Kommuikationspartner nicht kennen. Die Vermittlung wird durch Kafka komplett übernommen.

[21] Valtonen (2018).

4

Verarbeitung von Data Streams in Kafka

Mit Hilfe von Apache Kafka soll die Datenübermittlung zwischen ORESTES und dem zu entwickelnden Microservice für die Cache Revalidierung (im Verlauf auch REVALIDATOR genannt) ermöglicht werden. Die vorgestellte Umsetzung orientiert sich an den bereits vorgestellten Anforderungen. Das Ziel soll es sein, dass Kafka eine komplett autonome Datenübertragung zwischen allen Akteuren ermöglicht. Im Verlauf wird deutlich werden, dass der Einsatz von Kafka jedoch über die reine Kommunikation hinausgeht. Daher kann mit Kafka kann ebenfalls eine Möglichkeit geschaffen werden, dass die Revalidierungsaufträge an die korrekten Microservices gelangen ohne dass ORESTES die konkreten Instanzen kennen muss; Kafka soll hier die Rolle eines Vermittlers einnehmen. Die Kommunikation beschränkt sich nicht nur auf den Datenaustausch zwischen ORESTES und REVALIDATOR, auch die Microservices sollen sich in bestimmten Fällen über Kafka selbstständig organisieren können. Ebenfalls wird verdeutlicht wie sich die Bereitstellung von Apache Kafka im Betrieb in die bereits existierende Baqend Umgebung eingliedern wird.

Es wurde bereits beschrieben, dass ein Revalidierungsauftrag bei ORESTES eine Auflistung von Ressourcen enthält. Diese Ressourcen gilt es zu revalidieren. Die Revalidierung soll in der hier vorgeschlagenen Lösung nun nicht mehr von ORESTES selbst durchgeführt werden, sondern der Arbeitsaufwand soll an den neuen Microservice übergeben werden. Baqend setzt zahlreiche ORESTES Instanzen für die unterschiedlichen Kunden ein. Jeder ORESTES Server soll Zugriff auf die gleichen Microservices haben. Je nach Auslastung des Systems werden mehr oder weniger Microservices

ausgeführt; die konkrete Anzahl kann also variieren. Wenn es die Auslastung erlaubt, sollen mehrere Microservices einen gemeinsamen Revalidierungsauftrag bearbeiten können. Das Zusammenspiel der unterschiedlichen ORESTES und Microservice Instanzen kann Abbildung 4.1 entnommen werden. Dort wird verdeutlicht, dass keine direkte Beziehung zwischen ORESTES und REVALIDATOR besteht und Kafka in jedem Fall als Middleware verwendet wird. Im Folgenden wird eine Möglichkeit vorgestellt, die die Gesamtheit aller Aufträge verwaltet und wie letztere effizient von den neuen Services revalidiert werden können.

Abbildung 4.1: Beziehung zwischen den einzelnen Akteuren im neuen System

4.1 Grundlegende Definitionen

Applikationen, die über Kafka kommunizieren, können zwei verschiedene Rollen einnehmen: Produzenten oder Konsumenten, je nachdem ob eine Nachricht übermittelt oder verarbeitet wird. Die Gestaltung der übermittelten Key-Value-Nachrichten selbst kann frei vorgenommen werden. Komplexe Objektstrukturen oder andere Datenformate müssen jedoch vor der Übermittlung serialisiert werden. Übermittelte Nachrichten werden in Kafka in sogenannten „Topics“ in eingehender Reihenfolge zusammen mit einem Zeitstempel abgelegt. Topics wiederum sind in Partitionen gegliedert. Diese werden in Kafka verwendet, um den gespeicherten Inhalt in einem Kafka-Cluster zur Ausfallsicherheit zu replizieren.

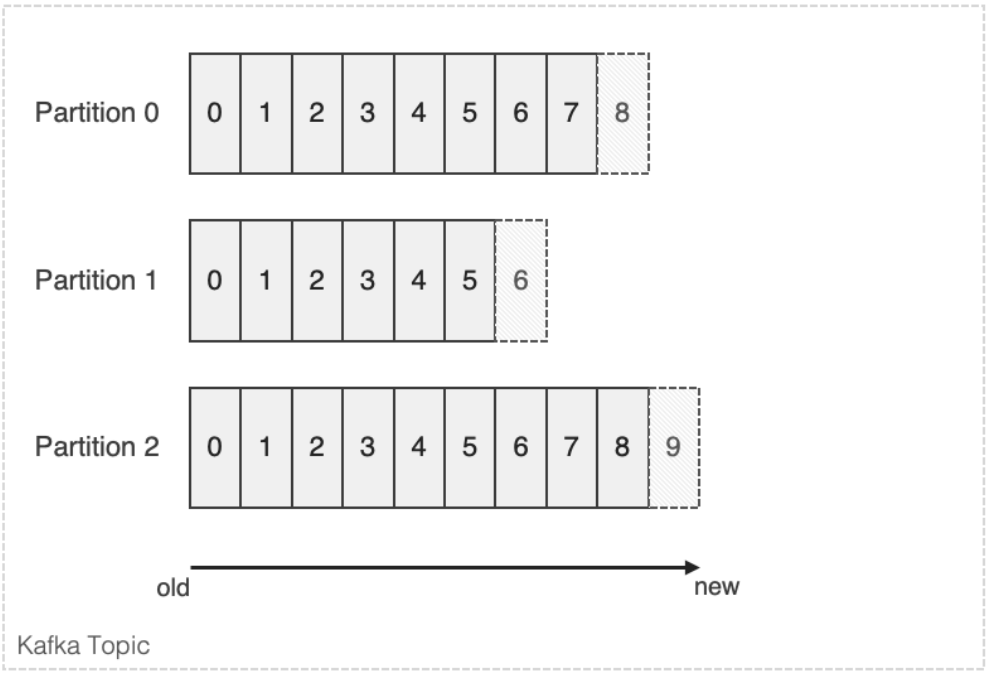

Abbildung 4.2: Speicherung von Daten in einem Kafka Topic[22]

Abbildung 4.2 zeigt den Aufbau eines Topics in Kafka. Die genaue Definierung eines Topics obliegt dem Entwickler, in der Regel beschreibt ein Topic jedoch eine einzelne Art von Nachricht, die regelmäßig zwischen mindestens zwei Systemen ausgetauscht wird. Ein Produzent – eine Applikation, die eine Nachricht übermittelt – spezifiziert neben dem eigentlichen Inhalt auch in welchem Topic die Nachricht abgelegt werden soll.[23] Bei einer erfolgreichen Übermittlung speichert Kafka die Daten in einer der Partitionen ab und vergibt jeweils einen fortlaufenden Offset. Konsumenten initiieren ihre Teilhabe im System, indem sie ein bestimmtes Topic abonnieren. Werden dort neue Nachrichten von Produzenten veröffentlicht, werden sie an die Konsumenten weitergeleitet. Kafka speichert für jeden Konsumenten den bereits gelesenen Offset. Gleichzeitig bestätigt ein Konsument den Erhalt einer Nachricht.[24] Dieser Mechanismus garantiert, dass eine Nachricht mindestens und gleichzeitig maximal einmal von einem Konsumenten gelesen wird.

[22] The Apache Software Foundation (2017b).
[23] Manche Implementierungen lassen zusätzlich auch die Definition einer bestimmten Partition zu.
[24] Dies ist jedoch abhängig von der konkreten Implementierung der verschiedenen Kafka APIs.

Die Nachrichtenübermittlung findet vollkommen asynchron statt. Nachrichten können zwar in Echtzeit übermittelt werden, aber nur wenn ein Konsument aktiv einem Topic folgt. Ansonsten wird die Übermittlung verzögert bis ein Konsument verfügbar ist. Kafka ermöglicht es, dass unterschiedliche Produzenten gleichzeitig Nachrichten in einem gemeinsamen Topic veröffentlichen können. Die gleichzeitige Verarbeitung von mehreren Konsumenten ist ebenfalls möglich. Es ist jedoch anzumerken, dass jeder Konsument seinen eigenen Offset führt. Konsumenten können in sogenannten „Consumer Groups" zusammengeschlossen werden. Diese Gruppierung ermöglicht, dass sich mehrere Konsumenten einen gemeinsamen Offset teilen. Abbildung 4.3 zeigt eine entscheidende Limitation: Eine Partition kann nur von einem Konsumenten in einer Gruppe gelesen werden. Möchte man also mehrere Konsumenten gleichzeitig Nachrichten eines Topics lesen lassen, dann muss es mindestens so viele Partitionen wie Konsumenten geben.

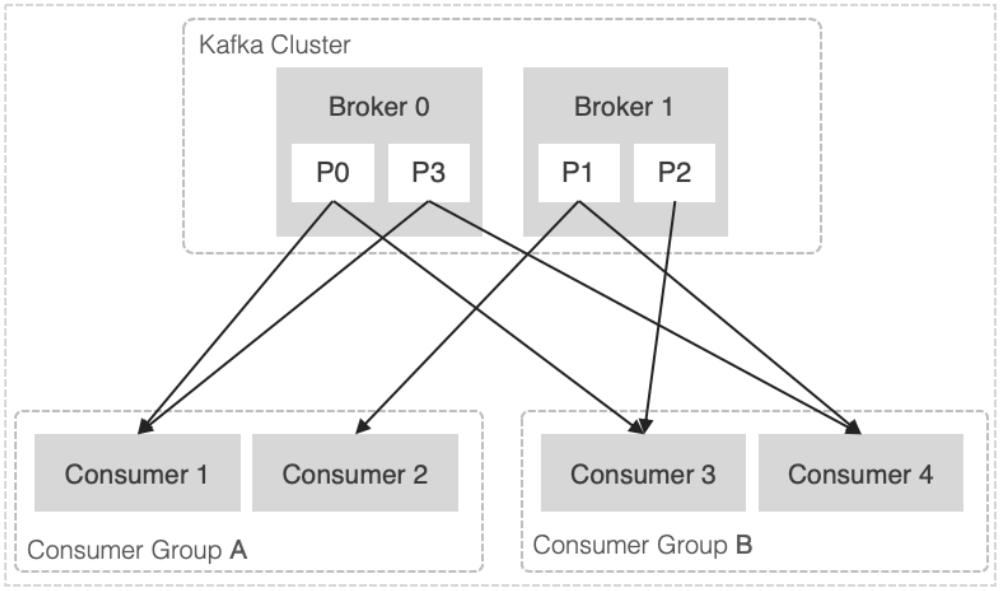

Abbildung 4.3: Gruppierte Konsumenten im Zusammenhang mit Partitionen[25]

Durch einen veränderbaren Faktor kann die Anzahl der Partitionen für ein Topic definiert werden. Die eigentliche Speicherung und Verwaltung der Daten finden bei

[25] The Apache Software Foundation (2017b).

Kafka in sogenannten „Brokern“ statt. In der Regel werden pro Kafka-Cluster mehrere Broker eingesetzt. Sie dienen der Lastverteilung, aber auch der Ausfallsicherheit. Die Partitionen werden so auf die einzelnen Broker verteilt, dass im Fehlverhalten eines Teilnehmers im System die Partitionen auf die noch verfügbaren Broker neu verteilt werden. Ein Datenverlust ist daher bei ausreichender Anzahl von Brokern ausgeschlossen. Gleichzeitig wird sichergestellt, dass das Cluster jederzeit erreichbar ist.

Der Einsatz von Kafka ermöglicht drei verschiedene Aufgabengebiete in einem System gebündelt: Die Speicherung von Daten, die Übermittlung von Daten in einem Messaging-System und die Verarbeitung von Datenströmen in Echtzeit.

4.2 Vorbereitung zur Umsetzung der Anforderungen in Kafka

Es wurde bereits beschrieben, wie ein zu implementierender Microservice die bestehende Revalidierungsroutine von ORESTES auslagern soll. Der neue Service wird von ORESTES eine Auflistung von Ressourcen erhalten, die es zu revalidieren gilt. Die Ergebnisse werden dann teilweise schon beim Microservice verarbeitet oder an ORESTES zurückgesendet. Die genaue Implementierung folgt in Kap. 5 An dieser Stelle reicht ein grundlegendes Verständnis über den Nutzen des Service aus. Mit Hilfe von Kafka soll unter anderem die Datenübertragung zwischen Revalidierungsservice und ORESTES umgesetzt werden. Im Verlauf wurde ein grundlegendes Verständnis von Apache Kafka vermittelt. Nun soll beschrieben werden, wie der konkrete Einsatz der Nachrichtenverwaltung in der vorgeschlagenen Lösung aussieht.

Verwaltung von Revalidierungsaufträgen

Die konkrete Umsetzung sieht vor, dass sowohl ORESTES als auch Microservice zu verschiedenen Zeiten die Rolle des Konsumenten und des Produzenten einnehmen, je nachdem welche Richtung der Informationsfluss einnimmt. Ein Revalidierungsauftrag wird in ORESTES erstellt und regelmäßig ausgeführt. Er enthält eine einmalige Identifizierungsnummer und alle Ressourcen, die revalidiert werden sollen. In einer denkbaren

Umsetzung (Abbildung 4.4) können ORESTES Instanzen einen Auftrag in einem „Job Topic“ veröffentlichen. Eine einzelne Nachricht enthält in einer serialisierten Datenstruktur (etwa JSON) eine Auflistung aller Ressourcen, die revalidiert werden sollen. Der Revalidierungsservice empfängt die Nachrichten und bearbeitet die Aufträge in eingehender Reihenfolge.

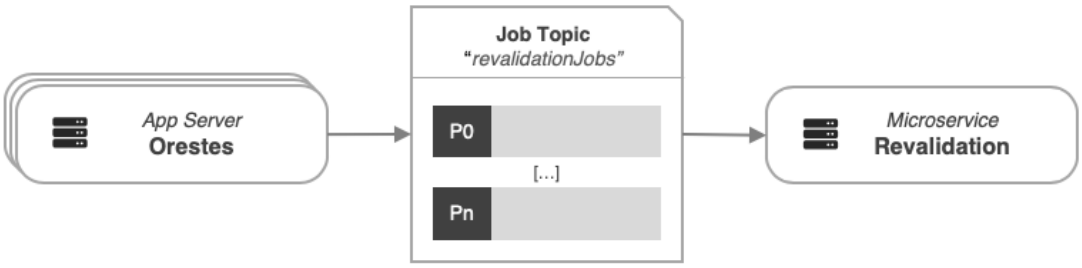

Abbildung 4.4: Denkbare Umsetzung für Revalidierungsaufträge

In dieser Lösung kann allerdings die Last nicht auf verschiedene Microservices verteilt werden. Denkbar ist, dass mehrere Revalidierungsservices simultan ausgeführt werden. Ein Leerlauf sollte deswegen vermieden werden. Gleichzeitig ist die Datenmenge, die in einer einzelnen Nachricht übermittelt werden soll bei mehreren hunderttausend Ressourcen ineffizient für die Übermittlung in Kafka. Es gilt zudem weiterhin einen Weg zu finden, wie der Microservice die Ergebnisse seiner Revalidierung an die korrekte ORESTES Instanz zurücksenden kann.

Kafka als API Gateway

Ein grundlegendes Problem bei der Verwendung von Microservices ist die Auffindbarkeit. Für die Service-Discovery können Systeme eingesetzt werden, mit denen Applikationen einen laufenden Service für einen bestimmten Zweck finden können.[26] Im vorliegenden Fall könnte ORESTES bei dem Einsatz eines einzigen Microservice die konkrete IP speichern, um ihn zu adressieren. Allerdings soll die Last auf verschiedene Serviceinstanzen verteilt werden. Es besteht damit die Problematik, wie ORESTES einen

[26] Nadareishvili et al. (2016), S. 96.

Revalidierungsauftrag an einen Service schicken kann, ohne ihn konkret zu kennen bzw. ohne einen Weg ihn zu adressieren. Für diesen Zweck werden sogenannte „API-Gateways“ eingesetzt. Diese Systeme dienen als Vermittler: Eine Applikation wendet sich mit einem Anliegen an ein API-Gateway, welches die Anfrage an einen entsprechenden verfügbaren Microservice weiterleitet – alles unter Berücksichtigung von Lastverteilung und anderen Eigenschaften.[27]

Um die Komplexität und damit die Anzahl der Systeme nicht unnötig zu erhöhen, wird in der vorgeschlagenen Umsetzung Kafka als API-Gateway eingesetzt. Auch wenn keinen eigentlichen Zweck der Software darstellt, können die Funktionalitäten eines solchen Gateways mit Kafka abgebildet werden. Anders als bei einem gewöhnlichen Gateway werden Anfragen allerdings nicht an einen Service weitergeleitet, sondern Serviceinstanzen bearbeiten selbstständig verfügbare Aufgaben. Dies soll im Verlauf verdeutlicht werden. In bisherigen wissenschaftlichen Veröffentlichungen konnte keine ähnliche Verwendung von Kafka gefunden werden.

Die im Verlauf beschriebene Umsetzung wird um ein „Broadcast Topic“ ergänzt (Abbildung 4.5). Die Idee dieser Art von Topic ist, dass verschiedene Applikationen gleichzeitig Produzenten und Konsumenten sind. Die dort übermittelten Nachrichten sind für alle Akteure gleichermaßen relevant. Die Nachrichten werden in einer einzelnen Partition abgelegt, weil nur so garantiert werden kann, dass die Reihenfolge beim Schreiben und Lesen identisch ist.[28] Alle Serviceinstanzen (in der Abbildung mit „RS“ gekennzeichnet“) abonnieren ein gemeinsames „*Job Status Topic*“. ORESTES veröffentlicht dort einen Hinweis, dass ein neuer Revalidierungsauftrag vorliegt, jedoch keine konkreten zu revalidierenden Ressourcen. Ein verfügbarer Service, der gerade keine andere Revalidierung durchführt, übermittelt dann über das gleiche Topic, dass er die

[27] Nadareishvili et al. (2016), S. 97.

[28] Nachrichten werden zwar in eingehender Reihenfolge in einem Topic abgelegt, aber auf alle verfügbaren Partitionen verteilt. Beim Konsum werden Partitionen unter Umständen in anderer Reihenfolge ausgegeben. Nur durch die Verwendung einer einzelnen Partition kann die korrekte Reihenfolge beibehalten werden. Hierbei nimmt man dann allerdings alle damit einhergehenden Nachteile in Kauf.

Revalidierung von diesem Auftrag übernehmen wird. Alle anderen Services konsumieren alle beschriebenen Nachrichten gleichermaßen. Sie erhalten ebenfalls den neuen Auftrag, allerdings auch die Nachricht, dass schon ein anderer Service die Revalidierung übernimmt und warten deswegen auf den nächsten Auftrag.

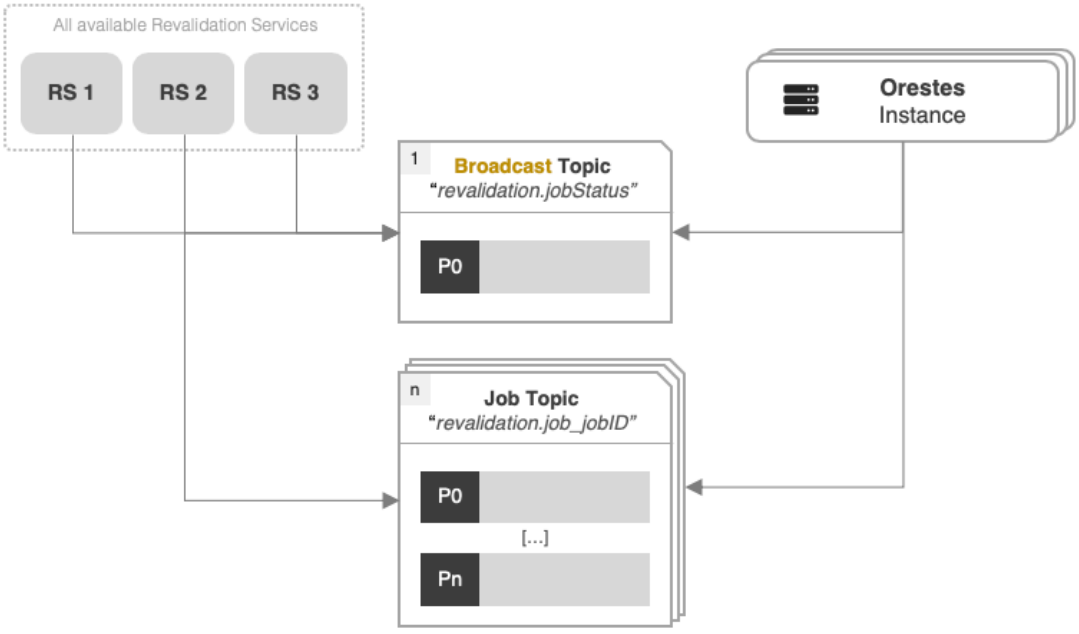

Abbildung 4.5: Verwendung eines Broadcast Topic für die Vermittlung von Aufträgen

Die konkreten Ressourcen, die revalidiert werden sollen, werden auf Pakete mit je 100 Datensätzen verteilt. Jedes Paket entspricht einer Nachricht in Kafka und wird in einem eigenen Topic für den Job veröffentlicht. So wird das Broadcast Topic für die Auftragsverwaltung von den Topics der eigentlichen Daten getrennt. Dies spart unnötigen Datenkonsum. Gleichzeitig erhält jeder Job ein Attribut, mit dem definiert werden kann, wie viele Microservices einen Auftrag gleichzeitig revalidieren sollen. Somit hat ORESTES einen Einfluss auf die Lastverteilung der Aufträge, weil das Gesamtproblem auf kleine Datenpakete verteilt wird, die von unterschiedlichen Konsumenten bearbeitet werden können. Ein Konsument erhält erst dann das nächste Arbeitspaket, wenn die Abarbeitung des vorherigen abgeschlossen ist.

Das vorgestellte Broadcast Topic für die Auftragsverwaltung weist allerdings noch eine Schwachstelle auf. In einem Test mit einem Auftrag, der nur einen Revalidierungsservice benötigt, konnten sich fälschlicherweise mehrere verfügbare Services gleichzeitig für die Revalidierung registrieren. Da jeder Service einen einzelnen Konsumenten darstellt, bearbeiteten mehrere Applikationen damit die gleichen Arbeitspakete. Die Echtzeit Datenübertragung von Kafka führte zu diesem Problem. In der ersten Nachricht im Auftragstopic wurde von ORESTES veröffentlicht; dass ein neuer Job verfügbar ist. Alle Services reagieren gleichzeitig auf diese Nachricht, indem sie die Revalidierung des Auftrags übernehmen wollen und dies korrekt in dem Topic veröffentlichen. Es muss daher eine Möglichkeit gefunden werden, wie man trotz Echtzeitübertragung einen Austausch zwischen den Services ermöglicht, sodass nur die geforderte Anzahl von Applikationen an einer Revalidierung arbeiten können.

Broadcast Topic für Subscriptions

Die bereits vorgestellte Umsetzung wird um ein weiteres Broadcast Topic ergänzt (Abbildung 4.6). Das Topic „Subscriptions" beinhaltet den Austausch aller verfügbaren Revalidierungsservices in Bezug auf die Übernahme (Subscription) eines Revalidierungsauftrags. Wenn ein Service startet, veröffentlicht er seinen initialen Status („available") im Subscription Topic. Von ORESTES stammt weiterhin die Nachricht mit dem Hinweis zu einem neuen Revalidierungsauftrag im entsprechenden Topic. Alle verfügbaren Services nehmen dies zur Kenntnis und starten in einem ersten Schritt einen Versuch zur Übernahme der Revalidierung. Dafür aktualisieren sie ihren Status im entsprechenden Topic und geben an, dass sie sich für einen Auftrag mit definierter Identifikationsnummer registrieren wollen: „*RS 1 attempt Job_1*". Kafka speichert alle Statusänderungen in der Reihenfolge, in der sie eintreffen in die einzige Partition des Subscription Topics. Alle Services sind gleichzeitig als Konsumenten dort registriert und lesen alle Nachrichten in derselben Reihenfolge. Aus diesem Grund werden sie nicht nur die eigene Nachricht mit dem Registrierungsversuch lesen, sondern auch die

der anderen Services. Im Beispiel hat „RS 1“ den Versuch als erstes veröffentlicht und ändert seinen Status zur Bearbeitung des Auftrags. „RS 2“ war nicht schnell genug, hat festgestellt, dass „RS 1“ schneller war – und wartet wieder auf einen anderen Job.

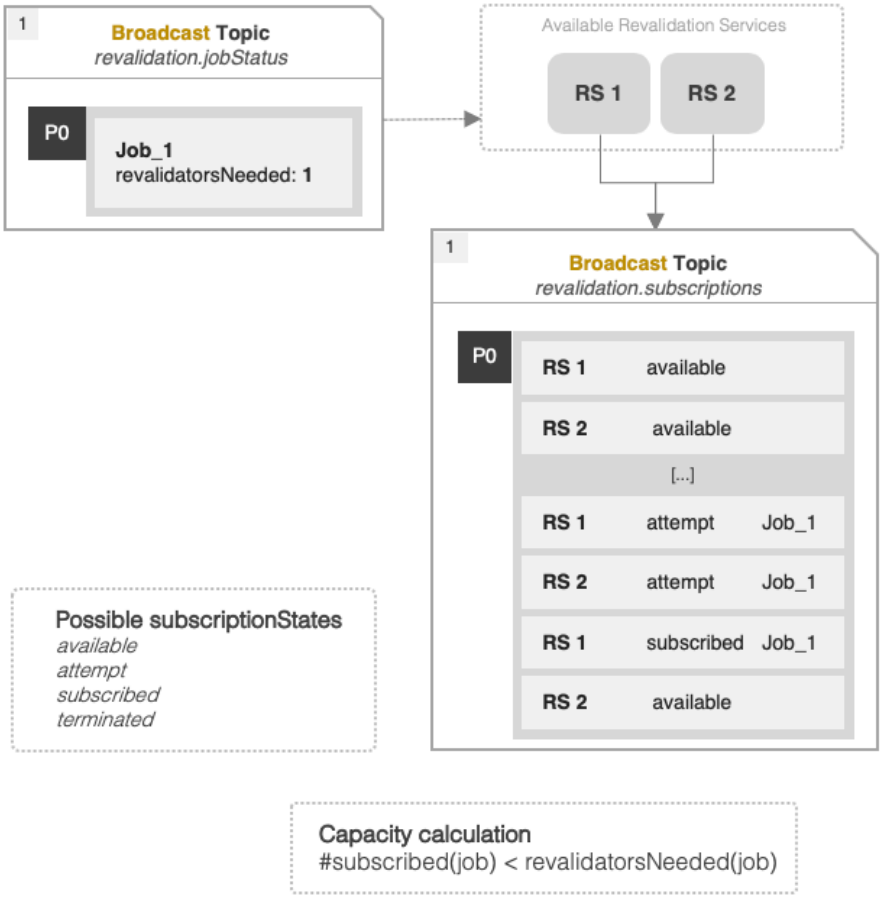

Abbildung 4.6: Broadcast Topic für Job Subscriptions

Diese Implementierung ermöglicht es, dass nur so viele Revalidierungsservices an einem Auftrag arbeiten, wie dies gewünscht ist. Dies setzt allerdings voraus, dass die Services die Verwaltung selbst übernehmen. Anders als bei einem klassischen API-Gateway gibt es hier keine kontrollierende Instanz. Sollte ein Service seine Ausführung vorzeitig beenden, so ändert sich sein Status entsprechend. Die verbleibende Revalidierung kann dann von einem anderen verfügbaren Service übernommen werden.

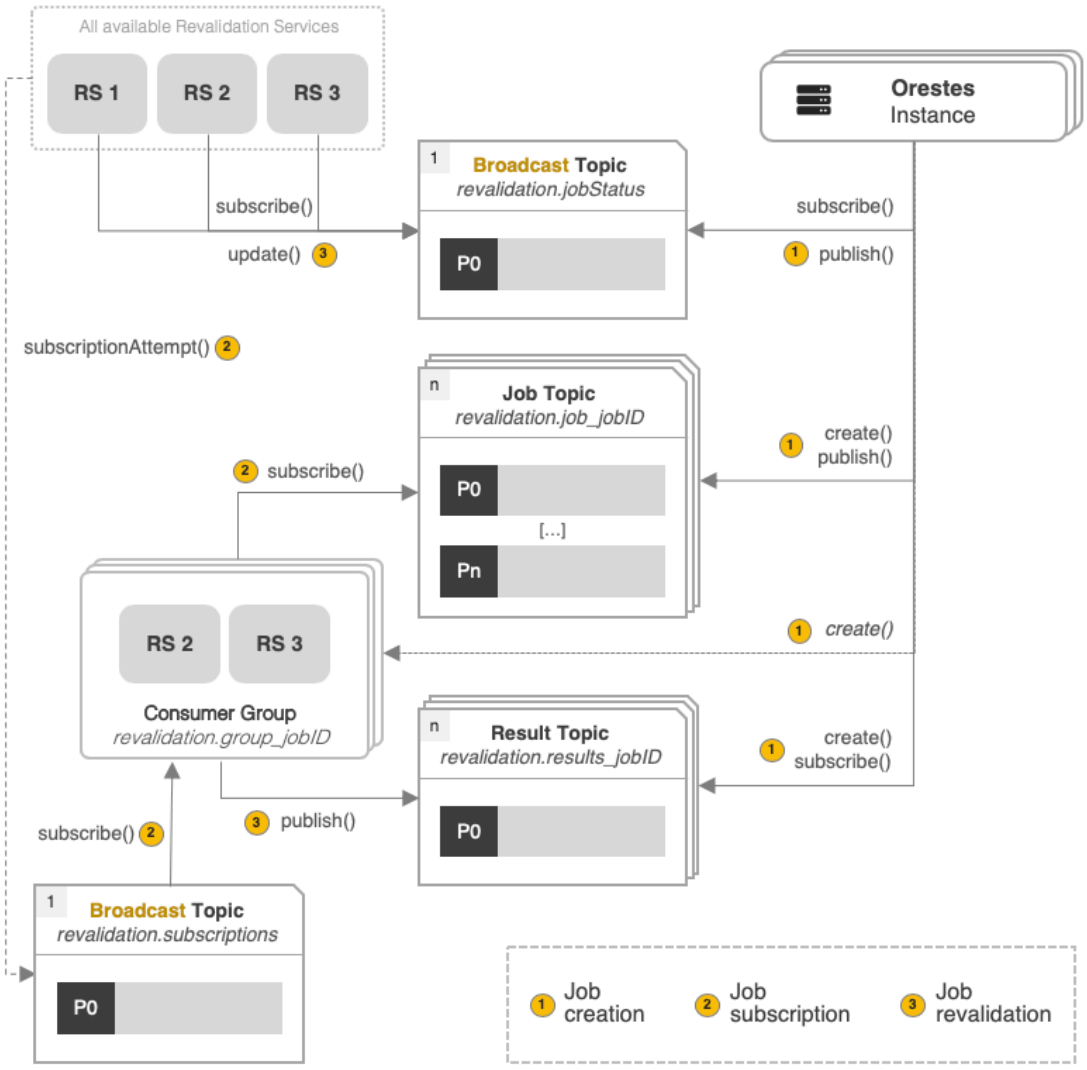

Abbildung 4.7: Gesamte Kommunikation über Kafka

4.3 Vorstellung der Gesamtlösung

Die einzelnen Teillösungen, die im Verlauf erarbeitet wurden, werden hier in der vorgestellten Gesamtlösung (Abbildung 4.7) zusammengefügt und präsentiert. Der Revalidierungsprozess kann in drei Schritte gegliedert werden: (1) Job Erstellung, (2) Job Subscription und (3) Job Revalidierung.

Schritt 1: Job Erstellung

ORESTES beginnt den Prozess mit dem Start eines definierten Revalidierungsauftrags. Ein Auftrag enthält bestimmte Attribute – unter anderem ID und Anzahl benötigter REVALIDATOR – und eine Auflistung zu prüfender Ressourcen des Origin Servers des Baqend Kunden. Die Attribute werden in einer neuen Job Status Nachricht im gleichnamigen Broadcast Topic in Kafka veröffentlicht. Das zugehörige Datenobjekt ist in Abbildung 4.8 dargestellt. Gleichzeitig erstellt ORESTES zwei neue Kafka Topics. Das Topic für die Übertragung der Ressourcen wird mit einer variablen Anzahl Partitionen erstellt. Dies dient der Möglichkeit der Lastverteilung auf verschiedene Revalidierungsservices, wie im Verlauf bereits verdeutlicht wurde. Das Topic für den Rücktransport der Ergebnisse der Revalidierung an ORESTES enthält hingegen nur eine einzelne Partition. Dieses Topic wird von ORESTES als Konsument abonniert. Die Ressourcen werden in Datenpakete zu je 100 unterteilt. Im zugehörigen Datenobjekt (Abbildung 4.9) ist erkennbar, dass neben der URL auch weitere Attribute – wie ein eTag – für die einzelnen Ressourcen übermittelt werden.

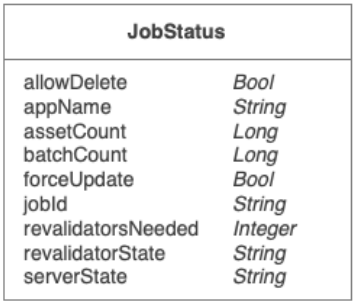

Abbildung 4.8: Job Status Datenobjekt

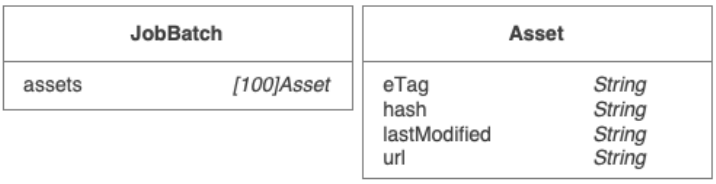

Abbildung 4.9: Job Paket Datenobjekt

Schritt 2: Job Subscription

Alle verfügbaren Microservices abonnieren die Nachrichten im Job Status Broadcast Topic und lesen den neuen Revalidierungsauftrag von ORESTES. Führt ein Service aktuell keine andere Revalidierung durch – es besteht also Kapazität für die Durchführung eines neuen Auftrags – so versucht er sich für diesen Job zu registrieren. In Kapitel 4.2 wurde der Prozess der Subscription schon erklärt. Es wird sichergestellt, dass nur maximal so viele Services eine Revalidierung eines Auftrags durchführen können, wie es im entsprechenden Attribut definiert ist. Nachdem sich ein Service für die Revalidierung eines Auftrags erfolgreich eintragen konnte, abonniert er als Teil einer Consumer Group das von ORESTES erstellte Job Topic für diesen Auftrag. Die Consumer Group ermöglicht es, dass die Datenpakete – die auf verschiedene Partitionen verteilt gespeichert werden – von mehreren Services gleichzeitig abgearbeitet werden können, sollte dies erforderlich sein.

Schritt 3: Job Revalidierung

Der letzte Schritt beschreibt die eigentliche Durchführung der Revalidierung und wird bei der Beschreibung der Implementierung des Microservice in Kapitel 5.3noch ausführlich erklärt. An dieser Stelle reicht zu erwähnen, dass die REVALIDATOR, die einen Auftrag durchführen, nach und nach die Nachrichten und damit die Datenpakete von ORESTES empfangen. Für jede enthaltene Ressource wird die Revalidierung durchgeführt. Die Ergebnisse aller Ressourcen eines Pakets werden in einem Ergebnis Objekt (Abbildung 4.10) zusammengefasst und im Ergebnis Topic veröffentlicht. ORESTES wartet dort als Konsument auf die Ergebnisse und empfängt und verarbeitet sie.

JobResult	
averageResponse	*Integer*
results	*[]Result*
revalidatorId	*String*

Result	
asset	*Asset*
isDeleted	*Bool*
serverResponse	*Integer*
serverError	*Bool*
wasStale	*Bool*

Abbildung 4.10: Job Ergebnis Datenobjekt

4.4 Einsatz im Betrieb

Auch wenn der konkrete Betrieb von Kafka über den eigentlichen Umfang dieser Abarbeitung hinaus geht, ist es dennoch für ein korrektes Verständnis wichtig, ihn hier theoretisch zu beschreiben. Die Darstellung des Gesamtumfelds von Baqend inklusive ORESTES und SPEED KIT wurde in Abbildung 4.11 um die neuen Komponenten Apache Kafka und REVALIDATOR ergänzt. Die Ausführung von Kafka würde bei Baqend, wie die anderen Systeme auch im Amazon Web Services Umfeld ermöglicht werden. Anders als andere Messaging-Systeme schreibt Kafka eingehende Nachrichten der Topics direkt in den persistenten Speicher und behält diese Daten nicht im Arbeitsspeicher. Dies hat den Vorteil, dass weniger Arbeitsspeicher benötigt wird, vor allem aber, dass im Falle eines Systemausfalls alle Daten ohne Probleme wiederhergestellt werden können.[29] Die vorgestellte Lösung sieht es vor, dass die übermittelten Daten und die dazugehörigen Topics eines Revalidierungsauftrags nach der endgültigen Bearbeitung von ORESTES gelöscht werden, weil sie nicht länger benötigt werden. Damit werden diese Daten nur während der Bearbeitung gespeichert – es wird insgesamt nicht viel Festplattenspeicher benötigt. Sollte die Löschung der Daten fehlschlagen, werden sie dennoch automatisch gelöscht. Kafka speichert Daten nach einer definierten Vorhaltezeit, danach werden sie automatisch aus dem Cluster entfernt.

[29] Confluent (2019b).

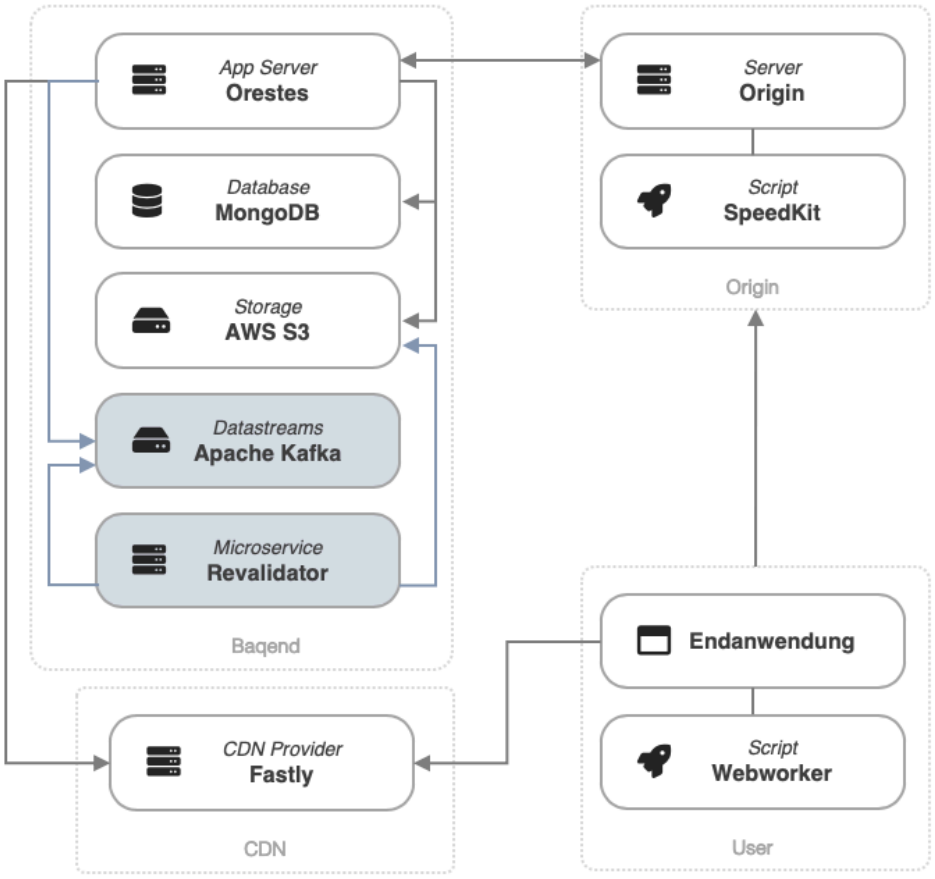

Abbildung 4.11: Einsatz von Kafka im Betrieb bei Baqend

Die eingesetzten Broker in einem Kafka Cluster übernehmen die Verwaltung der gespeicherten Daten. Die Verwaltung der Broker selber übernimmt ein weiteres System: ZooKeeper. Der Aufbau eines Clusters kann Abbildung 4.12 entnommen werden. Im produktiven Einsatz ist die Anzahl der beiden Komponenten relevant. Die Anzahl der Broker ist entscheidend für die Replizierung der Daten im gesamten Cluster. Für Testzwecke kann Kafka zwar mit einem einzelnen Broker ausgeführt werden, im Betrieb würde man jedoch mehrere Instanzen einsetzen. Die gespeicherten Daten werden dabei so auf die Broker verteilt, dass sie möglichst effizient verwaltet werden können, gleichzeitig beim Ausfall eines Brokers aber wiederhergestellt werden können. In diesem Fall würden alle restlichen Broker die Verwaltung der Daten übernehmen. Meldet sich der verlorene Knoten nach einem Neustart im Cluster, werden alle Daten erneut zwischen allen Brokern aufgeteilt. Für kleine bis mittlere Systeme empfiehlt sich

der Einsatz von je drei ZooKeeper und Kafka Brokern.[30] So werden möglichst wenige Hardwareressourcen benötigt, gleichzeitig aber eine Ausfallsicherheit erstellt, die eine hohe Verfügbarkeit des Kafka Clusters ermöglichen soll.

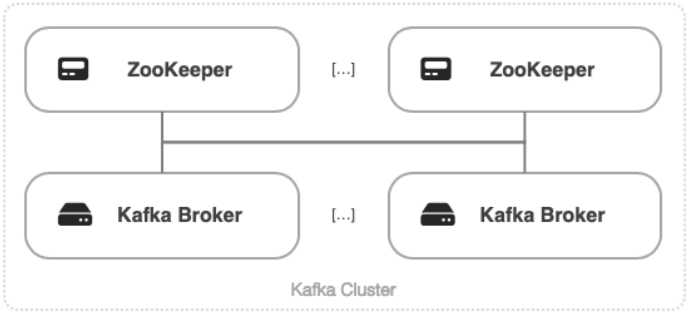

Abbildung 4.12: Aufbau eines Kafka Clusters[31]

Sollte Kafka bei Baqend in Zukunft auch in anderen Bereichen Anwendung finden, kann die bestehende Konfiguration auch für neue Vorhaben verwendet werden. Zusätzliche Broker sind jederzeit ohne Probleme in das System integrierbar.

4.5 Zusammenfassung

Zwei Hauptprobleme bei der bestehenden Revalidierungsroutine in ORESTES aus Kapitel 3.2betrafen Defizite im Fehlverhalten von ORESTES und den generellen Ressourcenverbrauch. Die vorgeschlagene Lösung erforderte die Auslagerung der Routine in einen Microservice. Dies bedingt gleichzeitig die Implementierung einer Schnittstelle zwischen dem neuen Microservice und ORESTES. Apache Kafka wurde als Möglichkeit vorgestellt, mit der große Datenmengen zwischen verschiedenen rollenbasierten Applikationen vermittelt werden können. Mit der vorgeschlagenen Implementierung ist es möglich, dass ORESTES Revalidierungsaufträge auf eine variable Anzahl REVALIDATOR verteilen kann. Kafka übernimmt dabei neben der Nachrichtenübermittlung auch die Rolle eines API-Gateways und leitet Anfragen an verfügbare Microservices weiter.

[30] Bromhead (2018).
[31] Cloudurable (2017).

Theoretisch sind die gewonnenen Erkenntnisse nicht auf den Einsatz von Kafka beschränkt. Mit kleinen Anpassungen wäre die Umsetzung auch mit anderen Messaging-Systemen denkbar. Kafka stellte sich in Tests jedoch als stabiles und ressourcenarmes System heraus, was in der Evaluation in Kapitel 6 noch verdeutlicht werden soll. Allerdings besticht Kafka ohne zusätzliche Konfiguration durch einfache Skalierbarkeit, Persistenz der Daten und vor allem durch Zuverlässigkeit bei der Datenübertragung. Letztere ist vor allem hervorzuheben, weil durch den gespeicherten Offset der Konsumenten eine erfolgreiche Übertragung garantiert werden kann.

Mit der Vorstellung von Broadcast Topics wurde eine Möglichkeit gefunden, wie Applikationen – gleichzeitig in Konsumenten- und Produzentenrolle – sich über Kafka organisieren können. Durch Kafkas Eigenschaft Nachrichten in der gleichen Reihenfolge zu speichern und sie auszugeben können sich Microservices für Revalidierungsaufträge ohne Doppelung registrieren. Vorausgesetzt wird, dass diese Organisation durch die Services selbstständig stattfindet, es gibt keine andere Kontrollinstanz. Der Mechanismus funktioniert jedoch vollkommen autonom und kann auch auf Ausfälle der Services reagieren. Vorausgesetzt sie starten nach einem Ausfall automatisch neu, repariert sich das gesamte System selbst.

Das Gesamtproblem – die Revalidierung von einer Auflistung von Ressourcen – wird vor der Übertragung in kleine Datenpakete verteilt. Dadurch wird nicht nur die Übertragung zu Kafka effizienter gestaltet, sondern auch die Möglichkeit zur gleichzeitigen Revalidierung durch mehrere Microservices geschaffen. Die Ergebnisse der Revalidierung finden über separate Kafka Topics den Weg zurück zur korrekten ORESTES Instanz, wo sie abschließend behandelt werden können.

5

Implementierung eines Microservices

5.1 Auswahl der Programmiersprache

Bevor mit der Konzeptionierung des Revalidierungs-Microservice begonnen werden kann, muss zuerst die Wahl einer geeigneten Programmiersprache getroffen werden. An dieser Stelle wird auf einen Vergleich verschiedener Alternativen verzichtet. Vielmehr sollen die funktionalen Anforderungen an die vorgestellte Anwendung die Auswahl beeinflussen. Nach der Erarbeitung der zu erfüllenden Kriterien folgt die Vorstellung der getroffenen Auswahl.

Anwendungen, die als Microservices gelten, werden durch bestimmte Charakteristiken definiert und werden beschrieben als ein „cohesive, independent process interacting via messages“[32]. Services werden zur Erfüllung eines einzelnen kleinen Teilproblems implementiert und unterscheiden sich darin von klassischen monolithischen Systemen. Die bearbeiteten Aufgaben sind oft so allgemeingültig, dass Microservices von vielen verschiedenen Anwendungen gleichzeitig verwendet werden können.[33] Im Kontext der vorliegenden Arbeit soll der neue REVALIDATOR eine Auflistung von URLs von ORESTES erhalten. Über HTTP werden diese vom Origin Server eines Baqend Kundens angefragt. Der Vergleich eines neuen generierten Hashwerts des Rückgabewerts mit einem bereits gespeicherten ermöglicht die Revalidierung des Caches von SPEED KIT. Bestimmte wünschenswerte Eigenschaften von Microservices im Allgemeinen sind auch für diesen Kontext relevant. Im Folgenden werden Kriterien von Adrian

[32] Dragoni et al. (2017), S. 2.
[33] Nadareishvili et al. (2016), S. 6.

Mouat[34] verwendet, die die Auswahl einer passenden Programmiersprache beeinflussen.

Eine erstrebenswerte Eigenschaft von Microservices ist eine interne Zustandslosigkeit. Dabei soll vor allem auf die Speicherung von Daten innerhalb eines Service verzichtet werden. Daten werden nur für die Verwendung von externen Datenquellen bezogen und nach Beendigung dort wieder abgelegt. Der Revalidierungsservice wird während einer Revalidierung einen bereits beschriebenen Zustand führen. Bei einem Ausfall muss jedoch eine andere Instanz mit der Revalidierung an der gleichen Stelle fortfahren können, damit der Auftrag korrekt bearbeitet werden kann. Im generellen sollen Ausfälle natürlich vermieden werden, indem auf Verhalten im Fehlerfall korrekt reagiert wird. Damit ist allerdings nicht auszuschließen, dass eine Anwendung dennoch unerwünscht beendet wird. Verlorene Instanzen sollen sich selbstständig wiederherstellen können, indem sie schnellstmöglich neustarten und mit der Revalidierung eines Auftrags fortfahren können. Es ist davon auszugehen, dass Baqend im Betrieb mehrere Instanzen des neuen Service einsetzen wird. Mit Hilfe von Cloud-Computing kann der Hardwareverbrauch taktgenau abgerechnet werden. Aus betriebswirtschaftlichem Interesse soll die gewählte Programmiersprache in Bezug auf Hardwarenutzung daher effizient sein.

Mouat beschreibt weiterhin, dass man auf den Einsatz VM-basierter (virtuelle Maschine) Programmiersprachen verzichten soll. Die Hauptargumente gegen eine Nutzung von beispielsweise Java liegt im hohen Arbeitsspeicherbedarf und einem langsamen Programmstart. Damit ist Java für die Verwendung des REVALIDATOR ausgeschlossen, auch wenn die Nutzung im Umfeld von Baqend naheliegend gewesen wäre, da unter anderem ORESTES darin implementiert wurde. Der zu implementierende Service wird mit ORESTES über Kafka kommunizieren. Eine Programmiersprache sollte

[34] Mouat (2019).

daher eine einfache Anbindung an das Messaging-System ermöglichen. Viele vorgefertigte Bibliotheken von Kafka sind für einzelne Sprachen verfügbar, unterscheiden sich allerdings in der konkreten Implementierung und Anwendbarkeit.

Auswahl der Programmiersprache

Unter Berücksichtigung der erarbeiteten Anforderungen an eine Programmiersprache soll der Revalidierungsservice in Go (oder GoLang) implementiert werden. Go wird als Open Source Projekt von Google seit 2012 entwickelt und stetig aktualisiert. Anwendungen, die in Go implementiert werden, sind sehr leichtgewichtig und zeichnen sich durch eine gute Skalierbarkeit aus. Kompilierte Anwendungen werden in Binärcode überführt. Die Ausführung ist damit hardwarenah ohne VM möglich und beinhaltet alle Abhängigkeiten zu eingebundenen Bibliotheken.[35] Viele Standardbibliotheken sind verfügbar, unter anderem eine benötigte Implementierung von HTTP. Durch einen effizienten Garbage Collector verwenden Anwendungen in Go möglichst wenig Arbeitsspeicher.

Go zeichnet sich weiterhin mit einer effizienten Implementierung von Nebenläufigkeit aus. In sogenannten „Goroutines" kann die Ausführung einer Anwendung auf alle verfügbaren Prozessoren verteilt werden. Wohingegen Java für jeden eröffneten Thread 1MB Heap Memory allokiert, sind es bei Go pro Goroutine nur 2KB.[36] Für Goroutinen gibt es zusätzlich einfache Möglichkeiten untereinander zu kommunizieren. Die konkrete Verteilung der Routinen auf die verfügbare Hardware wird effizient in der Laufzeit einer Anwendung durchgeführt. Für Go sind ebenfalls mehrere aktuelle implementierte Kafka Schnittstellen verfügbar, die für dieses Projekt genutzt werden können.

[35] Nilsson (o. J.).
[36] Patel (2017).

5.2 Konzeptionierung des Microservice

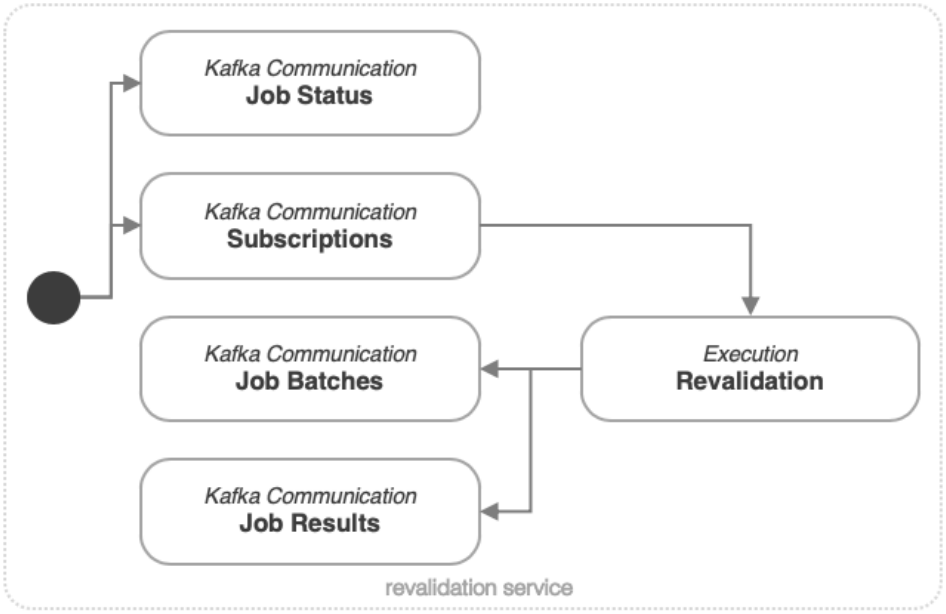

Abbildung 5.1: Modularer Aufbau des Revalidierungsservice

Der neue Revalidierungsservice wird Aufträge von ORESTES erhalten, die es zu bearbeiten gilt. Das gesamte Aufgabengebiet lässt sich in Teilgebiete unterteilen, die gleichermaßen modular implementiert werden können (Abbildung 5.1). Die Kommunikation mit Hilfe von Kafka stellt an verschiedenen Stellen eine wichtige Rolle dar. Nur über einen korrekten Austausch mit ORESTES und anderen Instanzen des REVALIDATORS kann sichergestellt werden, dass die Applikation die gewünschten Ergebnisse liefert. Eine Schnittstelle zu Kafka ließe sich anhand umfangreicher Dokumentation selber vornehmen. Unter anderem stellt Apache für jede Rolle – hier Consumer und Producer – eine eigene API zur Verfügung.[37] Eine eigene Implementierung würde sich allerdings aufwändig gestalten. Gleichzeitig gibt es viele verfügbare Bibliotheken, die

[37] The Apache Software Foundation (2017a).

die gewünschten Funktionalitäten bereits erfüllen. Die zwei meistbenutzten Kafka Bibliotheken für Go sind „kafka-go“ von Segment[38] und „confluent-kafka-go“ von Confluent[39]. Beide Implementierungen haben in ihrer genauen Anwendung kleine Unterschiede. Dadurch, dass sie die von Apache festgelegten APIs für Kafka verwenden, ist die konkrete Funktionalität die gleiche. Für diese Ausarbeitung wird kafka-go von Segment verwendet, weil das zugehörige GitHub Repository über häufigere Aktivitäten in Form von Aktualisierungen besticht.

Job Status Verarbeitung und Subscription

Mit der Erstellung eines neuen Revalidierungsauftrags veröffentlicht ORESTES im *Job Status* Topic eine neue Nachricht anhand des Datenmodells aus Abbildung 4.8. Eine Nachricht enthält Meta-Informationen zu einem Auftrag, die der Microservice zur Revalidierung benötigt. Die enthaltene Auftragsnummer definiert zudem, in welchen Topics die zu revalidierenden Datenpakete veröffentlicht und die Ergebnisse erwartet werden. Es gibt keine kontrollierende Instanz, die einen Auftrag eindeutig einem freien Service zuweist. Vielmehr müssen die einzelnen Microservices dieses Topic durchgehend überwachen, um auf Statusänderungen reagieren und neue Aufträge für eine Revalidierung selbstständig finden zu können. Für jeden Auftrag gibt es zwei Status Attribute – je für ORESTES und Microservice. Diese Attribute geben den Fortschritt eines Auftrags an, werden aber auch zur Fehlerbehandlung verwendet. Sowohl ORESTES als auch Service können beispielsweise einen Auftrag aus unterschiedlichen Gründen abbrechen. Für jede Statusänderung wird eine aktualisierte Nachricht im Job Status Topic veröffentlicht.

[38] Segment.io (2019).
[39] Confluent (2019a).

Im Broadcast Topic „*Subscriptions*“ tauschen sich die Microservices untereinander aus, um sicherzustellen, dass nicht mehr Instanzen gleichzeitig an einem Auftrag arbeiten, wie dies definiert wurde. Der genaue Mechanismus wurde bereits in Kapitel 4.2 beschrieben.

Durchführung der Revalidierung

Nachdem ein Service selbständig einen freien Auftrag gefunden und die Registrierung durchgeführt hat, kann die eigentliche Revalidierung beginnen. Für jeden Job wird ein eigenes Kafka Topic erstellt, in das die einzelnen Datenpakete veröffentlicht werden. Jedes Datenpaket enthält bis zu 100 Ressourcen. Für jede Ressource wird eine URL und weitere Metadaten, wie Hashwert und eTag, abgelegt. Auch wenn der Ablauf an verschiedenen Stellen optimiert werden soll, muss das Ergebnis der Revalidierung in jedem Fall identisch mit der bisherigen Implementierung aus ORESTES sein.

Der Microservice bezieht nach und nach die einzelnen Datenpakete aus dem entsprechenden Topic. Für jede URL einer Ressource wird eine HTTP Anfrage an den entsprechenden Origin Server eines Baqend Kunden gestellt. Vom Ergebniswert wird ein Hashwert berechnet und mit dem bisherigen gespeicherten verglichen. Unterscheiden sie sich nicht, dann ist die im Cache gespeicherte Ressource weiterhin aktuell. Sind die beiden Hashwerte jedoch unterschiedlich, gibt es eine neue Version der Ressource und muss im Cache aktualisiert werden. Baqend speichert alle Ressourcen in einem Amazon Web Services Bucket, der vom Microservice in diesem Fall aktualisiert werden muss. Die weiteren Metadaten, die für jede Ressource gespeichert werden, werden auch weiterhin von ORESTES verwaltet. Diese Daten haben auch Einfluss auf andere Komponenten in Baqends System, weshalb der Microservice die Verwaltung nicht übernehmen soll. Alle Ergebnisse der Revalidierungen eines Datenpakets werden in einem Ergebnispaket zusammengefasst und in einem entsprechenden Kafka Topic veröffentlicht. ORESTES empfängt diese Nachrichten und kann die weitere Bearbeitung vornehmen.

ORESTES erlaubt weitere Einstellungen für einen Revalidierungsauftrag, die vom neuen Service berücksichtigt werden müssen. Das Attribut *allowDelete* zum Beispiel gibt an, ob Ressourcen nach einer Revalidierung aus dem Cache komplett, teilweise oder überhaupt nicht gelöscht werden dürfen.

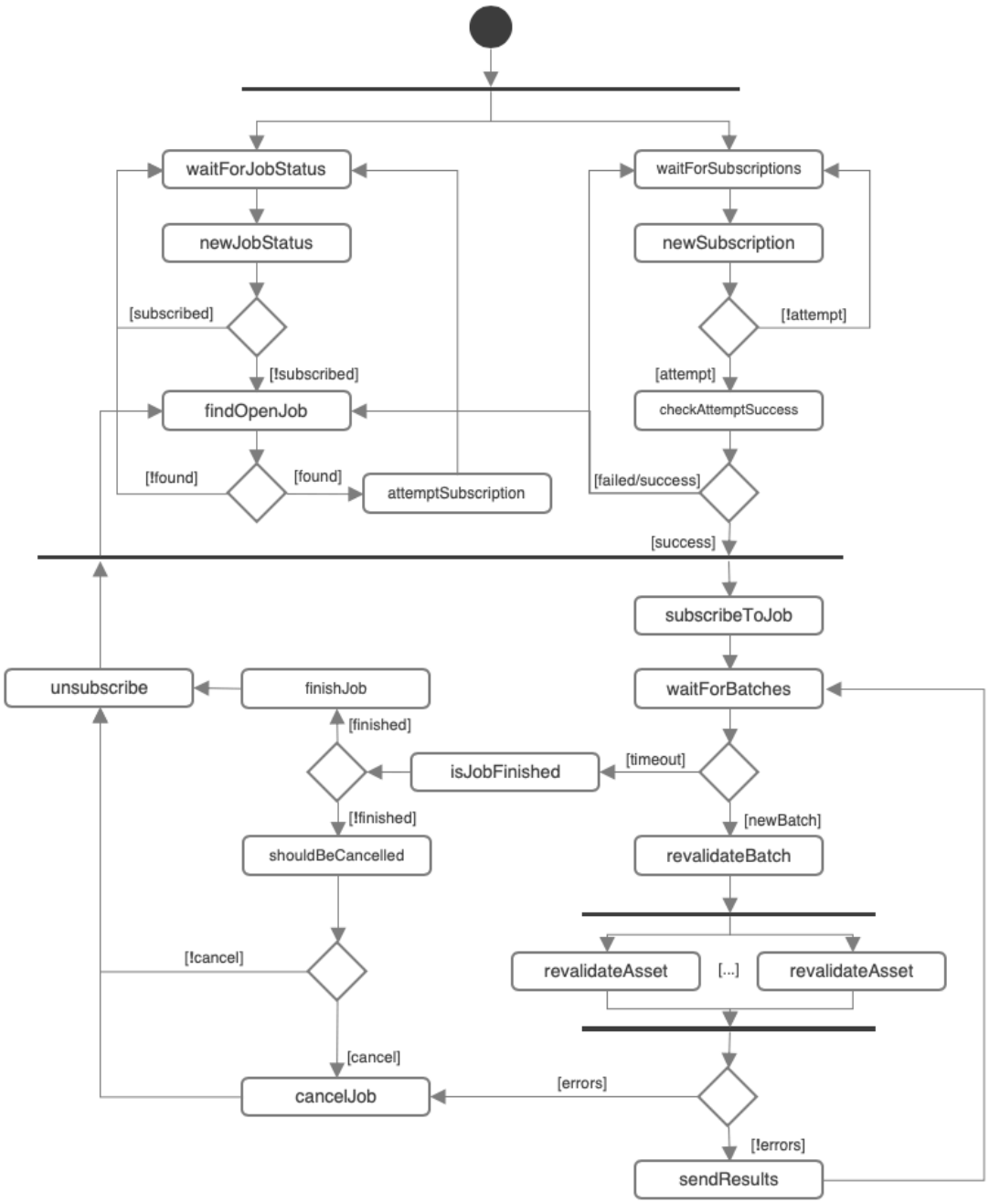

Abbildung 5.2: Zustandsdiagramm des Revalidierungsservice

5.3 Umsetzung der Anforderungen

Abbildung 5.2 stellt als Zustandsdiagramm den gesamten Programmablauf des Microservice dar. Dort ist ebenfalls zu sehen, dass kein Endzustand eingezeichnet ist. Planmäßig läuft der REVALIDATOR ohne Unterbrechung. Abstürze sollen zwar verhindert werden, sind aber theoretisch in jedem Zustand möglich. Vielmehr wird der Microservice extern beendet, sollte er nicht weiter benötigt werden. Mit dem Start der Applikation abonniert der Service gleichzeitig die beiden Broadcast Topics „*Subscriptions*“ und „*JobStatus*“ als Kafka Konsument. Die Verwaltung der dort eintreffenden Nachrichten geschieht nebenläufig und wird zu jeder Zeit durchgeführt. Mit jedem Erhalt einer Nachricht werden Schritte angestoßen, die den weiteren Programmverlauf einleiten. Job Status und Subscriptions werden jeweils in eigenen internen Datenstrukturen abgelegt. Der Programmablauf lässt sich in die Module Subscription-Verwaltung, Job-Status-Verwaltung, Revalidierung und Fehlerbehandlung unterteilen, die im Verlauf in gleicher Untergliederung vorgestellt werden.

Verwaltung der Subscriptions

In einem Attribut kann ORESTES bestimmen, wie viele REVALIDATOR maximal gleichzeitig einen gemeinsamen Auftrag bearbeiten dürfen. Damit können umfangreiche Aufträge je nach Auslastung des Gesamtsystems schneller durchgeführt werden. Die unterschiedlichen Instanzen des Microservice tauschen sich eigenständig untereinander aus, um dies zu erfüllen. Bevor ein REVALIDATOR einen Auftrag bearbeiten kann, muss er sich zuvor mit allen anderen Instanzen abstimmen. Diese Logik wurde bereits in Kapitel 4.3umfangreich vorgestellt. Im Subscription Topic gibt ein REVALIDATOR seinen aktuellen Registrierungszustand an. Sein Initialzustand („available“) erreicht er auch nach Beendigung eines Auftrags wieder. Eine Nachricht in diesem Topic enthält die Zuordnung von Revalidator ID, Auftrags ID und dem entsprechenden Zustand. Plant ein Service *S1* beispielsweise die Übernahme von Auftrag *A1* lautet das 3-Tupel:

{„S1“, „A1“, „attempt“}

Quellcode 5-1 zeigt die konkrete Implementierung dieser Logik. Beim Empfang einer neuen Subscription Nachricht (`newJobSubscriptionMessage`) wird die interne Datenstruktur aktualisiert. Jeder Microservice speichert nur die Zuordnung der Instanzen zu einem Auftrag. Geht ein REVALIDATOR wieder in seinen Initialzustand („available“) wird er aus dem internen Speicher gelöscht. Enthält eine empfangene Nachricht die Auftragsnummer, bei der der Microservice gerade eine Registrierung beabsichtigt, erfolgt die weitere Bearbeitung (`checkSubscriptionAttempt`) der Nachricht. Es gilt zu prüfen, ob sein Registrierungsversuch schnell genug erfolgte oder ob andere Instanzen schneller waren und er einen anderen Auftrag suchen muss. Aufbauend auf obigem Beispiel wird in dieser Routine die Anzahl der erfolgreichen Registrierungen und die Anzahl der Registrierungsversuche zu *A1* ermittelt. Mit der folgenden Formel wird die Kapazität eines Auftrags festgestellt:

Kapazität, wenn: Anzahl Registrierungen + Anzahl Versuche < Benötigte Revalidator

Bei Feststellung verfügbarer Kapazität kann der Service die eigentliche Revalidierung beginnen, seine Registrierung war also erfolgreich. Er veröffentlicht dann das aktualisierte 3-Tupel im Broadcast Topic:

{„S1“, „A1“, „subscribed“}

Im anderen Fall bricht er seinen Registrierungsversuch ab – veröffentlicht seinen Initialzustand – und sucht nach einem anderen verfügbaren Auftrag:

{„S1“, null, „available“}

```
// newJobSubscriptionMessage Save all incoming messages in internal data structure.
func newJobSubscriptionMessage(jobSubsriptionMessage []byte) {

    [...]

    if jobSubscription.SubscriptionType == "available" {
        deleteFromSavedSubscriptions(jobSubscription.RevalidatorID)
    } else {
        savedSubscriptions = append(savedSubscriptions, jobSubscription)
    }

    if subscriptionAttempt == jobSubscription.JobID {
        checkSubscriptionAttempt(jobSubscription.JobID)
    }
}

// checkSubscriptionAttempt Current service is in `attempt` to subscribe to `Job`. Check
 if it was quick enough, otherwise cancel the try.
func checkSubscriptionAttempt(jobID string) {
    subscribed := 0
    attempts := 0

    for _, subscription := range savedSubscriptions {
        if subscription.JobID == jobID {
            switch subscription.SubscriptionType {
            case "subscribed":
                subscribed++
            case "attempt":
                attempts++
                if subscription.RevalidatorID == revalidatorID {
                    if subscribed + attempts <= savedJobs[jobID].RevalidatorsNeeded {
                        sendJobSubscription(jobID, "subscribed")
                        go subscribeToJob(savedJobs[jobID])
                        subscriptionAttempt = ""
                        return
                    }
                    sendJobSubscription("null", "available")
                    log.Infof("Could not subscribe quick enough for `Job` %s. Maybe next
 time!", jobID)
                    subscriptionAttempt = ""
                    return
                }
            }
        }
    }
}
```

Quellcode 5-1: Job Subscription

Finden eines offenen Auftrags

In einer Liste speichert jeder REVALIDATOR ein Datenobjekt (Abbildung 4.8) für jeden verfügbaren Auftrag. Für einen neuen Auftrag, den ORESTES im Job Status Topic veröffentlicht, wird ein Objekt in dieser Liste angelegt. Erfolgen dann Aktualisierungen, wird das entsprechende Objekt nur noch angepasst. Nach jedem Empfang einer neuen Job Status Nachricht prüft der Microservice, ob er schon eine Revalidierung durchführt oder ob er eine Registrierung versucht. Ist beides nicht der Fall, sucht er einen verfügbaren Job. Dafür iteriert er seine gespeicherte Liste aller Aufträge und prüft, ob er irgendwo eine Revalidierung vornehmen kann. Gesucht wird der älteste Auftrag, der nicht beendet oder abgebrochen ist und noch Kapazität für einen weiteren REVALIDATOR aufweist. Konnte ein Auftrag gefunden werden, der diese Kriterien erfüllt, wird die Registrierung versucht, in dem er eine entsprechende Nachricht im Subscriptions Topic veröffentlicht. Diesen Zustand behält er so lange, bis entweder Erfolg oder Misserfolg seines Versuchs feststellen konnte, wie im Verlauf bereits beschrieben wurde.

Verarbeitung von Datenpaketen

Nach erfolgreicher Registrierung eines Auftrags kann die Revalidierung begonnen werden. Der Microservice abonniert als Konsument das Kafka Topic, über das ORESTES die zu revalidierenden Datenpakete veröffentlicht. Als Mitglied einer gemeinsamen Consumer Group teilen sich alle Microservices eines Auftrags den Arbeitsaufwand. So bearbeiten die Mitglieder Datenpakete nicht doppelt. Mit dem Empfang eines neuen Datenpakets beginnt die Revalidierung. In den definierten Anforderungen in Kapitel 3.3wurde festgelegt, dass der Microservice auf die Antwortgeschwindigkeit des ORIGIN Servers reagieren soll. Bei der bisherigen Umsetzung werden maximal sechs gleichzeitige Anfragen an einen Server gesendet. Dies soll die generelle Erreichbarkeit nicht beeinträchtigen. Jedoch wurde herausgestellt, dass die Anzahl gleichzeitiger Anfragen eine große Auswirkung auf die Bearbeitungszeit eines Revalidierungsauftrags hat. Aus

diesem Grund soll die Antwortgeschwindigkeit eines Webservers die Anzahl der gleichzeitigen Anfragen beeinflussen. Dieser Mechanismus zur Feststellung gleichzeitiger Anfragen wird im Folgenden noch genauer vorgestellt.

Quellcode 5-2 zeigt die Verarbeitung eines Datenpakets. Die eigentliche Revalidierung wird von einem `revalidationWorker` durchgeführt. Ein Worker ist eine Methode, die als Goroutine nebenläufig ausgeführt wird. Jede Methode führt so lange die Revalidierung einzelner Ressourcen durch, bis das Datenpaket komplett bearbeitet wurde. Die Anzahl der gleichzeitigen Anfragen an einen Server definiert, wie viele Worker gestartet werden. Die Ressourcen eines Datenpakets – bis zu 100 – werden in einen Pool (`assetChannel`) geschrieben. Alle Worker können auf diesen Pool zugreifen und sich nach und nach Ressourcen zur Revalidierung herausnehmen. Mit Hilfe der Pools kann vermieden werden, dass die Assets gleichmäßig auf die Worker verteilt werden müssen. Langsame Worker verzögern damit nicht die Bearbeitungszeit eines Pakets. Die Methode `RevalidateBatch` wird so lange blockiert (`resultWaitGroup.Wait()`), bis die letzte Ressource aus dem Pool von einem Worker bearbeitet wurde. Die Goroutinen werden dadurch alle wieder geschlossen und die benutzten Hardwareressourcen freigegeben. Die Worker haben während der Revalidierung die Ergebnisse in einem weiteren gemeinsamen Pool (`resultChannel`) veröffentlicht. Diese Ergebnisse werden nun in einem neuen Datenpaket zusammengefasst und in Kafka veröffentlicht.

```
func revalidateBatch(chunk *messagetypes.JobBatch, job *messagetypes.JobStatus) {
    assetChannel := make(chan messagetypes.RevalidationAsset, len(chunk.Assets))
    resultChannel := make(chan messagetypes.RevalidationResult, len(chunk.Assets))
    var resultWaitGroup sync.WaitGroup

    for worker := 1; worker <= concurrentRequests; worker++ {
        go revalidationWorker(worker, job.AllowDelete, job.ForceUpdate, assetChannel,
        resultChannel, &resultWaitGroup)
    }

    for _, asset := range chunk.Assets {
        assetChannel <- asset
        resultWaitGroup.Add(1)
    }

    close(assetChannel)
    resultWaitGroup.Wait()
    close(resultChannel)

    [...]

    sendRevalidationResult(&jobResult, job)
}
```

Quellcode 5-2: Verarbeitung von Datenpaketen

Durchführung der Revalidierung

In der Methode `revalidateAsset` (Quellcode 5-3) stellt jeder bereits beschriebenen Worker eine HTTP Anfrage an die entsprechende URL einer Ressource. Je nach Antwort des Servers fällt das Ergebnis unterschiedlich aus. Falls aus einer früheren Revalidierung verfügbar, können in entsprechenden Headern die Attribute „eTag“ und „lastModified“ übergeben werden. Dies kann die Anfrage beschleunigen und die Datenmenge der Antwort reduzieren, weil der angefragte Server bereits feststellen kann, ob die Ressource aktualisiert wurde oder nicht. Die Übergabe der Attribute kann allerdings mit dem Attribut „forceUpdate“ eines Auftrags überschrieben sein, falls dies in ORESTES so definiert wurde. In diesem Fall werden die Header nicht übermittelt, auch wenn die entsprechenden Attribute vorliegen. Die Ressourcen werden dann in jedem Fall vom Server erneut abgefragt.

Der Statuscode, der bei jeder HTTP Antwort mitgeliefert wird, definiert den weiteren Verlauf der Methode. Konnte der Server mit dem übermittelten „eTag“ oder „lastModified“ feststellen, dass die gespeicherte Ressource seitdem nicht modifiziert wurde, lautet der Statuscode „304 Not Modified“. Als Ergebnis der Revalidierung wird dementsprechend nur zurückgegeben, dass die Ressource im Cache nicht abgelaufen ist: `WasStale: false`.

Befindet sich der Statuscode zwischen 400 und 499, liegt ein Clientfehler vor. In den meisten Fällen gibt der Server 404 zurück. In diesem Fall kann die angefragte Ressource am Server nicht gefunden werden und kann sicher aus dem Cache entfernt werden. Dafür löscht der Microservice den entsprechenden Eintrag im AWS S3 Bucket. Im Ergebnis der Revalidierung wird dann angegeben, dass die Ressource abgelaufen ist und bereits durch den Service gelöscht wurde.

Wird keiner der bereits beschriebenen Statuscode abhängigen Fälle ausgelöst, bedeutet dies, dass der Server zumindest keine Änderung der angefragten Ressource feststellen konnte. In diesem Fall wird vom `Body` der Serverantwort ein Hashwert mit Hilfe des CRC32 Verfahrens berechnet. Der berechnete Hashwert wird mit dem Wert aus der letzten Revalidierung der Ressource verglichen. Unterscheiden sie sich nicht, entspricht der Ergebniswert dem vom 304 Statuscode. Sind sie jedoch unterschiedlich, muss die Ressource im Cache aktualisiert werden. Dafür wird der entsprechende Eintrag im AWS S3 Bucket mit der Serverantwort aktualisiert. Etwaige neue eTag oder LastModified Werte werden dann im Ergebnis der Revalidierung an ORESTES zurückgegeben, wo die Verwaltung dieser Attribute in einer Datenbank vorgenommen werden.

```
func revalidateAsset(asset messagetypes.RevalidationAsset, allowDelete bool, forceUpdate bool)
messagetypes.RevalidationResult {
    body, response, responseTime, err := httpservice.HTTPGetRequest(asset, forceUpdate)
    if err != nil {
        log.Error("Error while making HTTP request: ", err)
        return messagetypes.RevalidationResult{WasStale: true, Asset: asset}
    }

    switch {
    case response.StatusCode == 304:
        return messagetypes.RevalidationResult{WasStale: false, Asset: asset,
       ServerResponseTime: responseTime}

    case response.StatusCode >= 400 && response.StatusCode < 500 && allowDelete:
       [...] // Delete in S3 Bucket
       return messagetypes.RevalidationResult{WasStale: true, IsDeleted: true, Asset: asset,
       ServerResponseTime: responseTime}

    case response.StatusCode >= 500:
        [...] // Handle Server Errors
        return messagetypes.RevalidationResult{Asset: asset, ServerResponseTime: responseTime,
       ServerError: true}
    }

    newHash := common.CreateCRC32HashString(body)

    if common.CompareHash(asset.Hash, newHash) == 0 {
        return messagetypes.RevalidationResult{WasStale: false, Asset: asset,
       ServerResponseTime: responseTime}
    }
    [...] // Update in S3 Bucket
    asset.ETag = response.Header.Get("ETag")
    asset.Hash = newHash
    return messagetypes.RevalidationResult{WasStale: true, Asset: asset, ServerResponseTime:
       responseTime}
}
```

Quellcode 5-3: Durchführung der Revalidierung

Ermittlung maximaler gleichzeitiger Anfragen

ORESTES kann mit der Verteilung eines umfangreichen Revalidierungsauftrags auf mehrere Microservices die Bearbeitungszeit verkürzen. In Kapitel 3.2wurde zudem verdeutlicht wie diese Zeit durch die gleichzeitigen Verbindungen an einen Server beeinflusst wird. Die Anforderungen definieren deshalb, dass die Antwortgeschwindig-

keit eines Servers die Anzahl paralleler Anfragen beeinflussen soll. Bislang stellt ORESTES sechs Anfragen an einen Server, unabhängig ob die Antwort schnell oder langsam erfolgt. In der neuen Implementierung soll ein Server, der besonders schnell auf Anfragen reagiert, mehr als sechs Anfragen erhalten, damit ein Auftrag schneller bearbeitet werden kann.

Es gibt zahlreiche Techniken, wie Anfragen serverseitig limitiert werden können. Darunter fällt zum Beispiel die Idee des „Leaky Buckets“: Erhält ein Server mehr Anfragen als definiert (sein Eimer ist in dieser Verbildlichung voll), dann werden alle weiteren Anfragen abgewiesen.[40] Diese Techniken lassen sich jedoch wenig bis gar nicht auf den erforderlichen Mechanismus übertragen. Eine ähnliche gewünschte Limitierung der Anfragerate für einen Client beschreiben jedoch Shkapenyuk und Suel mit ihrem „Domain-Based-Throttling“ für Web Crawler.[41] Die Autoren erreichen dies jedoch nicht durch Überwachung der Antwortrate, sondern durch Verteilung der Anfragen auf Zeitpunkte in der Zukunft. Dies stellt für den REVALIDATOR keine Option dar, weil Aufträge möglichst schnell bearbeitet werden sollen. Aus diesem Grund wird im Folgenden eine eigene Lösung vorgeschlagen, mit der die gleichzeitigen Serverzugriffe maximiert werden sollen.

Es wurde festgelegt, dass jeder REVALIDATOR seine eigene Anfragerate ermittelt. Es wäre allerdings durchaus denkbar, einen ermittelten Wert zwischen allen Serviceinstanzen eines gemeinsamen Auftrags zu synchronisieren. Dies erhöht jedoch die Komplexität der Ausarbeitung. Gleichzeitig sind Szenarien vorstellbar, in denen die Antwortzeit des Servers je nach anfragender Serviceinstanz unterschiedlich ausfallen könnte. Aus diesem Grund soll jeder Service die Anfragerate abhängig von der Serverantwortzeit selbstständig ermitteln. An dieser Stelle beschreibt die Serverantwortzeit nicht die Gesamtzeit der Übertragung. Diese Zeit wäre abhängig von der Datenmenge, die angefragt wird. Vielmehr wird hier die Zeit gemessen, die verstreicht bis die erste

[40] Google (2019).
[41] Shkapenyuk/Suel (2002).

Reaktion auf eine Anfrage an einen Server wieder beim Microservice eingetroffen ist.[42] Diese Zeit ist abhängig von Netzwerkengpässen und Auslastung des Servers und sollte bei unterschiedlichen Ressourcen ungefähr gleich groß ausfallen.

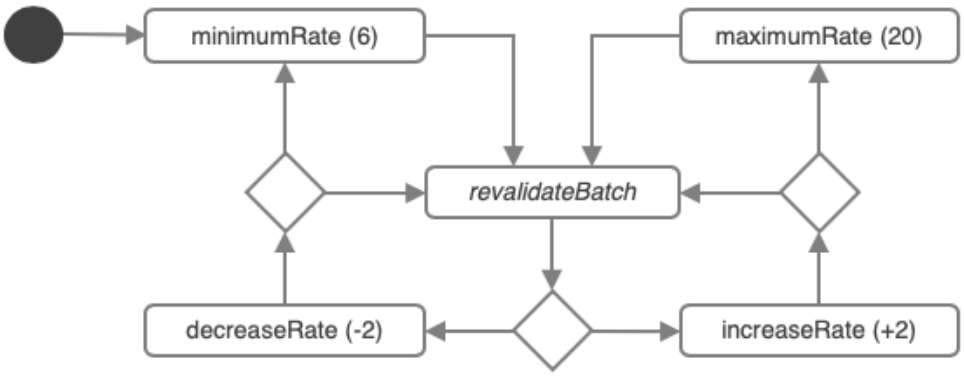

Abbildung 5.3: Mechanismus zur Feststellung gleichzeitiger Serverzugriffe

In Abbildung 5.3 ist der vorgestellte Mechanismus dargestellt. Das erste Datenpaket eines neuen Auftrags wird mit einem Startwert – sechs parallele Anfragen – revalidiert. Dieser Wert hat sich bei ORESTES über Jahre als guter Durchschnittswert herausgestellt und soll übernommen werden. Bei jeder der 100 abgefragten Ressourcen eines Datenpakets wird die Antwortrate des Servers ermittelt. Daraus kann die durchschnittliche Antwortzeit eines gesamten Datenpakets in Millisekunden berechnet werden. Überschreitet sie einen definierten Schwellenwert nicht, kann die Anzahl gleichzeitiger Verbindungen erhöht werden. Eine Erhöhung bedeutet, dass bei der Revalidierung des nächsten Datenpakets des gleichen Auftrags zwei weitere `Worker` – also zwei weitere parallele Verbindungen zum Server – ausgeführt werden. Bei diesem nächsten Datenpaket wird ebenfalls wieder die durchschnittliche Antwortzeit ermittelt. Ab dem zweiten Paket wird diese Zeit allerdings mit dem Wert des vorherigen Durchgangs verglichen. So kann festgestellt werden, welche Auswirkungen die Anpassung der Anfragerate auf den Server hatte. Ist sie ungefähr gleichgeblieben oder hat sich um maximal 100 Millisekunden verschlechtert, werden im nächsten Durchgang zwei wei-

[42] Google (o. J.).

tere Verbindungen aufgebaut. Hat sich die Zeit jedoch um mindestens 100 Millisekunden verschlechtert, werden die parallelen Verbindungen für das nächste Datenpaket wieder um zwei reduziert. Messungenauigkeiten können hierbei mit dem 200 Millisekunden Spektrum zwischen Erhöhung und Verminderung der Verbindungen ausgeglichen werden.

Der Startwert stellt auch gleichzeitig den Minimalwert des Mechanismus dar. In jedem Fall wird ein Service – unabhängig davon wie langsam ein Server antwortet – mindestens sechs parallele Anfragen stellen, damit ein Auftrag nicht unendlich lang bearbeitet wird. Gleichzeitig stellt 20 den Maximalwert des Mechanismus dar. Unabhängig davon wie schnell der Server antwortet, soll dieser Wert nicht überschritten werden. Der Maximalwert ist im produktiven Einsatz stark von Hardware- und vor allem Netzwerkkapazität des Microservice abhängig. Er soll verhindern, dass sich der REVALIDATOR selbst überlastet.

Eine effiziente Definition des genannten Schwellenwerts hat sich mangels fehlender Erfahrungswerte oder Literatur als schwierig herausgestellt. LittleData hat bei der Analyse von über 3.500 Webseiten eine durchschnittliche Antwortzeit von ca. 550 Millisekunden ermitteln können.[43] Google beschreibt eine Antwortzeit über 200 Millisekunden als „ausbaufähig".[44] Der vorgestellte Mechanismus nutzt den Schwellenwert, um herauszufinden, ob die Anfragerate nach der Revalidierung des ersten Datenpakets erhöht werden kann oder nicht. Deswegen wird dieser Wert pro Revalidierung nur einmal verwendet. Zudem pendelt sich die Anfragerate nach ein paar Durchgängen des Mechanismus automatisch ein. Aus diesem Grund wird der Schwellenwert aufbauend auf den Erkenntnissen der genannten Quellen hier mit 350 Millisekunden festgelegt.

[43] Littledata (2019).
[44] Google (o. J.).

5.4 Fehlerbehandlung

Der grundlegende Programmablauf wurde im Verlauf nun umfangreich beschrieben. Bei ersten Tests haben sich jedoch Engpässe und Fehler in der Implementierung ergeben, die an dieser Stelle verdeutlicht werden sollen. Darauf aufbauend werden Mechanismen vorgestellt, die Fehlverhalten in erster Linie vorbeugen sollen.

Zustandsverwaltung zur Kontrolle einer Revalidierung

Bei der Übermittlung der Ressourcen eines Auftrags veröffentlicht ORESTES fortlaufend Datenpakete im entsprechenden Kafka Topic. Je nachdem wie viele Pakete ein Auftrag enthält dauert dieser Prozess nur einige Sekunden. Er kann sich allerdings auch über mehrere Minuten erstrecken. Mit der Übernahme des Auftrags von einem Microservice konsumiert dieser nach und nach die Datenpakete und führt die Revalidierung durch. Denkbar ist, dass die Übertragung der Daten und die Abarbeitung gleichzeitig stattfinden. Es ist für den Microservice nicht ohne weiteres möglich zu erkennen, wann das letzte Datenpaket übermittelt wurde. Treffen zu einem Zeitpunkt keine weiteren Datenpakete mehr ein, muss der REVALIDATOR jedoch wissen, ob er noch auf weitere Datenpakete warten soll oder ob der Auftrag bereits komplett übertragen wurde. Es wäre möglich, dass ORESTES im Job Status die Anzahl der Datenpakete beschreibt. Ein Service kann dann ermitteln, wie viele Datenpakete zur Fertigstellung noch fehlen. Allerdings würde diese Methode bei der Verteilung auf mehrere Serviceinstanzen nicht funktionieren. Eine andere mögliche Umsetzung wäre, das ORESTES dem letzten Paket ein zusätzliches Attribut übergibt, dass den Abschluss eines Auftrags kenntlich macht. Bei Tests konnte jedoch sichtbar gemacht werden, warum dies in der Praxis nicht korrekt funktionieren kann. Abbildung 5.4 stellt ein mögliches Szenario dar: 14 Datenpakete werden durch Kafka automatisch auf zwei verfügbare Partitionen eines Topics verteilt. Das hervorgehobene letzte Datenpaket wurde jedoch in Partition 0 gespeichert. Es ist nicht sicher zu sagen, in welcher Reihenfolge Kafka die Daten an einen Microservice ausgibt. Unter Umständen empfängt ein Service das „letzte“ Datenpaket

schon bevor Partition 1 komplett ausgeliefert wurde. Der Microservice würde das Ende des Auftrags zu früh erkennen und damit die Revalidierung vorzeitig abbrechen.

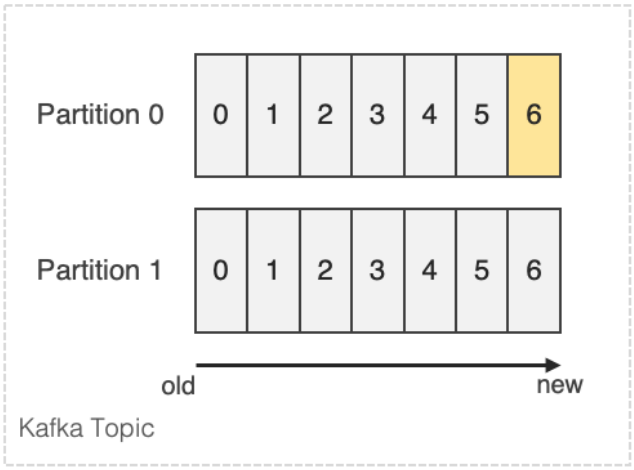

Abbildung 5.4: Darstellung des letzten Datenpakets eines Topics

Aus diesem Grund führt sowohl ORESTES als auch die Microservices einen eigenen Zustand für jeden Auftrag. Sie sind in Abbildung 5.5 in der Reihenfolge des Eintreffens abgebildet und werden in der vorgestellten Lösung beim *Job Status* im entsprechenden Kafka Topic gespeichert. Mit der Veröffentlichung eines neuen Auftrags erreicht ORESTES den ersten Zustand: „Transmitting“. Wenn ein Service die Revalidierung beginnt lautet sein Zustand „Revalidating“. Der Microservice kann über den Zustand von ORESTES feststellen, ob er noch weitere Datenpakete erwarten kann oder nicht. Erst wenn der Zustand zu „Wait-For-Results“ aktualisiert wurde, gibt ORESTES zu erkennen, dass alle Daten eines Auftrags übermittelt wurden. Mit der Revalidierung des letzten Datenpakets kann der Microservice die Beendigung entsprechend veröffentlichen. ORESTES hat noch einen weiteren Zustand, der beibehalten wird, während die übermittelten Ergebnisse noch verarbeitet werden. Damit kann im Fehlverhalten grob nachvollzogen werden, an welcher Stelle Fehler entstanden sind.

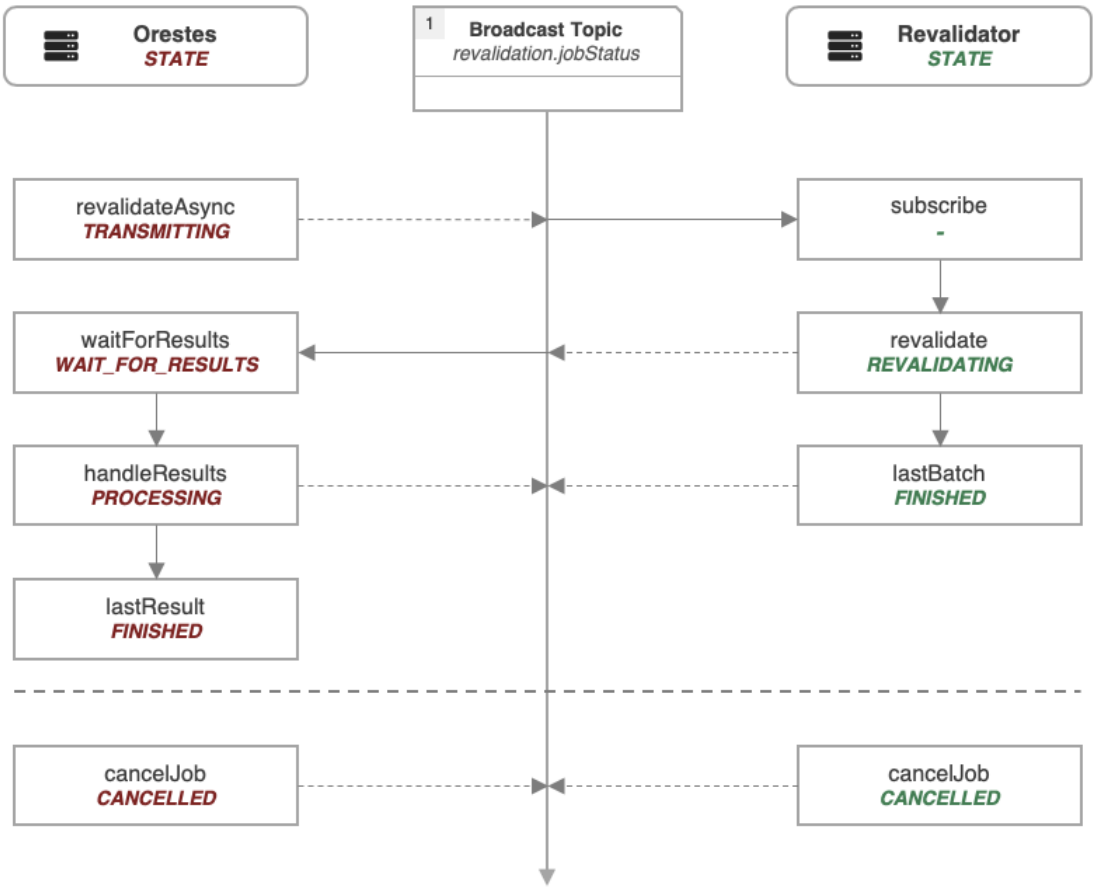

Abbildung 5.5: Zustände von Orestes und Revalidierungsservice

Probleme bei gleichzeitiger Bearbeitung mehrere Revalidator

Mit den vorgestellten Zuständen kann ein einzelner Microservice korrekt feststellen, wann das Ende einer Revalidierung erreicht wurde und kann den Auftrag entsprechend beenden. Abbildung 5.6 zeigt jedoch ein Szenario, bei dem zwei Microservices gleichzeitig einen gemeinsamen Auftrag bearbeiten. Hier hat ORESTES bereits alle 14 Datenpakete veröffentlicht, der Status lautet entsprechend „Wait-For-Results“. Kafka hat jedem Microservice eine Partition zugewiesen, wobei Revalidator 1 schon am Ende seiner Partition angekommen ist. Aufbauend auf dem Status von ORESTES stellt dieser nun das Ende des Auftrags fest, obwohl Revalidator 2 noch weitere Datenpakete zu bearbeiten hat. Der abgeleitete Mechanismus zur Fertigstellung eines Auftrags muss entsprechend angepasst werden: Ein Microservice stellt das Ende eines Auftrags fest,

wenn für eine bestimmte Zeit kein neues Datenpaket mehr im abonnierten Kafka Topic eintrifft, ORESTES das Ende der Übermittlung signalisiert hat und kein weiterer Microservice mehr an der Revalidierung beteiligt ist. Dies setzt voraus, dass Revalidator 1 im beschriebenen Beispiel die Revalidierung des Auftrags verlässt, ohne Änderungen an Zuständen vorzunehmen. Wenn nur ein Service pro Auftrag zugeteilt ist, kann außerdem sichergestellt werden, dass Kafka in jedem Fall alle verfügbaren Partitionen und damit etwaige nicht gelesenen Datenpakete an den verbliebenen Service ausspielt.

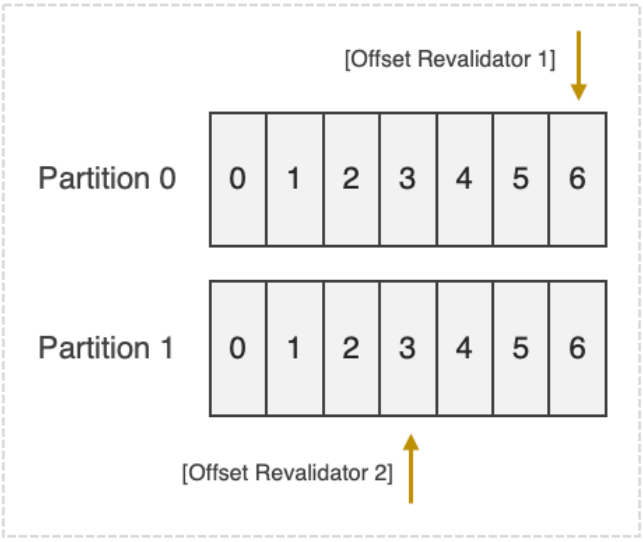

Abbildung 5.6: Problemstellung bei gleichzeitiger Bearbeitung mehrere Microservices

Es sind Szenarien denkbar, in denen ein Microservice einen Revalidierungsauftrag abbrechen kann. Wenn ein Server über mehrere Versuche und einen langen Zeitraum hinweg nicht erreichbar ist, soll der Auftrag storniert werden. Der Microservice soll dementsprechend den eigenen Zustand auf „Cancelled“ setzen. Etwaige andere beteiligte REVALIDATOR reagieren mit dem Abbruch der Revalidierung und ORESTES wartet nicht weiter auf den Empfang der Ergebnisse. Der Abbruch sollte auf jeden Fall durch eine Person überprüft werden, also entsprechend sichtbar dokumentiert werden. Ebenso kann ORESTES einen Auftrag vorzeitig beenden. Entweder geschieht dies

durch Nutzereingabe oder andere Fehlverhalten, die eine weitere Bearbeitung unmöglich machen. In diesem Fall sollen die Microservices den Auftrag nicht weiterbearbeiten.

Kontrollmechanismen für Fehlverhalten

Die bisherige Implementierung der Revalidierung in ORESTES überspringt Ressourcen, wenn ein Serverfehler vorliegt. Diese Art von Fehler gibt nicht in jedem Fall eine konkrete Auskunft über den Zustand einer Ressource. In der vorgestellten Lösung soll dies entsprechend verbessert werden. Alle fehlgeschlagenen Anfragen von Ressourcen eines Datenpakets werden vom REVALIDATOR zusammengefasst und im *Job Topic* erneut veröffentlicht. So wird die Revalidierung zu einem späteren Zeitpunkt erneut versucht. Jeder Ressource wird in diesem Fall allerdings ein Attribut vergeben, dass den ersten Versuch kennzeichnet. Damit kann vermieden werden, dass Ressourcen unendlich oft erneut angefragt werden, sollte sich die Fehlerart nicht ändern. Im wiederholten Fehlerfall wird die Ressource aus dem Cache entfernt, weil nicht sicher festgestellt werden kann, ob eine aktuellere Version vorliegt oder nicht. Ist ein Server komplett nicht erreichbar, kann der Microservice die Revalidierung des entsprechenden Auftrags temporär aussetzen. Er versucht dann einen anderen Auftrag zu finden.

Ausfall eines Microservice

Die Selbstorganisation der Services bringt weitere Schwierigkeiten mit sich. Fällt ein REVALIDATOR unvorhergesehen aus und kann den Absturz nicht veröffentlichen, gibt es keine Instanz, die das Fehlverhalten beobachten kann. War der Service einem Auftrag zugeordnet, so bleibt dieser in „Bearbeitung“, obwohl der Service eigentlich nicht mehr erreichbar ist. Die Anforderungen an den Microservice sahen jedoch vor, dass nach einem Absturz ein automatischer Neustart unternommen wird. Jeder Service hat zudem eine eindeutige ID, die ihn identifiziert. Nach dem Neustart wird der Microservice seinen Initialzustand „available“ wieder erreichen. Der Job ist damit wieder zur Bearbeitung frei.

5.5 Einsatz im Betrieb

An dieser Stelle soll, wie auch schon Kapitel 4.4 kein konkretes Betriebskonzept wiedergeben werden. Vielmehr sollen die theoretischen Grundlagen vermittelt werden, wie der neue Microservice in einer praktischen Umsetzung ausgeführt werden kann. Der Service wurde unter der Voraussetzung implementiert, möglichst wenig Hardwareressourcen zu benutzen. Go ermöglicht es beispielsweise zu entscheiden, ob Variablen als Wert oder Referenz übergeben werden. Es wurde versucht, alle Methoden dementsprechend speicherschonend zu implementieren. Baqend wird diesen Service bei Amazon Web Services ausführen. Aufbauend auf den Experimenten, die noch in Kapitel 6 beschrieben werden, konnten die Hardwareanforderungen abgeschätzt werden. Eine „t3.micro“ EC2 Instanz mit zwei CPU Kernen und einem Gigabyte Arbeitsspeicher sollten ausreichend sein für die Ausführung eines einzelnen Revalidierungsservice.[45] Sollte sich bei der Gewinnung von Erfahrungswerten jedoch herausstellen, dass bestimmte Baqend Kunden so leistungsfähige Webserver aufweisen, dass sie mehr als 20 parallele Verbindungen bearbeiten können, wären EC2 Instanzen mit mehr CPU Kernen wünschenswert. Je nach Konfiguration wäre diese Skalierung jedoch auch automatisch möglich. In diesem Fall würde AWS den CPU Engpass automatisch feststellen und entsprechende weitere Ressourcen zur Verfügung stellen.

Auch wenn die REVALIDATOR so konzipiert sind, dass sie autonom agieren und neustarten können, fehlt eine Applikation, die die Anzahl der Microservices verwalten kann. Sollten sich Revalidierungsaufträge häufen und die aktuelle Anzahl verfügbarer Microservices nicht ausreichen, um diese rechtzeitig zu bearbeiten, wäre ein Automatismus wünschenswert, mit dem neue Instanzen gestartet werden, wenn sie benötigt werden.

[45] Amazon Web Services (2019).

5.6 Zusammenfassung

Im vergangenen Kapitel wurde die Konzeptionierung und Implementierung eines neuen Microservice vorgestellt, der Revalidierungsaufträge von ORESTES bearbeiten soll. Aufbauend auf dem Ziel die Ausführung der Revalidierung von der Hardware von ORESTES zu trennen, sollte der Microservice als isoliertes System eigene Ressourcen verwenden. Alle ORESTES Server der Baqend Kunden teilen sich bei der vorgestellten Lösung die gleichen Microservices. Der Austausch zwischen den Applikationen erfolgt über Kafka. ORESTES veröffentlicht neue Revalidierungsaufträge dort, die die zu revalidierenden Ressourcen enthalten. Die unterschiedlichen Instanzen der Microservices, REVALIDATOR genannt, tauschen sich selbstständig untereinander aus, um festzulegen welche Instanz welchen Job revalidieren soll. Ein Microservice bezeichnet ein Architekturprinzip, das sich dadurch auszeichnet möglichst kleingewichtig zu sein und nur einen einzelnen Zweck erfüllt.

Auch wenn der eigentliche Zweck des Service recht schnell zu definieren und zu beschreiben ist, konnte im Verlauf dargestellt werden, welche Herausforderungen die Verteilung von Revalidierungsaufträgen mit sich bringt. An verschiedenen Stellen sind Mechanismen entwickelt worden, die eine effiziente Abarbeitung der Aufträge ermöglichen sollen. So wurde zum Beispiel eine Technik zur Maximierung der Abfragerate an einen Server vorgestellt. Der neue Microservice kann damit Aufträge möglichst schnell abarbeiten und sollte einen Leistungsvorteil gegenüber der bisherigen Implementierung in ORESTES aufweisen.

Es wurde zudem gezeigt, dass Go eine gute Grundlage für die Implementierung eines Microservice im gegebenen Kontext bietet. Zahlreiche Eigenschaften der Sprache bilden die wünschenswerten Eigenschaften einer solchen Applikation ab. Go besticht dabei vor allem durch eine effiziente Implementierung von Nebenläufigkeit, die bei der vorgestellten Anwendung sinnvoll eingesetzt werden kann. Abschließend wurde theoretische Betrieb des neuen Microservice bei Amazon Web Service vorgestellt.

6 Evaluation

In diesem Kapitel gilt es zu evaluieren, ob die gewünschten Anforderungen in der vorgestellten Microservice-Kafka-Lösung erreicht werden konnten. Die Auslagerung der Implementierung hat in jedem Fall die gleichnamigen Anforderungen ohne Prüfung erfüllt. In Versuchen soll festgestellt werden, ob die implementierte dynamische Abfragerate tatsächlich Revalidierungsaufträge schneller bearbeiten lässt. Im Zuge dieser Versuche kann gleichzeitig die generelle Fähigkeit zur Skalierung überwacht und der Hardwareaufwand abgeschätzt werden.

6.1 Versuchsaufbau

Für die vorgestellten Versuche wurde ein HTTP-Server in Node.js, aufbauend Express implementiert. Dieser lässt sich für die verschiedenen Tests frei konfigurieren, so dass unter anderem Varianzen in seiner Antwortgeschwindigkeit simuliert werden können. Gleichzeitig kann über variable Prozentsätze angegeben werden, wie viele Anfragen mit einem Server- oder Clientfehler beantwortet werden sollen. Beide Mechanismen sollen das Verhalten eines echten Servers im Kontext der Versuche annähernd abbilden. Dieser Server dient als Repräsentant eines Webservers von einem Baqend Kunden. Für den Test liefert der Server nur eine durchschnittlich große statische Ressource aus.

ORESTES wird in den Versuchen ebenfalls simuliert. Mit Hilfe einer Go Applikation werden Revalidierungsaufträge mit frei definierbarem Umfang in Kafka veröffentlicht. Diese Aufträge enthalten Datenpakete mit 100 Ressourcen, dabei stellt jede Ressource

die bereits genannte des Webservers dar. In den Tests werden bis zu drei REVALIDATOR Microservices eingesetzt, die ohne Änderung für die Tests verwendet werden können. Für Kafka wird jeweils eine einzelne ZooKeeper und Broker Instanz ausgeführt.

In Tabelle 2 sind die verwendeten Hardware Komponenten dargestellt, auf denen die verschiedenen Applikationen ausgeführt werden. Um gegenseitige Einschränkungen zu vermeiden, wird jede Anwendung auf einem eigenen Host ausgeführt. In den Versuchen wurden bis zu drei Go Microservices verwendet, die jeweils auf eigener Hardware ausgeführt wurden. Der beschriebene ORESTES Simulator wurde für die Übertragung der Revalidierungsaufträge auf demselben Host wie das Kafka Cluster ausgeführt. Die Ausführung beschränkte sich jedoch immer auf nur wenige Minuten und brauchte dabei wenig Hardwareressourcen.

	Go Revalidator (3 Hosts)	**Kafka Cluster**	**Node.js Webserver**
CPU	1,4 GHz 4 Cores, 4 Threads *ARM Cortex A53 (32 Bit)*	3,1 GHz 2 Cores, 4 Threads *Intel i5-7267U (64 Bit)*	3,4 GHz 4 Cores, 8 Threads *Intel Xeon E3-1231V3 (64 Bit)*
RAM	1 GB LPDDR2	16 GB LPDDR3	16 GB DDR3
Netzwerk	1 Gigabit Ethernet	1 Gigabit Ethernet	1 Gigabit Ethernet
OS	Raspbian Buster Lite	macOS 10.15.2	Windows 10 1909

Tabelle 2: Verwendete Hosts für die Evaluation

6.2 Dynamische Abfragerate

Für den Test der dynamischen Abfragerate wurde ein einzelner Go Microservice eingesetzt. Dieser sollte minimal sechs parallele Anfragen an den Testserver senden, konnte diesen Wert jedoch abhängig von der Antwortrate auf bis zu 20 erhöhen. Der Webserver wurde so konfiguriert, dass er seine Antwortgeschwindigkeit bremsen kann. Dafür hat er die Auslieferung der angefragten Ressource für eine definierte Dauer verzögert. Nach 100 Anfragen – also einem Datenpaket – hat sich diese Dauer jeweils

geändert. In diesem Versuch sollte überprüft werden, ob der Microservice korrekt auf die Antwortgeschwindkeit des Webservers reagiert und die Anfragerate dementsprechend nach Definition anpasst. Der Test umfasste 20 Datenpakete, also 2000 Anfragen an den Server. Für jedes Datenpaket wurde eine Antwortzeit definiert, die der Server einhalten soll. Daraus konnte abgeleitet werden, welche Anfragerate der Microservice theoretisch errechnen soll. Der erwartete Wert konnte dann mit den Ergebnissen des Experiments verglichen werden. Die gesamten Daten des Versuchs sind in Anhang 1 zu sehen. Die visuelle Darstellung der verwendeten Datengrundlage ist in Abbildung 6.1 erkennbar. Die Serverantwortzeit wird der entsprechenden erwarteten Antwortrate gegenübergestellt. Es ist zu erkennen, dass die Antwortrate nur zwischen den Begrenzungen von Minimal- und Maximalwert variiert. Gleichbleibende oder verringerte Antwortgeschwindigkeiten führen zum Anstieg der Anfragerate. Im anderen Fall wird diese wieder reduziert.

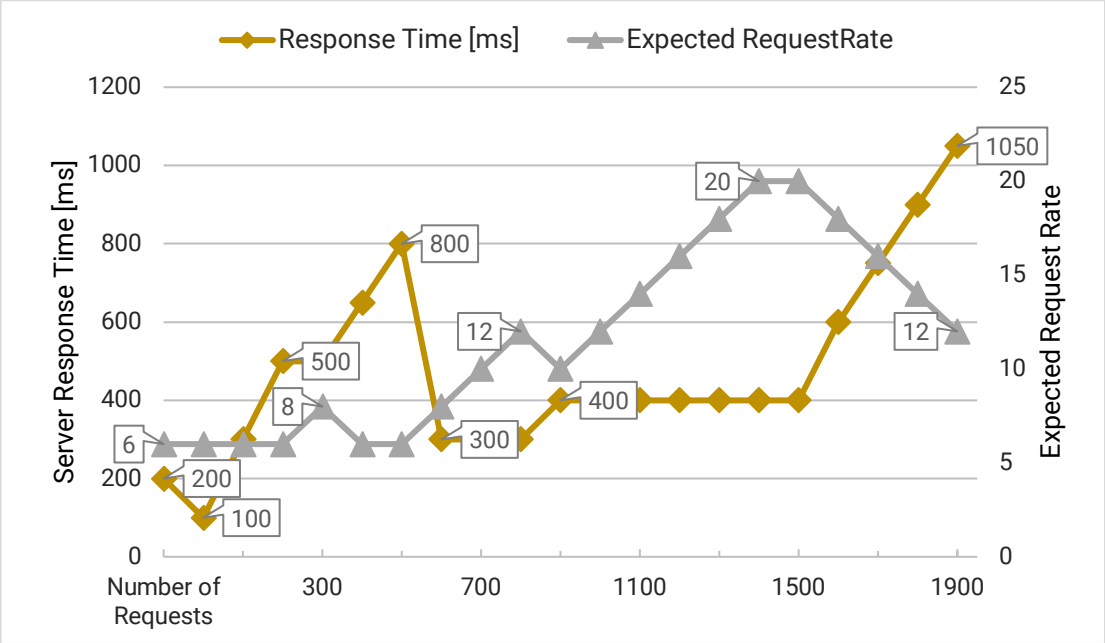

Abbildung 6.1: Versuchsaufbau für die dynamische Abfragerate

Beschleunigung der Bearbeitungszeit

Im Test hat der Microservice korrekt auf die Anforderungen reagieren können. Der Mechanismus zur Berechnung einer dynamischen Anfragerate konnte deswegen korrekt implementiert werden. Es gilt dennoch zu beweisen, dass dieser Mechanismus zur Beschleunigung der Bearbeitung von Revalidierungsaufträgen führen kann. Dies kann bereits durch ein einfaches Rechenbeispiel (Tabelle 3) erfolgen. Zur Grundlage wird ein Revalidierungsauftrag mit 100.000 Ressourcen genommen. Es wird für die Veranschaulichung angenommen, dass der Server durchschnittlich 300 Millisekunden für die Beantwortung benötigt. ORESTES würde den Aufwand mit sechs parallelen Anfragen bewältigen. Die errechnete Bearbeitungszeit beträgt ungefähr 84 Minuten. In diesem Beispiel sollen zwei Microservice Instanzen den gleichen Auftrag bearbeiten. Sie können die Anfragerate jeweils auf bis zu 50 parallele Verbindungen erhöhen, so dass bis zu 100 gleichzeitige Anfragen an den Server möglich sind. Die beiden Services würden den gleichen Aufwand in nur 5 Minuten bearbeiten können, also in sechs Prozent der Zeit, die ORESTES benötigt. In der Realität würde dies natürlich voraussetzen, dass der angefragte Webserver 100 parallele Anfragen ohne Einschränkung bearbeiten kann. Aus diesem Grund stellt dies nur den theoretischen Bestwert des Experiments dar. In jedem Fall kann abhängig von der Belastbarkeit eines Webservers mit der neuen Lösung die Bearbeitung der Revalidierungsaufträge schneller erfolgen, als bei der bisherigen Implementierung.

	Bisherige Revalidierung	**Go Microservice (2 Instanzen)**
Anzahl Ressourcen	100.000	100.000
Parallele Anfragen	6	bis 100
Bearbeitungszeit *Ø300ms Antwortzeit*	83,3 Minuten	Best Case: 5 Minuten

Tabelle 3: Rechenbeispiel Bearbeitungsdauer Revalidierung

6.3 Langzeittest

Das korrekte Verhalten des neuen Systems unter Belastung wurde in einem Langzeittest geprüft. Über einen Zeitraum von 12 Stunden wurden drei REVALIDATOR Instanzen gleichzeitig ausgeführt – jeder auf eigener Hardware. Der Webserver hat die Beantwortung seiner Anfragen in diesem Test unter bestimmten Bedingungen gebremst, um ein realistischeres Umfeld zu simulieren. Seine Bearbeitungszeit wurde zufällig um eine Dauer zwischen 100 und 600 Millisekunden verzögert. Die errechneten Verzögerungen wurden durch einen entsprechenden Algorithmus normalverteilt. Die erzeugten Varianzen wurden genutzt, damit die Microservices kontinuierlich ihre Anfrageraten anpassen mussten.

Die simulierte ORESTES Instanz in Go erzeugte in zufälligen Abständen (zwischen einer und fünf Minuten) neue Revalidierungsaufträge im Umfang zwischen 100 und 10.000 Ressourcen. Diese Aufträge konnten von einem bis drei REVALIDATOR Instanzen bearbeitet werden. Über den Zeitverlauf des Experiments wurden über 15.000 Datenpakete in über 700 Aufträgen in Kafka veröffentlicht. Jedes Datenpaket entsprach dabei einer Nachricht und für jeden Auftrag gab es zwei eigene Topics. Die Microservices veröffentlichten für jedes Datenpaket eine entsprechende Antwortnachricht in Kafka. In über 3.500 Nachrichten tauschten sie sich untereinander aus, um die Bearbeitung der Aufträge zu koordinieren.

Alle erstellten Aufträge konnten über den Versuchszeitraum erfolgreich und ohne Fehler bearbeitetet werden. Die Hardwareauslastung der Microservices wurde über den Versuchszeitraum aufgezeichnet. Arbeitsspeicher- und Prozessorauslastung für die ausgeführte Applikation unterschritten stark den erwarteten Wert. Jeder Microservice durfte bis zu 50 parallele Anfragen an den Webserver senden. Trotz des hohen Werts überschritt der Speicherbedarf nie 100 Megabyte. Die Prozessorauslastung lag häufig im einstelligen Prozentbereich, trotz der Nebenläufigkeit und der relativ schwachen Hardware, auf der der Service ausgeführt wurde. Hieraus lässt sich für den Einsatz im Betrieb ableiten, dass der Service auf sehr kleinen Hosts ausgeführt werden

kann. Gleichzeitig wäre auch denkbar normale Hosts zu verwenden, auf denen mehrere Microservice Instanzen gleichzeitig ausgeführt werden. In jedem Fall haben sich die gewünschten Eigenschaften von Go als Programmiersprache zur Implementierung eines Microservices im Testumfeld bestätigt.

Auch wenn der Test fast 40.000 Nachrichten in Kafka erzeugt hat, war die Hardwareauslastung des entsprechenden Systems im zu vernachlässigenden Bereich. Dies zeigt, dass das Nachrichtensystem für weitaus größere Datenmengen konzipiert wurde.

7 Ausblick

In der Evaluation konnte bestätigt werden, das mit Hilfe des neuen REVALIDATOR die Cache-Revalidierung im Umfeld von ORESTES optimiert werden kann. In diesem Kapitel sollen die logischen nächsten Schritte verdeutlicht werden. Die gewonnenen Erkenntnisse, vor allem in Bezug auf den Nachrichtenaustausch in und Einsatz von Apache Kafka, lassen sich auch auf andere Anwendungsgebiete bei Baqend sowie komplett andere Szenarien übertragen. Die Bearbeitung bestimmter Teilaufgaben ließ der beschränkte Umfang dieser Ausarbeitung nicht zu. Diese sollen im Folgenden ausblickend dargestellt werden.

7.1 Anwendbarkeit der Ergebnisse

Der implementierte Microservice bestach in Tests und der Evaluation mit hoher Zuverlässigkeit. Apache Kafka bietet großen Handlungsspielraum bezüglich der verarbeiteten Datenmenge und kann für den vorgesehenen Zweck bei Baqend eingesetzt werden. Dafür muss der konkrete Betrieb der beiden neuen Anwendungen noch fundierter behandelt werden. In den entsprechenden Kapiteln konnten bereits Empfehlungen ausgesprochen werden, wie die Ausführung in der Praxis theoretisch aussehen kann. Es wurden beispielsweise Hinweise zur Anzahl von Instanzen gegeben. In der Evaluation konnte dann der konkrete Hardwarebedarf abgeschätzt werden. Daraus lassen sich benötigte Cloudinstanzen bei Amazon Web Services ableiten, die für die Ausführung von REVALIDATOR und Apache Kafka gebucht werden müssen.

Die ermittelten Hardwareanforderungen für den Go Microservice fielen unerwartet gering aus. Eine Möglichkeit zur Ausführung in der Praxis sieht vor, mehrere Instanzen auf einem gemeinsamen Host gleichzeitig zu betrieben. Bei gewöhnlichen Microservices müssten entsprechende API-Endpunkte durch aufwendige Netzwerkkonfiguration erstellt werden. Kafka ermöglicht es, dass die Ausführung mehrerer Instanzen des gleichen Service vereinfacht wird. Ein Service braucht zum Betrieb nur die IP-Adressen der Kafka Broker. Beim Start registriert sich der REVALIDATOR dort. Ab diesem Zeitpunkt kann er Revalidierungsaufträge übernehmen. Jede weitere Instanz kann ohne zusätzliche Konfiguration ausgeführt werden.

7.2 Übertragung der Erkenntnisse

Das Grundkonzept von Apache Kafka sieht es vor, Konsumenten und Produzenten von Nachrichten voneinander zu entkoppeln. Im behandelten Fall bedeutet dies, dass ORESTES den genauen Empfänger seiner Revalidierungsaufträge nicht kennt. Es ist denkbar, dass es mehrere Ausprägungen von Konsumenten gleicher Nachrichtentypen geben kann. Bei der Entwicklung des Microservice erwies es sich oft als schwer den Überblick über alle laufenden REVALIDATOR Instanzen und Aufträge zu behalten. Aus diesem Grund wurde für Tests ein weiterer Konsument in Form einer Webanwendung entwickelt – das „Kafka Asset Revalidation Dashboard“, welches in Abbildung 7.1 zu erkennen ist. Ein Node.js Server sendet eine Vue.js Webwendung aus. Diese kann über eine Websocketverbindung fortlaufend Nachrichten mit dem Webserver austauschen. Der Node.js Server registriert sich als Konsument bei den Kafka Topics „*Job Status*“ und „*Subscriptions*“. Aus den Nachrichten der beiden Topics lässt sich ableiten, wie viele Instanzen des REVALIDATOR aktuell ausgeführt werden, welche Aufträge im System vorhanden sind und wie der entsprechenden Bearbeitungsfortschritt aussieht. Die verarbeiteten Informationen können in der Weboberfläche übersichtlich aufbereitet werden.

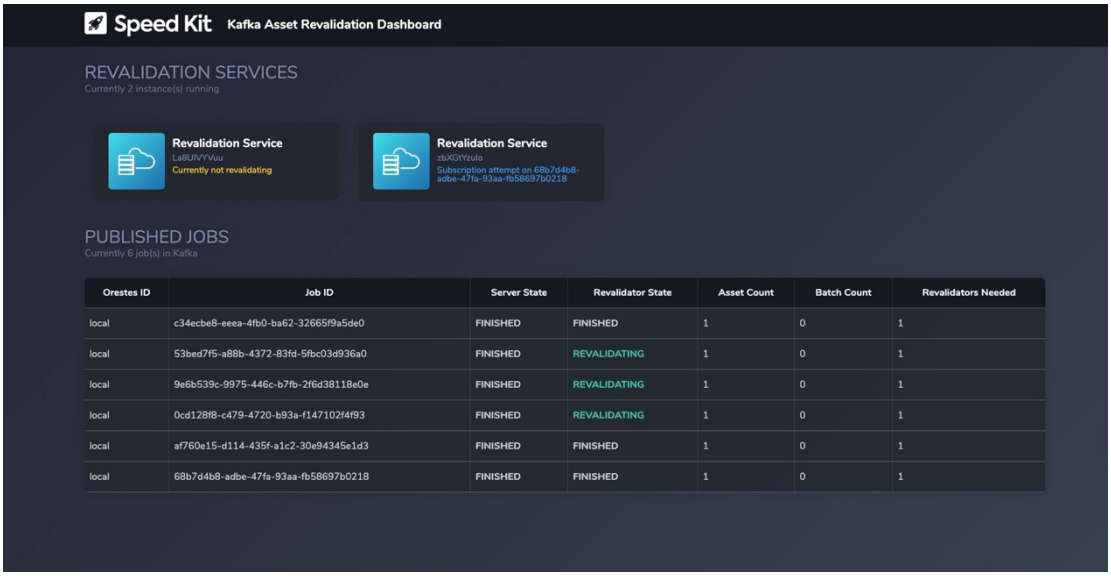

Abbildung 7.1: Beispiel zu Übertragung der Erkenntnisse: Kafka Asset Revalidation Dashboard

Das Dashboard ist ein gutes Beispiel für weitere Einsatzgebiete von Kafka bei Baqend. Aktuell gibt es bereits eine Möglichkeit den Fortschritt eines Revalidierungsauftrags zu beobachten. Dafür muss ORESTES jedoch regelmäßig einen Status aktualisieren. Mit dem Einsatz von Kafka wäre es möglich, dass die entsprechenden Anwendungen, die den Fortschritt überwachen wollen, selbstständig die Daten aus Kafka beziehen.

Natürlich lässt sich Kafka auch für andere Datenübertragungen einsetzen. Das System ist dafür ausgelegt äußerst umfangreiche Datenmengen zu verarbeiten. Die Evaluation konnte bereits zeigen, dass die Verarbeitung von Revalidierungsaufträgen alleine keine große Belastung für das Messagingsystem darstellt.

7.3 Transformation der Serverantworten

Die bisherige Implementierung der Cache-Revalidierung bei ORESTES verwendet in bestimmten Anwendungsfällen eine Transformation der empfangenen Serverantworten, um sie zu normieren. Für die Revalidierung einer Ressource wird von der erhaltenen Antwort des Webservers ein Hashwert gebildet und mit einem vorherigen verglichen. Es kommt zu Problemen bei bestimmten Webservern, die zum Beispiel im

Headbereich bei HTML-Dokumenten laufend unterschiedliche Tags einfügen. In diesem Fall kann es sein, dass eine Ressource – die sich nicht geändert hat – bei zwei aufeinanderfolgenden Anfragen unterschiedliche Ergebnisse in der Hashwertermittlung ergibt. Bei Baqend wird eine Node.js Anwendung eingesetzt, die die für den Inhalt irrelevanten Tags entfernt, um Ergebnisse vergleichbar zu machen. Diese Transformation wurde in den vorgestellten Microservice nicht implementiert. Die konkrete Umsetzung der Transformation sieht für jeden Baqend Kunden unterschiedlich aus und ist komplex einzubinden. Auch wenn die Implementierung in den neuen Microservice noch folgen soll, ist der sofortige Einsatz nicht gefährdet. Bei vielen Baqend Kunden ist die Transformation nicht notwendig. Diese Teilgruppe aller Kunden würde eine Erprobungsstufe für einen produktiven Einsatz ermöglichen.

8 Fazit

In dieser Ausarbeitung wurde ein System konzipiert und implementiert, das die Cache-Revalidierung von ORESTES effizient auslagern und beschleunigen soll. Zur Beantwortung der ersten Forschungsfrage konnten bei der Analyse der bestehenden Implementierung Limitationen herausgearbeitet werden, die die Leistungsfähigkeit aktuell einschränken. Äußerst umfangreiche Revalidierungsaufträge umfassen mehrere hunderttausend Ressourcen. Die Durchführung der Revalidierung ist verbunden mit der Abfrage aller betroffener Ressourcen eines Webservers. Der Ressourcenverbauch dieses Prozesses schränkt die generelle Belastbarkeit von ORESTES ein. Mehrere Kunden von Baqend teilen sich unter Umständen einen gemeinsamen Server. Die Revalidierung einer ORESTES Instanz kann sich deswegen auch negativ auf die Leistung einer anderen auswirken. Die abgeleitete Anforderung war, dass die Revalidierung auf eigener Hardware – losgelöst von ORESTES – durchgeführt werden soll. Weiterhin führen Defizite in der aktuellen Implementierung dazu, dass die Bearbeitung der Aufträge lange dauern kann und Fehlverhalten eines Webservers nicht korrekt behandelt wird.

Die gewünschte Auslagerung der Revalidierungslogik führte zur Konzipierung eines verteilten Systems. Die neue Anwendung muss auf einem neuen Weg Revalidierungsaufträge vom bestehenden ORESTES erhalten. Es wurde Apache Kafka vorgestellt: ein Messaging-System, das für die Verarbeitung großer Datenmengen konzipiert wurde. Mit Kafka sollte der Datenaustausch erreicht werden. Die neue Anwendung erfüllt nur den Zweck der Revalidierung und sollte von daher als Microservice implementiert werden.

In der Konzeption von Kafka im Kontext der vorgeschlagenen Lösung konnte mit dem Broadcast Topic ein Mechanismus vorgestellt werden, mit dem sich die Microservices autonom verwalten können, ohne ein weiteres externes System zu verwenden, wie es sonst üblich wäre. Kafka konnte in der verwendeten Situation Funktionalitäten eines API-Gateways abbilden. Nicht nur wird die Service-Discovery mit Kafka dargestellt, sondern auch die Datenübermittlung zwischen Produzenten und Konsumenten von Revalidierungsaufträgen ermöglicht.

Für den Revalidierungsservice wurde Go als mögliche Programmiersprache zur Implementierung argumentativ herausgearbeitet. Die leichtgewichtige Eigenschaft von Go eignet sich zur Umsetzung eines Microservice und ermöglicht den effizienten Einsatz von Nebenläufigkeit. Diese war für den REVALIDATOR besonders wichtig, weil er durch die vorgestellte dynamische Abfragerate Anfragen an den Server maximieren kann. Dadurch kann ein weiteres identifiziertes Problem des bestehenden Systems optimiert werden: Die Bearbeitungszeit umfangreicher Revalidierungsaufträge erstreckt sich über teilweise über einen ganzen Tag.

In der Evaluation konnte bewiesen werden, dass das vorgestellte System von Microservice und Apache Kafka über einen Prototypen hinausgeht und schon praxisnahe Szenarien erfolgreich bearbeiten konnte. Gleichzeitig überraschten die besonders geringen Hardwareanforderungen des neuen Microservice. Somit ist trotz der Leistungsfähigkeit die Ausführung auf günstigerer Hardware möglich, wodurch die Kosten für einen Einsatz im Betrieb auf einem geringen Niveau gehalten werden. Die zweite Forschungsfrage konnte deswegen in der Evaluation teilweise beantwortet werden. Die gewonnen Erkenntnisse von Broadcast Topic und dem Einsatz von Kafka im Umfeld von Baqend konnten schon mit dem in der Entwicklung implementierten Dashboard übertragen werden. Es konnte damit exemplarisch gezeigt werden, welche Möglichkeiten und Ansatzpunkte die vorgestellte Lösung bietet und welche Vorteile sie bei einem Einsatz weiterhin ermöglichen kann.

9 Literatur

Amazon Web Services, I. (2019): Amazon EC2 Instance Types. (https://aws.amazon.com/ec2/instance-types/ vom 13.12.2019).

Banavar, G./Chandra, T./Strom, R./et al. (1999): A case for message oriented middleware, In: International Symposium on Distributed Computing. O. O., 1–17.

Baqend GmbH (o. J. a): Refresh Policies. (http://www.baqend.com/guide/topics/speed-kit/refreshing/ vom 06.10.2019).

Baqend GmbH (o. J. b): Speed Kit. (https://www.baqend.com/speedkit.html vom 16.09.2019).

Bloom, B. H. (1970): Space/time trade-offs in hash coding with allowable errors, In: Communications of the ACM 13, H. 7, S. 422–426.

Bromhead, B. (2018): Apache Kafka: Ten Best Practices to Optimize Your Deployment. (https://www.infoq.com/articles/apache-kafka-best-practices-to-optimize-your-deployment/ vom 06.12.2019).

Cloudurable (2017): Kafka Architecture: Kafka Zookeeper Coordination. (http://cloudurable.com/blog/kafka-architecture/index.html vom 06.12.2019).

Confluent, I. (2019a): Confluent's Golang Client for Apache Kafka. (https://github.com/confluentinc/confluent-kafka-go vom 09.12.2019).

Confluent, I. (2019b): Running Kafka in Production. (https://docs.confluent.io/current/kafka/deployment.html vom 06.12.2019).

Dragoni, N./Giallorenzo, S./Lafuente, A. L./et al. (2017): Microservices: yesterday, today, and tomorrow, In: Present and ulterior software engineering. O. O., 195–216.

Flach, T./Dukkipati, N./Terzis, A./et al. (2013): Reducing web latency: the virtue of gentle aggression, In: ACM SIGCOMM Computer Communication Review. O. O., 159–170.

Foundation, A. S. (o. J.): HTTP Proxy Caching. (https://docs.trafficserver.apache.org/en/latest/admin-guide/configuration/cache-basics.en.html vom 30.06.2019).

Gessert, F. (2018): Low Latency for Cloud Data Management, O. O.

Google (2019): Rate-limiting strategies and techniques. (https://cloud.google.com/solutions/rate-limiting-strategies-techniques vom 13.12.2019).

Google (o. J.): Improve Server Response Time. (https://developers.google.com/speed/docs/insights/Server vom 13.12.2019).

HTTP Archive (2018): HTTP Archive. (https://httparchive.org/reports/state-of-the-web?start=2018_01_01&end=2019_01_01&view=list vom 12.06.2019).

Littledata (2019): What is the average server response time? (https://www.littledata.io/average/server-response-time vom 13.12.2019).

Mouat, A. (2019): Programming Languages for Microservices (and Containers). (https://blog.container-solutions.com/programming-languages-for-microservices-and-containers vom 07.12.2019).

Nadareishvili, I./Mitra, R./McLarty, M./et al. (2016): Microservice architecture: aligning principles, practices, and culture, O. O.

Najork, M./Heydon, A. (2002): High-performance web crawling, In: Handbook of massive data sets. O. O., 25–45.

Nilsson, S. (o. J.): Why Go? – Key advantages you may have overlooked. (https://yourbasic.org/golang/advantages-over-java-python/ vom 07.12.2019).

Offutt, J. (2002): Quality attributes of web software applications, In: IEEE software 19, H. 2, S. 25–32.

Padmanabhan, V. N./Mogul, J. C. (1995): Improving HTTP latency, In: Computer Networks and ISDN Systems 28, H. 1–2, S. 25–35.

Patel, K. (2017): Why should you learn Go? (https://medium.com/@kevalpatel2106/why-should-you-learn-go-f607681fad65 vom 07.12.2019).

Segment.io, I. (2019): kafka-go. (https://github.com/segmentio/kafka-go vom 09.12.2019).

Shkapenyuk, V./Suel, T. (2002): Design and implementation of a high-performance distributed web crawler, In: Proceedings 18th International Conference on Data Engineering. O. O., 357–368.

The Apache Software Foundation (2017a): Documentation. (https://kafka.apache.org/documentation vom 09.12.2019).

The Apache Software Foundation (2017b): Introduction. (https://kafka.apache.org/intro.html vom 04.12.2019).

Vakali, A./Pallis, G. (2003): Content Delivery Networks: Status and Trends, In: IEEE Internet Computing. doi: 10.1109/MIC.2003.1250586.

Valtonen, H. (2018): How Kafka Solves Common Microservice Communication Issues.

(https://dzone.com/articles/how-kafka-solves-common-microservice-communication vom 24.08.2019).

10 Anhang

Batch	Number of Requests	Response Time [ms]	Expected Request Rate
0	0	200	6
1	100	100	6
2	200	300	6
3	300	500	6
4	400	500	8
5	500	650	6
6	600	800	6
7	700	300	8
8	800	300	10
9	900	300	12
10	1000	400	10
11	1100	400	12
12	1200	400	14
13	1300	400	16
14	1400	400	18
15	1500	400	20
16	1600	400	20
17	1700	600	18
18	1800	750	16
19	1900	900	14
20	2000	1050	12

Anhang 1: Versuchsaufbau dynamische Abfragerate

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den

_____

_____

[Florian Schutz]