Fernuniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Moderne Programmiersysteme
Dr. Daniela Keller

Ausarbeitung

Lastverteilung von HTTP/2-Anfragen in Kubernetes

im Rahmen des Seminars
Moderne Programmiertechniken und -Methoden

Lars Hick

Betreuerin: Daniela Keller

Inhaltsverzeichnis

Überblick

Kubernetes ist im Cloud Nativen Bereich zu einem Industriestandard zur Container Orchestrierung geworden. In Kombination mit dem HTTP/2 Protokoll bietet Kubernetes jedoch immer noch einige Fallstricke, die es bei der Entwicklung und Bereitstellung einer auf beiden Technologien basierenden Anwendung zu beachten gibt. Zu Begin werden in Kapitel 1 die verwendeten Technologien sowie deren Grundlagen dargestellt. Nachdem der Grundstein gelegt ist, wird in kapitel 2 eine Beispielanwendung vorgestellt, welche als Referenz HTTP/1 verwendet und an der anschließend verschiedene Möglichkeiten zur Lastverteilung von HTTP/2 Anfragen in einem Kubernetes Cluster exerziert werden. Diese Implementierungen liefern genug Metriken, um sie miteinander zu vergleichen. Der Vergleich zwischen den Implementierungen mit Blick auf quantitativ messbare wie Fairness, Performance und qualitative wie die Nutzbarkeit wird in Kapitel 3 getroffen. Abschließend wird in Kapitel 4 das Ergebnis der Arbeit bewertet sowie in Kapitel 5 Grundlage für weitere Forschung gegeben.

1 Grundlagen

Dieses Kapitel gibt einen Überblick über verwendete Technologien und Konzepte. Zunächst werden relevante Protokolle und deren Funktionsweise erläutert. Anschließend wird eine Einführung über Kubernetes gegeben.

1.1 Hypertext Transfer Protocol

Das Hypertext Transfer Protocol (HTTP) ist ein Protokoll zur Kommunikation über das World Wide Web, welches von der Internetengineering Task Force (IETF) standardisiert und weiterentwickelt wird. Da HTTP sich im OSI-Modell der Anwendungsschicht zuordnen lässt, bringt das Protokoll unter anderem die Vorteile mit sich, dass es unabhängig von der darunterliegenden Netzwerktechnologie und einfach erweiterbar ist. So bildet HTTP die Grundlage für viele weitere Protokolle wie zum Beispiel Hypertext Transfer Protocol Secure (HTTPS) [14] zur verschlüsselten Übertragung von Daten oder das Simple Object Access Protocol (SOAP) [2] zur Kommunikation zwischen verteilten Systemen. Aufgrund seiner Erweiterbarkeit und einiger Performance Optimierungen in der Vergangenheit liegt HTTP heute in den Versionen 1.0, 1.1 und 2 vor. Eine version 3 ist momentan in Entwicklung. Die folgenden Abschnitte gehen kurz auf für die folgende Arbeit relevante versionsspezifische Besonderheiten ein.

1.1.1 HTTP/1.0 und HTTP/1.1

HTTP ist ein textbasiertes, zustandsloses Protokoll, was bedeutet, dass sogenannte aus Klartext bestehende Anfragen (Requests) unabhängig voneinander sind. Dieser Umstand erlaubt die horizontale Skalierung von Webservern und vereinfacht deren Implementierung. RFC1945 [11] spezifiziert HTTP/1.0. Zur Spezifikation gehört die Definition von Unique Resource Identifiers (URIs), welche gemeinhin bekannt sind als Internetadresse. Des Weiteren erfolgt die Definition verschiedener HTTP-Methoden wie unter anderen *GET* (Informationsabfrage), *HEAD* (Metadatenabfrage) und *POST* (Hochladen von Informationen) und Status Codes zur Auswertung der Server Antwort (Response). Nach HTTP/1.0 wird für jeden Request eine neue TCP-Verbindung etabliert, welche nach Übertragung der Antwort wieder geschlossen wird.

Da mit HTTP/1.0 die Grundlagen zur Kommunikation im Web gelegt sind, wird das Protokoll in RFC2616 [12] um Version HTTP/1.1 erweitert. Im Wesentlichen werden mit der neuen Version die folgenden Punkte zur Verbesserung der Perfor-

mance eingeführt:

- **Persistente Verbindungen:** TCP Verbindungen, welche zwischen Client und Server aufgebaut werden, werden wiederverwendet, anstatt geschlossen zu werden. Diese Konfiguration wird in HTTP/1.1 zum Standard, reduziert den Overhead des TCP-Handlings und führt zu schnelleren Antwortzeiten.
- **Pipelining:** Persistente Verbindungen erlauben das Senden mehrerer paralleler Requests, ohne auf die Antwort des Servers zu warten.
- **Chunked Transfer-Encoding:** Daten können in mehreren Teilen übertragen werden. Dieses Verhalten erlaubt das Streaming von Daten.

1.1.2 HTTP/2

Obwohl HTTP/1.1 mit dem Pipelining bereits die parallele Übertragung von Requests ermöglicht, birgt diese Technologie einige Nachteile. So können beim Pipelining über persistente Verbindungen zu unterschiedlichen Anwendungsszenarien Head of Line Blocking Probleme auftreten. Obwohl die Anfragen parallel und ohne auf die Antwort zu warten abgeschickt werden, müssen die Antworten in der entsprechenden Reihenfolge vom Server zurückgegeben werden. Im schlimmsten Fall geht eine Antwort verloren und die komplette Warteschlange muss auf die erneute Sendung der Antwort warten. Schnellere Requests werden so durch langsamere blockiert und schlussendlich erfolgt trotzdem eine sequentielle Abarbeitung der Anfragen [4].

HTTP/2 bringt mit dem Multiplexing eine Neuerung, welche die Problematik des Head of Line Blocking reduziert. Da die Request-/Response-Paare nun unabhängig voneinander über eine Verbindung multiplexed werden können, ist die Reihenfolge der empfangenen Antworten nun irrelevant. Wichtig zu erwähnen ist, dass Multiplexing das Head of Line Blocking Problem auf HTTP-Ebene eliminiert, jedoch nicht auf TCP-Ebene. Neben Multiplexing und weiteren Neuerungen bringt HTTP/2 außerdem die Priorisierung von Requests, wie auch binäre Kodierung und Kompression von Header Feldern mit sich, was zu wesentlich kleineren Paketgrößen und damit schnelleren Übertragungsraten führt. HTTP/2 ist abwärtskompatibel zu seinen Vorgängern und wird in RFC9113 [15] spezifiziert.

Corbel et al. zeigen in ihrer Arbeit *HTTP/1.1 pipelining vs HTTP2 in-the-clear: Performance comparison* [3] die Performanceverbesserungen von HTTP/2 gegenüber HTTP/1.1 auf.

1.2 gRPC

gRPC ist ein von Google entwickeltes, auf HTTP/2 basierendes Remote Procedure Call (RPC) Framework, welches Quelloffen zur Verfügung gestellt wird. Das Framework bringt die Besonderheit mit, dass Schnittstellendefinitionen Programmiersprachenunabhängig in der Interface Definition Language (IDL) namens Protocol Buffers Language (protobuf) [10] definiert werden können. Die Abstraktion der Schnittstellendefinitionen von der Implementierung erlaubt die Entwicklerübergreifende Verständigung auf eine gemeinsame Schnittstelle. Zudem bietet das Framework einen Compiler an, welcher die Schnittstellendefinitionen sowohl in Server-, als auch in Client-Code in einer Vielzahl von Programmiersprachen übersetzen kann. Mit gRPC kann ein strikter API-First Ansatz verfolgt werden, welcher die Entwicklung von Client und Server unabhängig voneinander ermöglicht. Da gRPC auf HTTP/2 basiert, welches von der kompletten Infrastruktur auf dem Weg zwischen Client und Server unterstützt sein muss, bietet sich gRPC vor allem zur Kommunikation zwischen verschiedenen Komponenten innerhalb eines Kubernetes Clusters an, wo die komplette Kommunikationsstrecke technisch in einer Hand liegen.

1.3 Kubernetes

Das Kapitel im Anschluss führt in das Software Ökosystem Kubernetes ein. Zu Beginn wird ein grundlegender Überblick über die Architektur und die Funktionsweise gegeben. Danach werden einige in der Praxis oft verwendete Komponenten und deren Konzepte erläutert. Da Kubernetes ein sehr umfangreiches Ökosystem ist, wird in diesem Kapitel ausschließlich auf die für die Arbeit relevanten und daher sich in Nutzung befindlichen Komponenten eingegangen.

1.3.1 Kubernetes Control Plane

Kubernetes ist ein von Google entwickeltes Container Orchestration Framework. Für die Cloud-Native Entwicklung eignet es sich besonders gut als Abstraktion mehrerer virtueller Maschinen für eine einfachere, hardware unabhängige Bereitstellung von Anwendungen, welche hoch verfügbar oder skalierbar sein sollen. Ein wesentlicher Bestandteil von Kubernetes ist die Control Plane, welche die zentrale Steuerung des Clusters zur Aufgabe hat. Die Control Plane besteht aus mehreren Komponenten, welche je nach Anwendungsfall in einer High-Availability Konfiguration laufen können. In jedem Fall besteht die Control Plane aus folgenden Komponenten:

- **API Server:** Über den API Server erfolgt die Kommunikation mit Kubernetes

und der Control Plane. Jede Interaktion mit einer Kubernetes Ressource lässt sich auf einen zugrunde liegenden Schnittstellen Aufruf an den API Server zurückführen.

- **Controller Manager:** Eine Sammlung verschiedenster Kontrollprozesse, welche nach dem Unix Principle *Do One Thing and Do It Well* [7] aufgeteilt sind.
- **etcd:** Ein Key-Value-Store, welcher als Persistenz für das Cluster und dessen State dient.
- **Scheduler:** Der Scheduler übernimmt die Aufgabe, Pods auf freie Nodes zu verteilen.

Des Weiteren befinden sich in einem Kubernetes Cluster, bestehend aus mehreren Virtual Machines (VMs), welche in diesem Kontext auch als Node bezeichnet werden, auf jedem Node folgende Komponenten:

- **Kubelet:** Das Kubelet stellt sicher, dass provisionierte Pods entsprechend ihrer Konfiguration laufen.
- **kube-proxy:** Stellt die Kommunikation der Kubernetes Komponenten miteinander sicher.

Die Linux Foundation bietet eine ausgezeichnete Dokumentation zur weitgehenderen Lektüre in die Kubernetes Control Plane [5].

Pod Der Pod ist die kleinste kubernetesspezifische Einheit. Er besteht aus einem oder mehreren Containern, welche konform der Standards der Open Container Initiative sind. Die unter Entwicklern am weitesten verbreitete Methode, Container zu bauen, ist mit Hilfe von Docker. Der Pod fungiert als ein Wrapper um seine Container, um diese als eine Einheit zu behandeln und in das Kubernetes Ökosystem integrierbar zu machen. Die Hauptaufgabe des Pods ist es, Informationen über Ressourcenverbrauch sowie den Gesundheitszustand des Stücks Software, welches in den Containern läuft, zu sammeln und der Control Plane bereitzustellen. Wird ein Pod aktualisiert, so muss dieser neu gestartet werden, da es sich dabei um eine unveränderliche Einheit handelt.

Deployment Mehrere Pods, welche die Konfiguration und bereitgestellten Container teilen, können in einem Deployment zusammengefasst werden. Ein Deployment bietet die Möglichkeit, mehrere Pods des gleichen Typs zentral zu verwalten. So wird durch diese Komponente eine Vielzahl für moderne Softwaretechnik relevanter Funktionen bereitgestellt:

- **Skalierbarkeit:** Die Anzahl der gleichzeitig laufenden Pods kann über ein Deployment definiert werden. So können High availability Anforderungen durch einfache Konfiguration des Deployments erfüllt werden.
- **Rolling Updates:** Liegt Software in einer neuen Version vor, so muss diese auch ausgerollt werden. Wie oben geschrieben, kann der Container in einem Pod nicht einfach ausgetauscht werden. Durch rolling Updates wird sichergestellt, dass die neue Softwareversion in neuen Pods bereitsteht, bevor Pods mit der alten Version heruntergefahren werden.
- **Rollbacks:** Wurde eine Version bereitgestellt, welche fehlerhaft ist, so kann über ein Deployment einfach die vorherige Version wiederhergestellt werden.

Service Um ein Deployment verfügbar zu machen, kann ein Service definiert werden. Je nach Typ erfüllt ein Service unterschiedliche Funktionen. Laut Dokumentation [6]existieren die folgenden:

- **ClusterIP:** Der Service erhält eine IP-Adresse, welche nur innerhalb des Clusters erreichbar ist.
- **NodePort:** Erlaubt den externen Zugriff auf den Service über das entsprechende Protokoll und den Port. Ermöglicht zudem die eigene Implementierung eines Loadbalancing verfahrens.
- **LoadBalancer:** Ist das Kubernetes Cluster über einen Cloud-Provider provisioniert, so wird in diesem Fall automatisch ein Loadbalancer für den Service erstellt, welcher auf diesen verweist.
- **ExternalName:** Erlaubt das direkte Mapping eines DNS Names auf einen Service.

1.3.2 Custom Resource Definitions

Jede Interaktion mit einem Kubernetes Cluster stellt einen API-Aufruf an die Control Plane dar. Custom Resource Definitions ermöglichen es dem Verwalter eines Kubernetes Cluster, diese API um eigene Ressourcen zu erweitern, welche dann über die typischen Kubernetes Definitionen bereitgestellt werden.

2 Kernkapitel

Folgendes Kapitel erläutert den Aufbau der Testanwendung. Zu Beginn wird die Anwendung für einen Verbindungsaufbau über HTTP/1 konfiguriert. Anschließend wird die Problematik mit HTTP/2 in der Standardkonfiguration und verschiedene Lösungsansätze dafür dargestellt. Da es sich beim HTTP/1 Load Balancing um den normalen Anwendungsfall für Kubernetes Loadbalancing handelt, dient dies zur Referenz in den anschließenden Konfigurationen, welche die Anwendung im gRPC-Modus und somit über HTTP/2 ausführen. Der gesamte Code der Anwendung ist online einsehbar [8].

2.1 Anwendungsaufbau

Das Kapitel gibt einen Überblick über den generellen Aufbau der Anwendung, um evaluierbare Daten zu generieren. Des Weiteren wird ein Überblick gegeben, wie generierte Daten von der implementierten Anwendung abgerufen und visuell dargestellt werden können.

2.1.1 Client-Server-Architektur

Da nicht die Anwendung selbst, sondern die Kommunikation zwischen zwei unterschiedlichen Pods und deren Replicas im Vordergrund steht, wird eine einfache Client-Server-Anwendung entwickelt.

Der Server stellt hierbei eine Schnittstelle zur Verfügung. Um einen gewissen Rechenaufwand zu erzeugen, berechnet die aufrufbare Schnittstelle für einen gegebenen Wert über das Heron Verfahren die Wurzel und gibt das Ergebnis als Antwort an den Aufrufer zurück.

Dem Server gegenüber steht der Client, welcher in regelmäßigen, konfigurierbaren Zeitabständen eine randomisiert ausgewählte Zahl an die vom Server bereitgestellte Schnittstelle als Anfrage sendet und auf die Antwort mit dem Ergebnis vom Server wartet, bevor eine nächste Anfrage abgesetzt wird.

Über Command Line Argumente können beide Anwendungen in folgenden verschiedenen Modi gestartet werden, welche in den referenzierten Sektionen detaillierter vorgestellt werden:

- HTTP/1 2.2
- HTTP/2 Standard 2.3.1
- HTTP/2 headless Service 2.3.2

- HTTP/2 Service Mesh 2.3.3

Für jeden Laufmodus wird die Anwendung in einem separaten Kubernetes Namespace vorkonfiguriert bereitgestellt. Aus Gründen der Vorhersehbarkeit wird auf Horizontal Pod Autoscaler und somit auf die automatische Skalierung der Anwendung auf Basis von Ressourcenmetriken verzichtet.

2.1.2 Auswertbarkeit

Um auswertbare Daten zu erhalten, stellen sowohl Server- als auch Client eine weitere Schnittstelle bereit, welche frei definierbare Metriken exponiert und einem Scraper zur regelmäßigen Abfrage zur Verfügung stellt. Grafik 1 visualisiert das komplette Setup, wobei die Client-Server-Anwendung, welche auf mehrere Namespaces in unterschiedlichen Konfigurationen läuft, abstrahiert dargestellt wird:

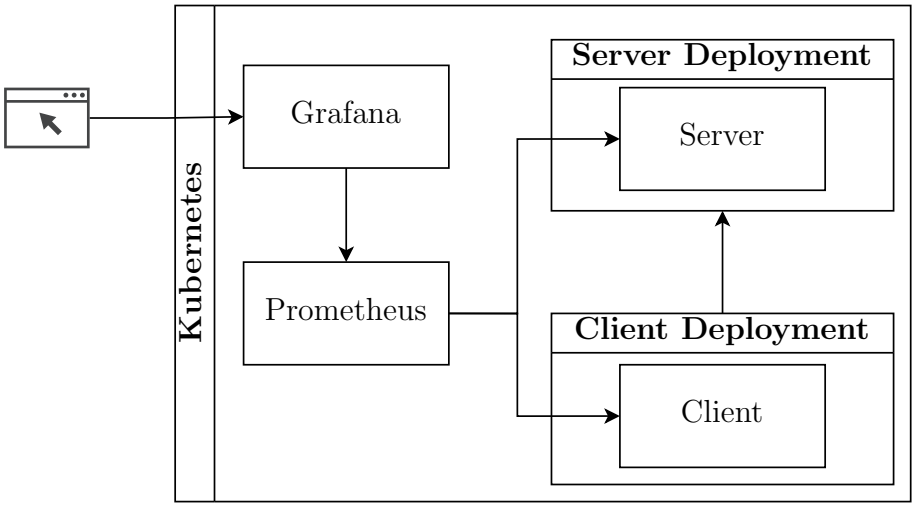

Abbildung 1: Anwendungsarchitektur

Wie zu erkennen ist, wird zur Datenanalyse der Cloud Native Metrik-Scraper Prometheus und zur Darstellung der Daten das Tool Grafana verwendet. Das zur Verfügung gestellte Repository kommt mit einem vorkonfigurierten Dashboard, welches für jeden Laufmodus die folgenden Metriken visualisiert:

- Anzahl Pods pro Client und Server
- Requests pro Sekunde für jeden Server Pod
- Client Latenz im 99ten, 95ten Perzentil sowie Median

Um verschiedene Skalierungsszenarien zu simulieren, wird für jede Laufvariante der in Grafik 2 dargestellte zeitliche Ablauf festgelegt:

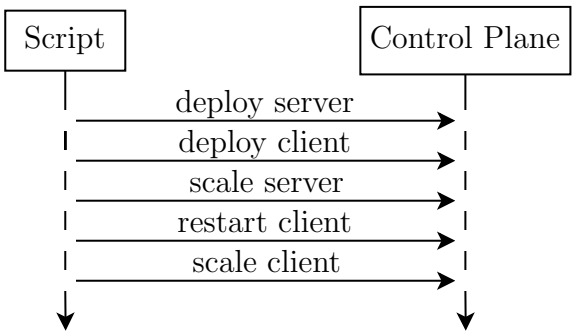

Abbildung 2: Interaktionen mit dem Kubernetes Cluster

Der Server startet mit zwei und der Client mit einer Instanz. Da gRPC, wie in 1.2 erwähnt, vorwiegend zur In-Cluster Kommunikation verwendet werden sollte, besteht sowohl für den Server, als auch für den Client die Notwendigkeit, bei Bedarf variabel skalierbar zu sein. Ersteres Szenario wird durch die Skalierung des Servers abgedeckt. Bevor der Client skaliert und somit ein erhöhtes Anfrageaufkommen für den Server generiert wird, wird dieser neu gestartet. Dies dient zur Veranschaulichung eines Sonderfalls, auf welchen in 2.3.2 eingegangen wird.

2.1.3 Weitere Tools und Hardware

Zur Bereitstellung des Kubernetes Clusters wird auf das Tool ctlptl von tilt zurückgegriffen. Dieses ist so konfiguriert, dass ein Kubernetes Cluster mit Hilfe von KinD (Kubernetes in Docker) provisioniert wird. Das Cluster umfasst einen Knoten für die Control Plane sowie vier weitere Knoten als Worker. Die im Folgenden vorgestellten Ergebnisse werden auf einem Macbook Pro mit einem Apple M2 Pro und 16 GB RAM ermittelt.

2.2 HTTP/1 Load Balancing

Zum Zeitpunkt der Entwicklung von Kubernetes handelte es sich bei HTTP/1 bereits um einen weit verbreiteten Industriestandard für das Web. Daher basieren heute viele Webanwendungen sowie Webserver auf dem Protokoll, weswegen das Container Orchestration Framework Load Balancing über HTTP/1 standardmäßig unterstützt. Um für das folgende Kapitel, welches sich mit HTTP/2 Load Balancing beschäftigt, Werte zu liefern, die repräsentieren, wie die Lastverteilung in einer funktionierenden Anwendung aussehen sollten, wird die vorgestellte Anwendung im HTTP/1 bzw. REST-Modus ausgeführt.

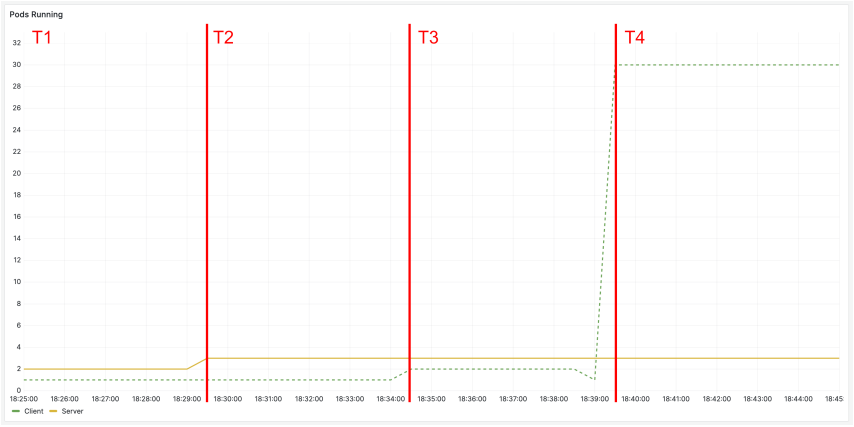

Abbildung 3: Client (gestrichelt) und Server (durchgezogen) Pods

Grafik 3 visualisiert den Verlauf der Replicagröße beider Deployments. Die folgenden Zeitpunkte sind hierbei besonders von Relevanz:

- **T1**: Server startet mit zwei Instanzen, der Client mit einer.
- **T2**: Der Server wird auf drei Instanzen skaliert
- **T3**: Der Client wird neu gestartet.
- **T4**: Der Client wird auf dreißig Instanzen skaliert.

Da von diesem Verhalten in den Sektionen 2.3.1, 2.3.2 und 2.3.3 nicht abgewichen wird und die Zeitpunkte als Referenz nun erläutert wurden, wird auf eine erneute Darstellung dessen im Folgenden verzichtet.

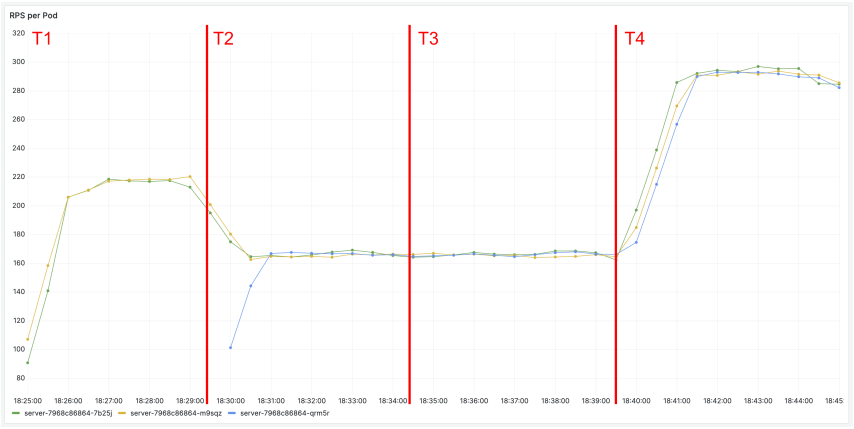

Abbildung 4: Server RPS pro REST Pod

Wie an der annähernden Äquivalenz beider Graphen bis T2 in 4 zu erkennen ist, werden die Client-Requests von Anfang an sehr gleich auf beide Instanzen des Servers verteilt. Sobald der Server zum Zeitpunkt T2 auf drei Instanzen skaliert wird, werden die Requests auf drei Server-Instanzen verteilt. Dies hat eine Lastverringerung der bereits vorher zur Verfügung gestellten Server-Instanzen sowie einen in Summe erhöhten Gesamtdurchsatz zur Folge. Der Neustart zum Zeitpunkt T3 sowie die Skalierung des Clients zum Zeitpunkt T4 haben keinen Einfluss auf die Lastverteilung der Anfragen zum Server, welche gleichbleibend gleich verteilt werden.

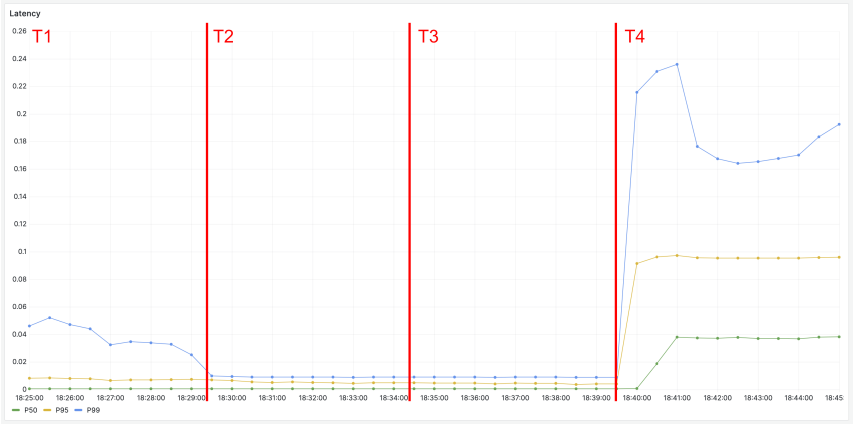

Abbildung 5: Rest Latenz (P99, P95, P50)

Die Latenz im 99. Perzentil, gezeigt in 5, verbessert sich zum Zeitpunkt T2 aufgrund der nun möglichen Lastverteilung auf drei Server Instanzen, welche in Summe mehr Rechenleistung zur Verfügung haben. Durch das hohe Aufkommen an Anfragen nach der Client-Skalierung zum Zeitpunkt T4 ist eine Verschlechterung der Latenz zu beobachten, da die Gesamtrechenleistung für den Server unangetastet bleibt.

2.3 HTTP/2 Load Balancing

Nachdem nun gezeigt wurde, wie sich die Lastverteilung einer HTTP/1 Anwendung mit dem Standard Tooling von Kubernetes auswirkt, zeigt dieses Kapitel, wie sich eine Lastverteilte Anwendung, welche auf HTTP/2 basiert und sich das Standard Verhalten von Kubernetes zu Nutze macht, verhält. An diesem Szenario werden einige Problematiken aufgezeigt, für welche in den folgenden Abschnitten nach einer Lösung gesucht wird.

2.3.1 Standard Load Balancing

Im Vergleich zu Szenario 2.2 wird die Beispielanwendung in diesem Szenario dazu konfiguriert, die gRPC-Implementierung zu verwenden. Bis auf diesen, maßgeblichen, Unterschied, bleibt die Konfiguration identisch.

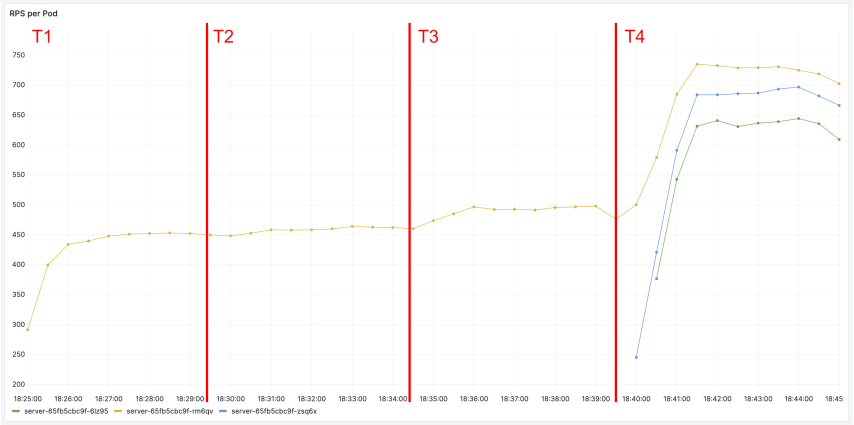

Abbildung 6: Server RPS pro HTTP/2 Standard Pod

Grafik 6 spiegelt das Problem dieses Setups gut wider. Von zeitpunkt T1 bis T4 ist zu erkennen, dass alle Requests von einem einzigen Server Pod beantwortet werden. Diese Tatsache lässt sich damit begründen, dass HTTP/2, wie in 1.1 schon erwähnt, auf langlebigen Verbindungen basiert. Zum Start des Clients wird somit eine beständige Verbindung zwischen Client und Server etabliert (siehe 7). Dies hat zur Folge, dass der TCP-handshake vermieden wird und führt zu dem Problem, dass jede einzelne Anfrage über die selbe TCP Verbindung immer an die gleiche Instanz des Servers gesendet wird.

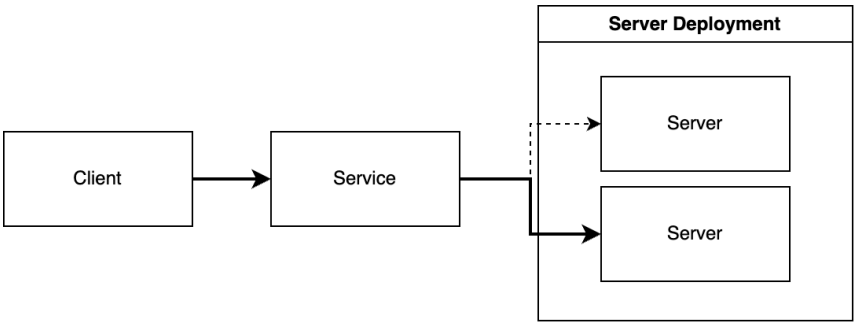

Abbildung 7: Default Implementierung Verbindungsaufbau

Dass sich weder zum Zeitpunkt T2, noch T3 am Verhalten etwas ändert, hat hier damit zu tun, dass schlicht der gleiche Server zufälligerweise ausgewählt wird. Da jeder Pod eine beständige Verbindung zu einem zum Startpunkt ausgewählten Server hat, kann man in diesem Fall nicht von Lastverteilung sprechen, auch wenn eine (unvorhersehbare) Verteilung der Pods auf exakt einen vorselektierten Server vollzogen wird.

Erst ab Zeitpunkt T4 werden die Anfragen der verschiedenen Clients von allen drei Servern beantwortet. Dies hängt einfach mit der großen Anzahl der Client Pods zusammen. Wichtig ist jedoch weiterhin, dass jeder einzelne dieser Pods immer mit genau demselben Server kommuniziert und somit keine Lastverteilung auf Request-Ebene stattfindet.

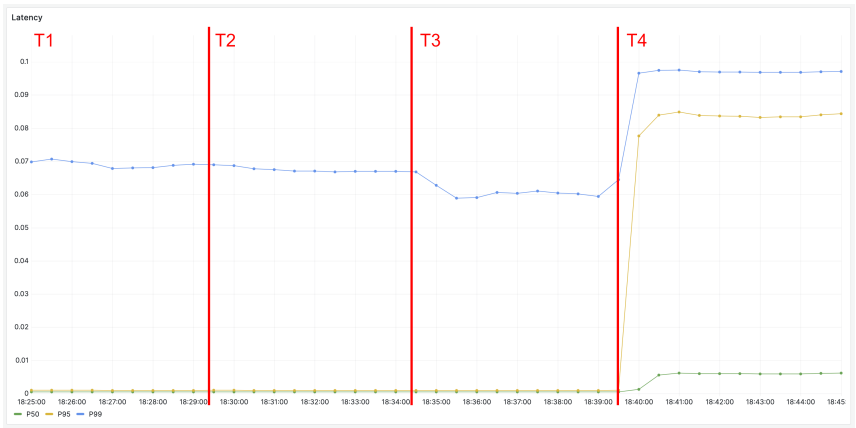

Abbildung 8: HTTP/2 Standard Latenz (P99, P95, P50)

In Puncto Latenz lässt sich in Grafik 8 erkennen, dass bis zu Zeitpunkt T4 keine nennenswerten Veränderungen stattfinden.

2.3.2 Headless Service

Nun, da gezeigt wurde, warum ein Kubernetes Service in der Standard Konfiguration keine Option zur Lastverteilung von HTTP/2-Requests ist, wird eine funktionieren-

de Variante zur Verteilung der Requests vorgestellt. Dieses Teilkapitel befasst sich mit Headless Services, welche unter minimalem konfigurativen Eingriff erlauben, die Last an aufkommenden HTTP/2 Anfragen auf mehrere Server Pods zu verteilen. Listing 1 zeigt, wie ein Service des Typs ClusterIP normalerweise in einer Kubernetes Umgebung definiert wird. Der Service erhält so eine eigene IP, welche von den Clients zentral aufgerufen wird, um auf die Replicas des Servers zuzugreifen. Ein Namespace-Lookup auf den Service gibt hier einfach die dem Service eigens zugewiesene IP-Adresse zurück.

Im Gegensatz dazu zeigt Listing 2 die Definition eines Headless Services. Ein Headless Service birgt die Besonderheit, dass in seiner Definition explizit angegeben wird, dass keine IP zugewiesen werden soll. Da der Service nun keine eigene IP besitzt, werden für einen Namespace Lookup alle bekannten Pod-IPs zurückgegeben [1]. Dieses Verhalten können wir uns im Client zunutze machen. Zum Start des Clients werden vom Server alle bekannten IP-Adressen der Anfragen-bearbeitenden Pods geliefert. Der Client kann nun selbst entscheiden, welcher Server zur Bearbeitung der Anfrage aufgerufen wird. Es findet clientseitiges Loadbalancing statt.

Listing 1: Standard Service

```
apiVersion: v1
kind: Service
metadata:
  name: server
  namespace: default
  labels:
    app: server
spec:
  selector:
    app: server
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      name: app
```

Listing 2: Headless Service

```
apiVersion: v1
kind: Service
metadata:
  name: server
  namespace: headless
  labels:
    app: server
spec:
  selector:
    app: server
  clusterIP: None
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      name: app
```

Abbildung 9: Server RPS pro Headless Service Pod

Grafik 9 zeigt das Anfrageaufkommen der Pods, welche über den Headless Service zur Verfügung gestellt werden. Es ist zu erkennen, dass zu T1 mit dem Loadbalancing der Anfragen auf beide verfügbaren Instanzen begonnen wird. Die Verteilung findet gleichmäßig statt, was sich wieder an der annähernden Äquivalenz beider Kurven erkennen lässt. Zum Zeitpunkt T2, wenn ein neuer Server Pod hinzukommt, ändert sich nichts. Eine Änderung ist jedoch zum Zeitpunkt T3, welcher den Neustart der Client Pods markiert, zu beobachten. Die aufkommenden Anfragen werden jetzt gleichmäßig auf alle drei Instanzen des Servers verteilt, was einer allgemeinen niedrigeren Last pro Pod entspricht. Das Verhalten, dass die Requests erst auf alle Server Instanzen verteilt werden, nachdem die Clients neu gestartet wurden, lässt sich mit der Funktionsweise des Headless Services erklären.

Zum Start des Clients (hier T1) werden beim Service alle verfügbaren Server IPs abgefragt (siehe 1 und 2 in Grafik 10). Da zu diesem Zeitpunkt die dritte Instanz noch nicht verfügbar ist, sind dem Client auch nach Skalierung des Servers weiterhin nur zwei der inzwischen drei IPs bekannt, zwischen welchen die Last per Client verteilt wird. Auch ab Zeitpunkt T4 werden alle Anfragen weiterhin gleichmäßig auf die drei vorhandenen Server Instanzen verteilt.

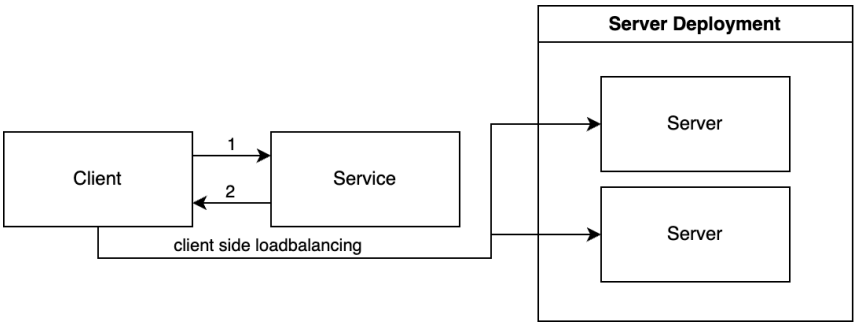

Abbildung 10: Headless Service Client Side Loadbalancing

Vergleicht man Grafik 9 mit Grafik 6, wird schnell ersichtlich, dass die in Listing 2 gezeigte, kleine Anpassung eine große Änderung bewirkt, um das Loadbalancing von HTTP/2 Requests zu verbessern. Zieht man jedoch die Ergebnisse in Grafik 4 heran, sieht man auch, dass HTTP/1 Loadbalancing hier im Vergleich qualitativ noch immer besser funktioniert, weil die Anfragen einen neuen Server Pod sofort erreichen, sobald dessen Verfügbarkeit sichergestellt ist.

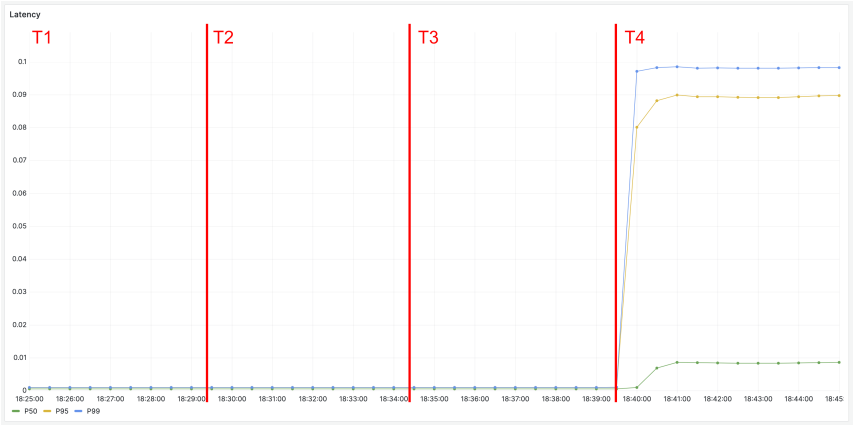

Abbildung 11: Headless Service Latenz (P99, P95, P50)

Figur 11 gibt einen Überblick über die Latenzen zu diesem Szenario. Bis zu Zeitpunkt T4 sind die Latenzen sehr unauffällig und nah beieinander.

2.3.3 Service Mesh

Mit zunehmender Komplexität Cloud nativer Anwendungen stieg der Bedarf, den Verwaltungsaufwand jener wieder zu minimieren. Hierfür wurden Service Meshes eingeführt. Ein Service Mesh dient als Layer zwischen Applikations- und Orchestrierungsebene und übernimmt eine Vielzahl Aufgaben wie, für unser Szenario relevant, das Netzwerkmanagement.

Jeder Pod, welcher dem Service Mesh zugewiesen wurde, wird mit einem sogenannten Sidecar bereitgestellt. Ein Sidecar ist ein zweiter Container innerhalb eines Pods, welcher einem bestimmten Zweck dient [9]. Im Falle eines Service Meshes handelt es sich um einen leichtgewichtigen Proxy server, welcher jeglichen Netzwerkverkehr unterbricht und kontrolliert.

Es gibt eine Vielzahl verschiedener Implementierungen für Service Meshes. Aufgrund der simplen Installationsmöglichkeit wird an dieser Stelle mit LinkerD fortgefahren. LinkerD bietet ein Command Line Interface an, welches einem hinterlegten Kubernetes Cluster die notwendigen CRDs sowie das Service Mesh selbst installiert.

Sobald das Service Mesh im Cluster installiert ist, lassen sich Pods über Annotationen zu einem Mesh zusammenfassen. Da im vorliegenden Beispiel jeder Anwendungsfall seinen eigenen Namespace besitzt, lässt sich - wie in Listing 4 zu sehen - über die Annotation des Namespaces einfach definieren, dass jeder beinhaltete Pod Teil des Service Meshs sein soll.

Listing 3: Default Namespace

```
apiVersion: v1
kind: Namespace
metadata:
  name: default
```

Listing 4: Service Mesh Namespace

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    linkerd.io/inject: enabled
  name: servicemesh
```

Nachdem der Namespace annotiert wurde, können die Pods wie gewohnt über ihre Deployments bereitgestellt werden. Abbildung 12 stellt die Kommunikation zwischen Client und Server Pods im Cluster dar. Es ist zu erkennen, dass das Verbindungsmanagement nun vollständig von den Proxies, welche als Sidecar injiziert werden, übernommen wird und jeglicher Traffic über diese geleitet wird.

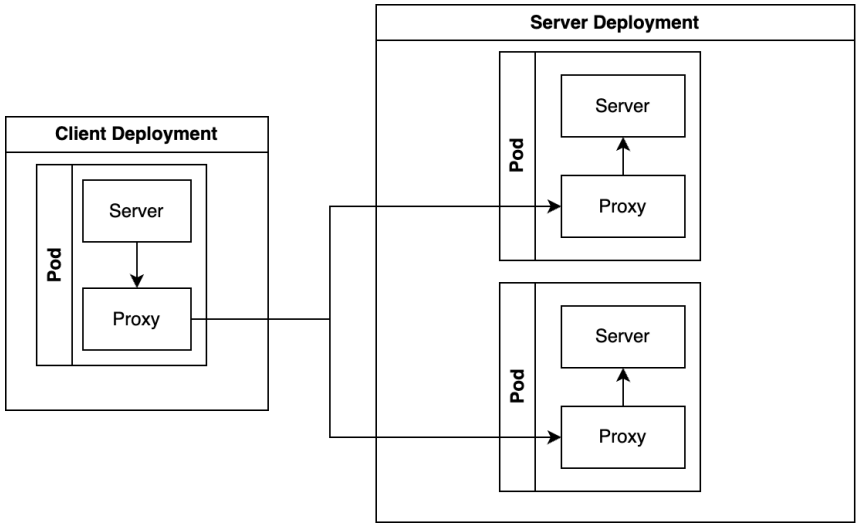

Abbildung 12: Connection Handling Service Mesh

Wie in den vorangegangenen Kapiteln folgt eine Grafik, welche den Durchsatz der entsprechenden Implementierung pro Server Pod visualisiert. In Grafik 13 ist zu erkennen, dass ab T1 eine Lastverteilung zwischen beiden verfügbaren Server Instanzen stattfindet. Ab Zeitpunkt T2, welcher die Skalierung des Server Deployments auf drei Instanzen markiert, ist zu sehen, dass die Last nun auf drei Instanzen verteilt wird, eine dritte Kurve kommt hinzu. Der Client Neustart zum Zeitpunkt

T3 hat keine sichtlichen Auswirkungen auf die am Server eintreffenden Anfragen pro Sekunde.

Auch eine Erhöhung der Last durch die Skalierung des Client deployments zeigt, dass die Last weiterhin auf die vorhandenen Server Instanzen verteilt wird.

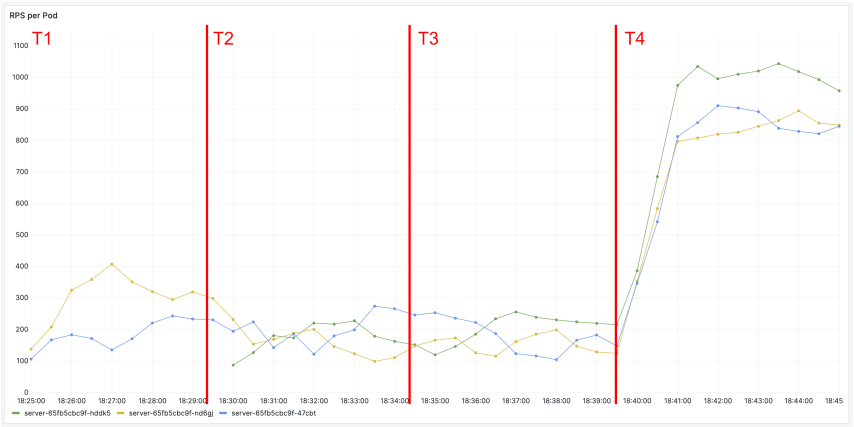

Abbildung 13: Server RPS pro Service Mesh Pod

Grafik 14 visualisiert die Latenz des vorgestellten Nutzungsszenarios. Man sieht, dass die Latenz im 99. Perzentil um einiges von der im 95. Perzentil abweicht. Ein detaillierter Vergleich der vorgestellten Metriken findet in Kapitel 3 statt.

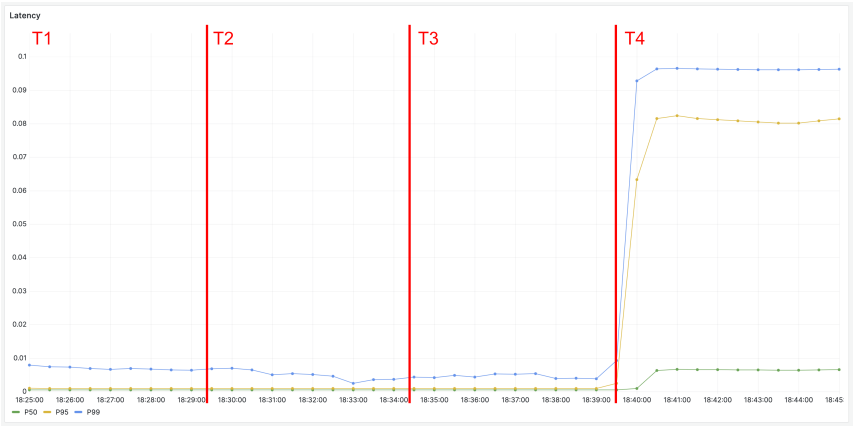

Abbildung 14: Service Mesh Latenz (P99, P95, P50)

3 Auswertung

Das folgende Kapitel beschäftigt sich mit der Auswertung der in den vorangegangen Kapiteln vorgestellten Metriken. Zu Beginn wird auf die Fairness der verschiedenen Implementierungen eingegangen. Anschließend werden die verschiedenen vorgestellten Varianten der Bereitstellung auf ihre Performance hin untersucht. Abschließend wird die Nutzbarkeit der Lösungen verglichen. Hier spielen weniger Metriken als Qualitätsmerkmale der Lösungen zur Nutzbarkeit in der Praxis eine Rolle.

3.1 Fairness

Fairness spielt in Hinblick auf Skalierbarkeit eine große Rolle und ist eine notwendige Bedingung für die Erreichbarkeit guter Performance. Ohne ein faires Loadbalancing kann nicht sichergestellt werden, dass jede Instanz eines Deployments bestmöglich ausgelastet werden kann.

Um die Fairness zu bewerten, genügt es zu beobachten, wie viele Anfragen jede Instanz in einem gegebenen Zeitraum erreichen. Da zum Zeitpunkt T2 eine Skalierung des Server Deployments stattfindet, wird der Zeitraum T2 bis Ende herangezogen. Auf Basis der Metrik, welche auch für den Durchsatz (RPS) herangezogen wird, ergibt sich Tabelle 3.1.

	Pod 1	Pod 2	Pod 3	Standardabweichung
REST	265757	262532	194967	32637.153
Standard	198091	665355	224484	214320.496
Headless	451578	440857	245620	94663.805
Mesh	446578	467815	467573	9954.668

Tabelle 3.1 zeigt, wie viele Anfragen jeder Pod pro Variante im Zeitraum von T2 bis zum Messende erhalten hat. Auf basis dieser Werte wird die Standardabweichung berechnet. Es ist zu erkennen, dass die Umsetzung über das Service Mesh mit einer Standardabweichung von 9.954,668 Anfragen am fairsten ist. An dieser Stelle ist zu erwähnen, dass diese Variante die einzige unter den HTTP/2 Loadbalancing Implementierungen ist, welche sogar fairer ist, als das Kubernetes Standard Loadbalancing von HTTP/1 Anfragen mit einer Standardabweichung von 32.637,153 Anfragen, gefolgt von der Headless Implementierung mit 94.663,805 und der Standard HTTP/2 Implementierung mit 214.320,496 abweichenden Anfragen.

3.2 Performance

Die Performance kann anhand zweier Metriken gemessen werden. Diese Metriken betreffen zum einen den serverseitig messbaren Durchsatz sowie die clientseitig messbare Latenz, also die Dauer von Absetzen der Anfrage bis zum Empfang der entsprechenden Antwort.

3.2.1 Durchsatz

Serverseitig wird hier beobachtet, wie viele Anfragen das Deployment in Summe beantworten konnte, es wird also der gesamte Durchsatz auf den definierten Zeitraum verglichen. Wie in Sektion 3.1 bereits beschrieben, ist es auch hier sinnvoll, nur den Zeitraum von T2 bis zum Messende zu betrachten.

	Durchsatz
REST	723256
Standard	1087930
Headless	1138055
Mesh	1381966

Tabelle 3.2.1 gibt hier einen Überblick über die ermittelten Werte. Schon am Vergleich zwischen REST und Standard ist eine erhöhung des Durchsatzes erkennbar, auch wenn Sektion 3.1 ergeben hat, dass die Standard Implementierung die unfairste ist. Dies hat den Hintergrund, dass der TCP Handshake, welcher einen zeitlichen Aufwand bedeutet, durch die Beständigkeit der HTTP/2-Verbindungen von der Anzahl aller Requests reduziert werden kann auf die Anzahl der Clients. Da die Implementierungen Standard, Headless und Mesh auf dem gleichen Protokoll aufbauen, ist ein Vergleich hier besser angebracht. Es lässt sich eine Korrelation zur Fairness erkennen, indem die Mesh Implementierung, welche am fairsten ist, auch den höchsten Durchsatz aufweist. Gefolgt wird diese Implementierung von der Headless, welche - ungeachtet der REST-Implementierung - am zweit fairsten ist.

3.2.2 Latenz

Clientseitig ergibt sich die Möglichkeit, das Ergebnis anhand seiner Latenzen zu vergleichen. Hier ist es sinnvoll, die Latenzen in dem zeitraum zu betrachten, in dem das System der höchsten Last ausgesetzt war. Dies trifft auf den Zeitraum T4 bis zum Messende zu, da hier durch die Skalierung des Clients auf 30 Pods am meisten Netzwerk-Traffic generiert wurde.

	Ø-P50	Ø-P95	Ø-P99
REST	0,032	0,095	0,179
Standard	0,005	0,083	0,097
Headless	0,007	0,088	0,098
Mesh	0,005	0,079	0,096

Tabelle 3.2.2 gibt zu erkennen, dass jede gRPC-Implementierung im Vergleich zur REST Implementierung eine deutlich niedrigere Latenz aufweisen kann. Vergleicht man innerhalb dieser Implementierungen die Latenzen miteinander, so lässt sich erkennen, dass Headless minimal langsamer ist als die Standard Implementierung, das hängt vermutlich mit dem client seitigen Loadbalancing zusammen, das den angesprochenen Server auswählt, was bei der Standardimplementierung wegfällt. Schneller als die Standard Implementierung ist noch die Mesh Implementierung. Da jedoch die Latenz Unterschiede im Allgemeinen sehr gering sind, können diese vernachlässigt werden und spielen im Vergleich zum Durchsatz und der Fairness eine eher untergeordnete Rolle.

3.3 Nutzbarkeit

Alle Implementierungen wurden nun in Bezug auf ihre Performanz bewertet. Neben der Performance ist aber auch die Nutzbarkeit ein wichtiger Aspekt. So kann nicht jede Implementierung für jedes Nutzungsszenario empfohlen werden. Daher folgt eine kurze Nutzbarkeitsbewertung nicht-quantitativer Eigenschaften.

3.3.1 Standard

Die Standard Implementierung sollte aufzeigen, warum verschiedene Ansätze des Loadbalancings für HTTP/2 Applikationen betrachtet werden. Im Allgemeinen ist diese Umsetzung nicht für den Produktivgebrauch zu empfehlen, da sie nur mit der Anzahl der Clients skaliert. Automatische Skalierung der Server im Falle eines erhöhten Anfrageaufkommens wäre Sinnlos, da nur neue Clients eine Verbindung zu neuen Server Instanzen herstellen können, was auch eher dem Zufall überlassen wird.

3.3.2 Headless

Diese Umsetzung ermöglicht das client seitige Load Balancing. Clients rufen zu Beginn alle adressierbaren IPs über den Headless Service ab und verteilen die Last

dann im Round Robin verfahren selbst. Vorteil gegenüber dem Standard verfahren ist, dass Loadbalancing zumindest zwischen allen bekannten Server Instanzen stattfindet. Nachteil an dieser Umsetzung ist, ähnlich dem Nachteil der Standard Umsetzung, dass sie nicht unbedingt serverseitig skalierbar ist. Wenn eine neue Server Instanz zur Verfügung gestellt wird, wird diese nur von CLients angesprochen, welche nach der neuen Server Instanz provisioniert werden. Daher eignet sich diese variante besonders für den Anwendungsfall, dass der Server gar nicht automatisch skaliert werden soll bei last, sondern stets in einer festen Replicagröße provisioniert ist, da sie abgesehen von der Modifikation des Services, keinerlei Mehraufwand im Vergleich zur Standard Implementierung bedeutet.

3.3.3 Service Mesh

Handelt es sich bei dem Ökosystem um ein hoch dynamisches, in dem der Server Lastabhängig skaliert werden soll, so ist diese Umsetzungsmöglichkeit die einzige, welche alle Anforderungen erfüllt. Wird ein neuer Server provisioniert, so wird dieser direkt von allen bereits existierenden Clients angesprochen. Zudem ist diese Variante die fairste und Leistungsfähigste, dafür bedarf sie der Installation eines Service Meshes und arbeitet somit nicht mit Bordmitteln, welche von Kubernetes mitgebracht werden.

4 Fazit

Da HTTP/2 nicht von jeglicher Infrastruktur unterstützt wird und sich daher nur für die interne Kommunikation zwischen Services innerhalb eines Clusters anbietet, sollte stets abgewogen werden, ob es sinnvoll ist, auf gRPC mit HTTP/2 zu setzen. gRPC bietet einige erleichterungen und einen standardisierten Entwicklungsworkflow durch den API First Ansatz. Dieser kommt jedoch mit dem Tradeoff, dass ein Service Mesh installiert und verwaltet werden muss, sollte man alle Features benötigen, welche beim HTTP/1 Loadbalancing von Kubernetes von Hause aus mitgebracht werden.

5 Ausblick

In diesem Paper wird erläutert, warum die Nutzung von HTTP/2 bis heute in einem Kubernetes Kontext problematisch ist. Es sind verschiedene Lösungen dargestellt und auf Fairness, Performance und Nutzbarkeit hin verglichen. Seit einiger Zeit befindet sich für gRPC eine weitere Möglichkeit zum Loadbalancing in Entwicklung, welche auf der Technologie xDS von Envoy [13], welche auch vom Service Mesh verwendet werden, basiert. Sie reduziert den Overhead, welcher ein Service Mesh mit sich bringt, bietet jedoch bisher keine vernünftige Möglichkeit, mit vorhandenen Tools bereitgestellt zu werden. So muss für jede Implementierung des xDS-Protokolls eine eigene xDS-Registry entwickelt werden, welche Service Discovery betreibt. Diese Technologie bietet eine vielversprechende, ressourcen sparende Erleichterung gegenüber Service Meshes.

Literatur

[1] Himanshu Agrawal. Networking in kubernetes. In *Kubernetes Fundamentals: A Step-by-Step Development and Interview Guide*, pages 211–247. Springer, 2023.

[2] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. May 2000.

[3] Romuald Corbel, Emile Stephan, and Nathalie Omnes. Http/1.1 pipelining vs http2 in-the-clear: Performance comparison. In *2016 13th International Conference on New Technologies for Distributed Systems (NOTERE)*, pages 1–6, 2016.

[4] Hugues de Saxcé, Iuniana Oprescu, and Yiping Chen. Is http/2 really faster than http/1.1? In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 293–299, 2015.

[5] The Linux Foundation. Kubernetes components, 2023.

[6] The Linux Foundation. Service, 2023.

[7] Mike Gancarz. *Linux and the Unix philosophy*. Digital Press, 2003.

[8] Lars Hick. Lastverteilung von http/2-anfragen in kubernetes, 2023.

[9] Arne Koschel, Marvin Bertram, Richard Bischof, Kevin Schulze, Marc Schaaf, and Irina Astrova. A look at service meshes. In *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*, pages 1–8. IEEE, 2021.

[10] Google LLC. Protocol buffers version 3 language specification, 2023.

[11] Henrik Nielsen, Roy T. Fielding, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.0. May 1996.

[12] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – http/1.1. June 1999.

[13] Envoy Project. xds rest and grpc protocol, 2023.

[14] Eric Rescorla. Http over tls. May 2000.

[15] Martin Thomson and Cory Benfield. Http/2. June 2022.

Ich erkläre, dass ich die vorliegende Abschlussarbeit mit dem Thema *Lastverteilung von HTTP/2-Anfragen in Kubernetes* selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich, inhaltlich oder sinngemäß entnommenen Stellen als solche den wissenschaftlichen Anforderungen entsprechend kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft und ausschließlich für Prüfungszwecke gespeichert wird. Außerdem räume ich dem Lehrgebiet das Recht ein, die Arbeit für eigene Lehr- und Forschungstätigkeiten auszuwerten und unter Angabe des Autors geeignet zu publizieren.

Hagen, den (Abgabedatum)

Lars Hick