Hochschule Bremerhaven

Fachbereich II
Management und Informationssysteme
Wirtschaftsinformatik B.Sc.

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science

Hochverfügbare Multi-Cloud-Infrastruktursysteme

Entwicklung und Bereitstellung von hochverfügbaren Webanwendungen

---

Vorgelegt von: Fabian Lignitz
Leon Sander
Vorgelegt am: 14. November 2023
Erstgutachter: Prof. Dr.-Ing. Oliver Radfelder
Zweitgutachterin: Prof. Dr.-Ing. Karin Vosseberg

Abstract

Die vorliegende Bachelorarbeit untersucht die Herausforderungen und Abhängigkeiten, die sich aus der weitverbreiteten Nutzung von Cloud-Diensten, insbesondere von großen Anbietern wie Google, Amazon und Microsoft, ergeben. Mit Milliarden von monatlichen Zugriffen auf ihre Plattformen sind diese Unternehmen nicht nur zentrale Akteure im Internet, sondern auch essenzielle Dienstleister für zahlreiche Unternehmen weltweit.

Die zentrale Fragestellung der Arbeit besteht darin, wie eine Infrastruktur gestaltet werden kann, die nicht allein von einem einzigen Cloud-Provider abhängig ist. Insbesondere wird der Fokus darauf gelegt, ein Failover-System zu konzipieren, das den reibungslosen Betrieb der Anwendungen auch im Falle eines Ausfalls eines Cloud-Dienstleisters gewährleistet. Die Zielsetzung dabei ist die Minimierung der Ausfallzeit und die Gewährleistung einer hochverfügbaren Infrastruktur. Dabei soll, so weit es möglich ist, auf proprietäre Lösungen verzichtet werden.

Die Arbeit befasst sich nicht nur mit der Gestaltung der Server- und Netzwerkinfrastruktur, sondern auch mit der Entwicklung einer robusten Anwendung, die grundlegende Technologien wie die Anbindung an eine Datenbank und die Verwendung von PHP-Sessions integriert. Die Struktur der Infrastruktur wird dabei von der DNS-Ebene ausgehend aufgebaut, um einen Single Points of Failure zu vermeiden.

Die kritischsten Komponenten, die HAProxy-Server und der Failover-Prozess über DNS, werden intensiv analysiert. Die Tests zeigen, dass der Failover-Prozess im schlechtesten Fall zwischen 30 und 90 Sekunden dauert, wobei die DNS-Resolver die Änderungen des A-Records erstaunlich schnell übernehmen.
Die Diskussion hebt hervor, dass die Replikation des Datenbankservers im Rahmen der Tests gut funktioniert hat, wobei potenzielle Risiken im Zusammenhang mit inkonsistenten Datenständen erörtert werden. Es wird betont, dass das größte Optimierungspotenzial im Übertragungsprozess der Sessions und der Funktionalität des DNS-Watcher liegt, bei dem unterschiedliche Mechanismen eine deutliche Beschleunigung ermöglichen könnten.

Abschließend kann gesagt werden, dass die gestellten Anforderungen an eine hochverfügbare Multi-Cloud-Infrastruktur erfüllt wurden. Die Ergebnisse der Failover-Tests haben die Erwartungen übertroffen, und die Ausfallzeiten von 30 bis 90 Sekunden werden als akzeptabel bewertet. Trotz identifizierter Verbesserungsmöglichkeiten wird die Zielsetzung der Arbeit insgesamt als erfüllt angesehen.

Inhaltsverzeichnis

1 Hinführung zum Arbeitsthema 5

2 Aufbau der Infrastruktur 7
2.1 Vorüberlegungen 7
2.2 Anforderungen an die Infrastruktur 8
2.3 Einstiegspunkt der Infrastruktur 9
2.3.1 Failover mittels Keepalived 10
2.3.2 Failover mittels DNS 13
2.3.3 Abwägung der Failover-Mechanismen DNS und Keepalived . . . 14
2.3.4 Umsetzung DNS 15
2.3.5 Konfiguration der Domain 17
2.4 Schnittstelle zu den Cloud-Providern 19
2.4.1 Google-Cloud Platform 19
2.4.2 Hetzner Cloud 22
2.5 Konfiguration der VMs 23
2.5.1 Erstellung der virtuellen Maschinen 24
2.5.2 Konfiguration von Nftables 25
2.5.3 Konfiguration von Fail2ban 26
2.6 Aufbau der Netzwerkinfrastruktur 27
2.7 Integration eines HAProxy 28
2.7.1 Anwendungsbereich in der Infrastruktur 29
2.7.2 Konfiguration von HAProxy 30
2.8 Aufbau der Watcher-Infrastruktur 31
2.9 Einrichtung der Anwendungsserver 34
2.9.1 Vorbereitung der VMs für die Anwendung 34
2.9.2 Übertragung der Sessions zwischen den Anwendungsservern . . . 35
2.10 Aufbau des Datenbankmanagementsystems 37
2.10.1 Grundlagen einer Datenbankreplikation 38
2.10.2 Installation von MariaDB 40
2.10.3 Vorbereitung der TLS-Verschlüsselung 40
2.10.4 Einrichtung der DB-Replikation mit TLS 42
2.11 Gesamtübersicht der aufgebauten Infrastruktur 44

3 Entwicklung einer Testanwendung 46
3.1 Anforderungen an die Anwendung 46
3.2 Implementierung der Testanwendung 47
3.3 Testen der Anwendungsimplementierung 51

4 Untersuchung der Verfügbarkeit 54
4.1 Definition der Testszenarien 54
4.2 Testen des Failover auf Layer 1 56
4.3 Testen des Failover auf Layer 2 56
4.4 Testen des Failover auf Layer 3 60
4.5 Testen des Failover auf Layer 4 62

5 Diskussion der Ergebnisse 66

6 Resümee und Ausblick 69

Literaturverzeichnis 71

Listingverzeichnis 75

Abbildungsverzeichnis 76

Tabellenverzeichnis 76

Abkürzungsverzeichnis 77

1 Hinführung zum Arbeitsthema

Die meistbesuchten Webseiten der Welt werden monatlich milliardenfach aufgerufen [1]. Allein Googles Suchmaschine *google.de* wird schätzungsweise 85 Milliarden mal im Monat von Menschen besucht. Gehen wir der einfachheitshalber von einer zeitlichen Gleichverteilung der Aufrufe aus, dann entspricht dies ca. 33.000 Aufrufen je Sekunde. Zusätzlich kommen noch weitere Dienste des Google Konzerns hinzu, welche sich ebenfalls unter den meistbesuchten Seiten der Welt befinden und bspw. im Falle von YouTube eine große Menge an transferierten Daten bedeuten.
Ähnliche Rechnungen lassen sich für die Dienste von Meta, Amazon oder Microsoft aufstellen. Gleichzeitig bieten gerade Google, Amazon und Microsoft Dienstleistungen als Cloud-Provider an [2]. Neben der Serverinfrastruktur für die eigenen Dienste sind sie damit verantwortlich für eine Vielzahl von Diensten unzähliger Unternehmen. Als Konsequenz besteht eine enorme Abhängigkeit von diesen Anbietern, nicht nur im wirtschaftlichen Sinne, sondern auch für jede Person, die das Internet nutzt.
So wird vor allem Amazon Web Services (AWS) als unsichtbares Rückgrat des Internets [3] bezeichnet und auch die Frankfurter Allgemeine Zeitung [4] ermittelte, dass 80% der DAX Konzerne AWS nutzen. Ebenso setzen andere Tech-Giganten wie Apple auf die Nutzung von AWS, dies verstärkt die Abhängigkeit des gesamten Internets und verschiedener Dienstleistungen von diesen Providern.
Fällt einer dieser Dienstleister aus, dann fallen große Bereiche des Internets aus, dies ist auch kein theoretisches Szenario, sondern bereits mehrfach eingetreten [5], [6], [7]. Auch wenn solche Ausfälle selten sind, stellt sich die Frage, wie eigene Anwendungen gegen ein solches Szenario abgesichert werden können und wie es die großen Tech-Konzerne schaffen ihre Anwendungen, quasi ohne oder nur mit sehr geringen Ausfallzeiten, bereitzustellen.

Die vorliegende Bachelorthesis soll sich mit diesen Fragen beschäftigen und schlussendlich ein Konzept für eine Infrastruktur liefern, die im Gesamten nicht von einem einzigen Cloud-Provider abhängig ist bzw. ein Failover-System in einer anderen Cloud-Infrastruktur bereitstellt.
Ziel soll es sein, dass das entworfene System mit minimaler Ausfallzeit wieder erreichbar ist. Der Schwerpunkt dieser Thesis liegt dabei zwar im Entwurf einer Server- und Netzwerkinfrastruktur, jedoch gehört zu einem hochverfügbaren System auch eine entsprechend robuste Applikation. Dazu soll eine Anwendung entworfen werden, welche zwar grundsätzlich eine gewisse Einfachheit besitzt, dennoch grundlegende Technologien verwendet. Zu diesen Technologien gehört für uns die Anbindung an eine Datenbank und die Verwendung von *PHP*-Sessions.
Die Infrastruktur soll beginnend mit dem Domain Name System (DNS) so aufgebaut werden, dass kein Single Point of Failure entsteht. Dafür kommen Technologien wie *HAProxy* und Datenbankreplikation zum Einsatz. Ebenso soll die Infrastruktur inkl. Failover über

verschiedene geografische Regionen verteilt werden.
Viele Architekturkomponenten lassen sich bei Cloud-Providern als Software-as-a-Service einkaufen. Ziel soll es auch sein zu erarbeiten, inwieweit ohne den Einsatz von proprietären und vom Provider abhängigen Diensten eine solche Umsetzung erfolgen kann. Dies wird von den Providern mal mehr oder weniger, vor allem aus betriebswirtschaftlichen Gründen erschwert.

Im Laufe dieser Bachelorthesis wollen wir unsere Vorüberlegungen darstellen und einen theoretischen Hintergrund zu den obigen Fragestellungen geben. Dabei sollen die Fragen beantworten, welche Probleme sich in Bezug auf die Bereitstellung hochverfügbarer Systeme ergeben und wie mögliche Lösungsansätze aussehen könnten, ebenso soll der Begriff Hochverfügbarkeit für uns definiert werden.
Aus diesen Überlegungen werden konkrete Anforderungen an eine Infrastruktur formuliert und diese Überlegungen schließlich in ein konkretes Infrastrukturkonzept überführt. Ebenso soll eine Anwendung, welche den obigen Anforderungen entspricht, entworfen werden.
Hierzu werden unterschiedliche Technologien verglichen und der Entscheidungsprozess dargestellt. Anschließend soll diese Infrastruktur verschiedenen Ausfallszenarien ausgesetzt werden.
Zum Abschluss werden diese Testszenarien ausgewertet, die Ergebnisse dargestellt und diskutiert. Dabei soll auch darauf eingegangen werden, ob sich eine praktische Umsetzung über einen Versuchsaufbau hinaus als realistische Möglichkeit erweist, welche Vor- und Nachteile sich ergeben und wie gut die Umsetzung gelungen ist.

2 Aufbau der Infrastruktur

Wenn über eine hochverfügbare Infrastruktur gesprochen wird, dann muss definiert werden, was unter diesem Begriff zu verstehen ist. Hochverfügbarkeit ist in vielen Bereichen rechtlich und formal definiert. So finden sich Verfügbarkeitsklassen die bestimmte Ausfallzeiten erlauben, sowie unzählige ISO Standards [8].
In der vorliegenden Arbeit sollen keine solchen formalen oder rechtliche Anforderungen erfüllt werden. Vielmehr sollen Anforderungen an eine Infrastruktur definiert werden, welche diese gegen übliche Ausfallszenarien absichert und so für die Nutzenden zu einer hohen Erreichbarkeit führt. Diese Anforderungen und die entsprechende Umsetzung sollen im folgenden erläutert werden.

2.1 Vorüberlegungen

Wird eine Infrastruktur bei einem der großen Cloud-Provider gehostet, dann versprechen die jeweiligen Anbieter eine weltweite (hohe) Verfügbarkeit. Um die Umsetzung machen die jeweiligen Anbieter ein Geheimnis. Im wirtschaftlichen Sinne ist dies verständlich, denn die Kund:innen sollen die Dienstleistungen möglichst bei ihnen einkaufen. Die Kund:innen befinden sich somit in der unangenehmen Position, sich auf eine vom Anbieter bereitgestellte Blackbox verlassen zu müssen. Dies sollte aus unserer Sicht ein gewisses Unbehagen verursachen, da sich in eine totale Abhängigkeit zu dem jeweiligen Provider begeben wird und dies, wie bereits geschildert, nicht immer nur gute Folgen hat.
Gleichzeitig sind die angebotenen Lösungen nicht zur jeweiligen Konkurrenz kompatibel und es muss sich zwangsweise auf diese verlassen oder die Funktionalität selbst bereitgestellt werden. Somit besteht die Wahl, sich auf nur einen Anbieter zu verlassen oder enorme Hürden beim Aufbau einer Cloud-Infrastruktur in Kauf zu nehmen (Vendor lock-in). Da die Anbieter jedoch auch von Ausfällen ihrer eigenen Infrastruktur betroffen sind, bedeutet dies für die Kund:innen im schlimmsten Fall einen Totalausfall ihrer Systeme.

Das Thema Hochverfügbarkeit ist dabei viel älter als die Cloud-Provider und somit existieren Strategien und erprobte Methoden um Hochverfügbarkeit in der Praxis umzusetzen. Möglichkeiten zur Datenbankreplikation existieren bspw. schon deutlich länger als Cloud Computing eine Rolle in der IT spielt. Im Cloud-Bereich wird sich, unserer Erfahrung nach, an vielen Stellen auf das Angebot der Provider verwiesen, während im Inhouse-Bereich die Infrastruktur horizontal oder vertikal skaliert wird, um den Ausfall eines Servers aufzufangen.
Aus unserer Sicht müssten sich diese Strategien mit der grundsätzlichen Funktionalität der Cloud-Provider, also dem Bereitstellen von Servern in unterschiedlichen (geografischen) Regionen, kombinieren lassen.

Unser Ziel soll sein, dass eine Infrastruktur entsteht, welche nicht auf die vorgefertigten (proprietären) Lösungen der Anbieter setzt, sondern als verteiltes System über mehrere Anbieter hinweg die klassischen und erprobten Strategien für eine robuste und hochverfügbare Infrastruktur und Anwendung nutzt. Dabei müssen natürlich Besonderheiten des Cloud-Hosting berücksichtigt werden. Aus diesen Vorüberlegungen haben wir Anforderungen an eine Infrastruktur entworfen.

2.2 Anforderungen an die Infrastruktur

Vor der technischen Beschreibung zum Aufbau der hochverfügbaren Multi-Cloud-Infrastruktur, ist es wichtig, die Anforderungen festzulegen, die an diese Infrastruktur gestellt werden:

- **Skalierbarkeit:** Die Infrastruktur sollte die Fähigkeit besitzen, sich flexibel an steigende oder fallende Arbeitslasten anzupassen. Skalierung sollte sowohl horizontal (Hinzufügen von Ressourcen) als auch vertikal (Erhöhen der Ressourcenleistung) möglich sein [9].
- **Redundanz und Ausfallsicherheit:** Die Infrastruktur muss so entworfen sein, dass sie Ausfälle von Komponenten oder sogar ganzen Cloud-Regionen auffangen kann, ohne den Betrieb zu beeinträchtigen. Redundante Ressourcen, Lastenausgleich und Failover-Mechanismen sind notwendig, um die Verfügbarkeit sicherzustellen.
- **Geografische Verteilung:** Die Infrastruktur sollte über mehrere geografische Regionen oder Cloud-Anbieter verteilt werden, um das Risiko von regionalen Ausfällen zu minimieren. Geografische Verteilung kann auch zur Verbesserung der Latenz für verschiedene Standorte beitragen.
- **Automatisierung:** Ein Großteil der Infrastrukturverwaltung sollte automatisiert sein, um die Bereitstellung, Skalierung, Wartung und Fehlerbehebung zu vereinfachen. Automatisierung ermöglicht schnelle Reaktionen auf Änderungen und Probleme.
- **Sicherheit:** Die Infrastruktur muss robuste Sicherheitsmaßnahmen umfassen, um Datenintegrität, Vertraulichkeit und Verfügbarkeit zu gewährleisten. Identitäts- und Zugriffsverwaltung, Verschlüsselung, Firewalls und Sicherheitsüberwachung sind wichtige Aspekte.
- **Performance:** Die Infrastruktur sollte eine konsistente und akzeptable Leistung bieten, unabhängig von den Lastbedingungen. Latenzzeiten, Durchsatz und Antwortzeiten sollten optimiert werden.

- **Kostenoptimierung:** Die Infrastruktur sollte so entworfen sein, dass Ressourcen effizient genutzt werden, um unnötige Kosten zu vermeiden.
- **Interoperabilität:** Die Infrastruktur sollte die Möglichkeit bieten, unterschiedliche Cloud-Anbieter und Technologien nahtlos miteinander zu verbinden. Dies ermöglicht Flexibilität und verhindert eine zu starke Abhängigkeit von einem einzelnen Anbieter.
- **Monitoring und Diagnose:** Die Infrastruktur sollte umfangreiche Überwachungsfunktionen bieten, um Probleme zu erkennen und rechtzeitig darauf reagieren zu können. Protokollierung, Metriken und Alarme sind hierfür unverzichtbar.

Die oben skizzierten Anforderungen bilden unseres Erachtens das Fundament für den Aufbau einer robusten Multi-Cloud-Infrastruktur. Ein ausgewogenes Zusammenspiel von Skalierbarkeit, Sicherheit, Leistung und Automatisierung soll die Grundlage für den reibungslosen Betrieb in einer dynamischen Umgebung bilden. Im nächsten Abschnitt soll der Aufbau der Infrastruktur näher beschrieben und die Schlüsselkomponenten beleuchten werden, die diese Anforderungen erfüllen.
Trotz den zuvor aufgestellten Anforderungen ist das Leben der Open-Source-Kultur in der Cloud-Welt die Wichtigste. Auch wenn dadurch technische, oder funktionelle Nachteile entstehen könnten, ist ein Ziel dieser Arbeit zu untersuchen, inwieweit der Aufbau dieser Infrastruktur unter Berücksichtigung dieser Kultur möglich ist.

2.3 Einstiegspunkt der Infrastruktur

Der Einstiegspunkt in die Infrastruktur, also die Schnittstelle, mit der ein potenzieller Client als Erstes eine Verbindung aufnimmt, ist bei der Erstellung einer Infrastruktur als besonders kritisch zu betrachten. Gerade im Kontext von hochverfügbaren Systemen müssen hier viele Punkte beachtet werden, um eine gewisse Robustheit und Zuverlässigkeit bei der Bereitstellung einer Softwareanwendung gewährleisten zu können.

Klassischerweise liegen Webanwendungen hinter einer Domain, welche wiederum auf eine IP-Adresse verweist, hinter der genau ein Server steht. Somit braucht es als Einstiegspunkt in die Infrastruktur eine Domain, einen passenden DNS-Eintrag und einen Server, der die Anfragen entgegennimmt. An dieser Stelle sind die ersten Überlegungen nötig, wie dieser Einstiegspunkt gegen Ausfälle abgesichert werden kann.
Als erstes Szenario muss betrachtet werden ,was passiert, wenn dieser Server ausfällt. In einem solchen Fall wäre die gesamte Anwendung nicht mehr erreichbar und somit auch alle dahinter liegenden Mechanismen obsolet. Dieses Ausfallszenario lässt sich, aus unserer Sicht, durch zwei interessante Möglichkeiten verhindern:

- **Umschwenken per Keepalived:** die öffentliche IP Adresse hinter der Anwendungs-Domain könnte standortübergreifend auf mehreren Servern konfiguriert sein. Per Keepalived könnte dann das Umschwenken auf einen aktiven Server im Falle eines Ausfalls erfolgen.
- **Umschwenken per DNS:** die öffentliche IP-Adresse ist lediglich auf einem Server konfiguriert. Während eines Ausfalls muss eine unabhängige Watcher-Infrastruktur den DNS-Eintrag aktualisieren.

Beide Ansätze haben ihre Vor- und Nachteile hinsichtlich der Komplexität, Skalierbarkeit und Reaktionsgeschwindigkeit. Im Folgenden wird näher betrachtet, wie diese beiden Möglichkeiten die Hochverfügbarkeit und Effizienz einer Multi-Cloud-Infrastruktur beeinflussen können und welche Variante in diesem Fall sinnvoll sein könnte.

2.3.1 Failover mittels Keepalived

Eine bewährte Methode zur Realisierung kontinuierlicher Dienstverfügbarkeit ist die Verwendung eines Failover-Tools, welches das Virtual Router Redundancy Protocol (VRRP) implementiert. *Keepalived* ist eines dieser Tools und kann mit der Hilfe dieses Protokolls redundante Systeme bereitstellen [10, S. 30]. VRRP verwendet konkret für die Realisierung eine Virtual IP-Address (VIP). Bei *Keepalived* wählen alle beteiligten Nodes einen Leader, welcher die virtuelle IP-Adresse erhalten soll und diese an seinem Interface hinzufügt (vgl. Listing 2.1). *Keepalived* bietet außerdem die Möglichkeit, jede Node mit einem Prioritätswert zu konfigurieren. Die verfügbare Node mit dem höchsten Wert erhält schließlich die VIP und ist folglich dafür verantwortlich, auf Requests an diese IP zu antworten [11].

Nach der Konfiguration warten alle Nodes auf regelmäßige Nachrichten des Leaders, durch welche der *Keepalived* Daemon sicherstellen kann, dass der aktive Leader noch verfügbar ist und auf Requests von Clients antworten kann. Würde man den Leader herunterfahren, wird automatisch die Node mit dem zweithöchsten Prioritätswert als neuer Leader eingesetzt werden.
Neben dem Herunterfahren einer Node lassen sich in *Keepalived* auch Skripte hinterlegen, mit denen der Daemon ebenfalls einen Zustand melden kann, der zum Umschwenken führt [11]. Beispielsweise könnte man mit so einem Skript überprüfen, ob ein Webserver noch einen Process identifier (PID) hat. Wenn nicht, lässt sich per Rückgabe des passenden Exit-Codes ein Umschwenken forcieren, wodurch eine andere Node zum Leader werden würde, auf der dieser Webserver noch aktiv ist.

```
root@10:~# ip -br a
lo               UNKNOWN        127.0.0.1/8 ::1/128
enp1s0           UP             10.10.0.10/24 10.10.0.9/32
↪  fe80::5054:ff:fe00:a/64
```

Listing 2.1: VIP-Konfiguration Keepalived

Da *Keepalived* lediglich auf jeder Node läuft und deren Zustand überwacht, braucht es einen Dienst, der die Request auch tatsächlich entgegennimmt. Insofern ist *Keepalived* alleine für die Absicherung des Einstiegspunktes in die Infrastruktur nicht ausreichend, es braucht einen Proxy, der den Traffic entgegennimmt und entsprechend an die Anwendungsebene weiterleitet.

Ein solcher Proxy könnte bspw. *HAProxy* sein. Wenn nur ein solcher Proxy zum Einsatz kommen würde, würde man den Single Point of Failure jedoch nur eine Schicht nach unten verlagern. Bei Ausfall des Proxys steht kein Failover bereit, um die Requests weiterzugeben, auch wenn *Keepalived* den Ausfall einer Node bemerkt. *Keepalived* löst somit zwar das Problem von Totalausfällen ganzer Nodes auf TCP/IP Ebene, aber erfordert ein weiteres Vorgehen auf jeder einzelnen Node.

Als Lösung für dieses Problem steht dabei die Möglichkeit zur Verfügung, dass per *Keepalived* eine VIP konfiguriert werden kann, welche mehreren *HAProxy*-Nodes zugeordnet werden kann [11]. Bei einem Ausfall einer Node kann durch ein Umschwenken ein Ausfall verhindert werden. Es gilt dabei zu betonen, dass die Nodes in allen Fällen über redundante Proxys verfügen müssen und dies sich nicht durch die Verwendung von *Keepalived* ergibt.

Die tatsächliche Besonderheit bei der Konfiguration von *Keepalived* ist, dass das Netzwerk der zu konfigurierenden VIP auf dem physischen Interface des Servers auch zur Verfügung stehen muss. Das stellt bei kleineren Infrastrukturen, oder Servern, die sowieso im selben Netzwerk stehen, kein großes Problem dar. Im Cloud-Kontext hat die *Keepalived* Konfiguration also eine andere Komplexität. Hier müssen wir standort- und providerübergreifend Firewallregeln konfigurieren und es schaffen eine IP-Adresse, bzw. ein IPv4-Netz zu verwalten, welches an verschiedenen Servern verschiedenster Standorte an das entsprechende Interface gehängt werden kann.
Für diesen Anwendungsfall gibt es das Border Gateway Protocol (BGP) Routing, mit dem so eine standortübergreifende Konfiguration desselben öffentlichen IPv4-Netzwerks umgesetzt werden kann [12]. Die großen Cloud-Provider haben dafür bereits entsprechende Schnittstellen entwickelt [13].

Wie in Abbildung 2.1 skizziert, könnte so eine Multi-Cloud-Infrastruktur mit der Konfiguration einer geteilten, öffentlichen BGP-IP beginnen, welche auf zwei *HAProxy*-Nodes die per Keepalived verwaltet werden verweist. Ein HTTP-Request würde in diesem Fall an dem aktiven *HAProxy*, also an der Node mit der konfigurierten VIP ankommen und an einen App-Server weitergeleitet werden.
Zur Optimierung von Latenzzeiten kann hier priorisiert von der *HAProxy*-Node der App-Server beider Locations angesprochen werden. Im Falle eines Ausfalls vom App-Server der Location-1 würde dann nur noch der App-Server aus Location-2 angesprochen werden. Im Beispiel wird für die Kommunikation der *Keepalived*-Nodes untereinander ein Virtual Private Network (VPN) Tunnel gebaut, welcher Multicast Traffic gestattet.

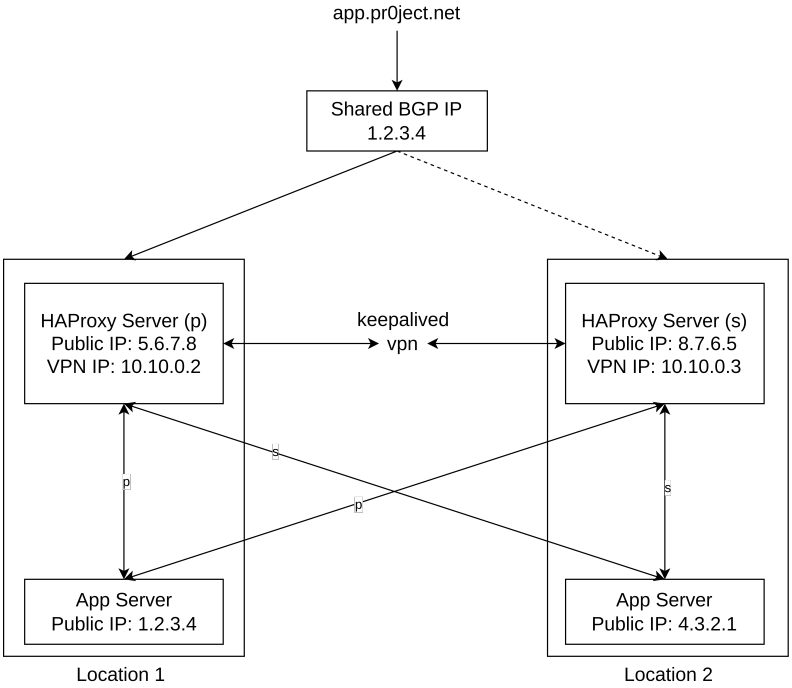

Abbildung 2.1: Einstiegspunkt der Infrastruktur mit Keepalived

Mit *Keepalived* lässt sich insgesamt ein stabiles System bauen. Konkret können in diesem Beispiel genau 50% der beteiligten Nodes ausfallen, ohne dass dies eine Auswirkung auf die Verfügbarkeit der Anwendung haben würde. Zusätzlich erfolgt das Umschwenken automatisiert und daher sehr schnell. Im Kontext von Cloud-Architekturen ist aber die Komplexität mit BGP, VPN für die Multicast Kommunikation, sowie die Konfiguration der Firewalls nicht zu unterschätzen.

2.3.2 Failover mittels DNS

Eine weitere Möglichkeit zum Umschwenken des aktiven Einstiegspunktes im Fehlerfall einer Multi-Cloud-Infrastruktur ist lediglich die Verwendung von DNS. Die Architektur von DNS ist bereits dezentral gestaltet und eignet sich damit optimal als Werkzeug in diesem Kontext [14].
Dies wird besonders deutlich bei der Gestaltung von Infrastrukturen mit mehreren Standorten, bei denen die Vorteile von DNS in Verbindung mit der Verwendung eines Watcher-Systems greifbar werden. So wird im Folgenden eine dynamische Handhabung der DNS-Auflösung anhand eines Szenarios, in dem die Hochverfügbarkeit von Diensten über zwei Standorte hinweg gewährleistet werden muss, betrachtet.

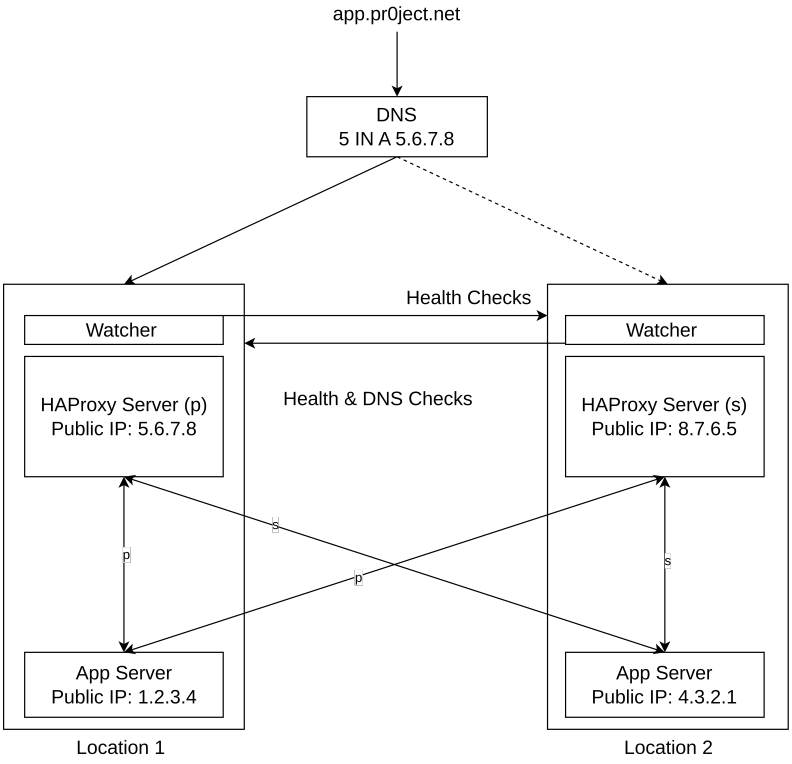

Abbildung 2.2: Einstiegspunkt der Infrastruktur mit DNS

In der Abbildung 2.2 wird die Architektur dieser Variante für den Einstiegspunkt genauer skizziert. Die Domain *app.pr0ject.net* zeigt hier auf die öffentliche IPv4-Adresse 5.6.7.8. Dies ist die IP-Adresse des *HAProxy*-Node aus Location-1. Diese *HAProxy*-Node wird also bei einem potenziellen HTTP-Request auf die Domain angesprochen und leitet an den als primär (p) priorisierten App-Server derselben Location weiter. Sollte dieser App-

Server ausfallen, ist eine Weiterleitung an den sekundären (s) App-Server aus Location-2 ebenfalls möglich.

Eine isolierte und unabhängige Watcher-Infrastruktur wird hier regelmäßig die Verfügbarkeit von *app.pr0ject.net* überprüfen und gegebenenfalls im DNS den A-Record auf die IP-Adresse 8.7.6.5 umkonfigurieren. Die Time to live (TTL) bei einem DNS A-Record gibt an, wie lange die IP-Adresse in Zwischenspeichern und Caches behalten werden soll, bevor sie aktualisiert wird [14]. Durch die Verwendung einer sehr geringen TTL ist somit in der Theorie ein recht schnelles Umschwenken der *HAProxy*-Node von Location-1 nach Location-2 möglich.

In der Praxis kann jedoch ein Cache Problem auftreten, das die Umschaltung verlangsamt und die Performance folglich beeinträchtigen kann, da einige DNS-Caches die TTL möglicherweise nicht strikt einhalten und somit über veraltete Informationen verfügen.
Die Art und Weise, wie DNS-Caches in verschiedenen Teilen der Welt konfiguriert sind, kann variieren. Einige Internet Service Provider (ISP) oder DNS-Server Betreiber könnten aggressivere Caching-Richtlinien implementieren, um die Netzwerkleistung zu optimieren. Dies könnte dazu führen, dass TTL-Werte in einigen Regionen weniger strikt eingehalten werden.
Insgesamt eignet sich DNS allerdings sehr gut für die Gestaltung des Einsteigpunktes einer hochverfügbaren Infrastruktur. Gerade durch die Dezentralität ist DNS so gestaltet, dass DNS-Abfragen an nahezu jedem Standort und jeder Anwendung ausgelöst werden können.
Dies ermöglicht eine breite Abdeckung und Verteilung der Zugriffe, was besonders wichtig ist, um den Ausfall einzelner Ressourcen oder Standorte zu kompensieren. Die Implementierung ist vergleichsweise einfach und erfordert keine komplexen Änderungen auf Anwendungsebene.
Zusätzlich ermöglicht die Verwendung niedriger TTL-Werte eine schnelle Reaktion auf Ausfälle und eine zügige Umleitung des Datenverkehrs auf funktionierende Ressourcen. Schließlich erlaubt die Unabhängigkeit von Drittanbietern oder Clustermanagern eine flexible und kosteneffiziente Implementierung zur Umsetzung der gestellten Anforderung.

2.3.3 Abwägung der Failover-Mechanismen DNS und Keepalived

Keepalived ist eine bewährte Methode zur Gewährleistung der Hochverfügbarkeit von Diensten in verteilten Umgebungen. Es verwendet Multicast-Health-Checks, um kontinuierlich den Status der verschiedenen Nodes zu überwachen. Diese Health-Checks ermöglichen eine enorm schnelle Erkennung von Ausfällen oder Problemen in Echtzeit. Sobald ein Ausfall erkannt wird, schaltet *Keepalived* automatisch den Datenverkehr auf die verbleibende, funktionierende Node um.

Dieser schnelle und automatisierte Umschaltprozess ist ein entscheidender Vorteil von *Keepalived* und trägt dazu bei, die Ausfallzeit auf ein Minimum zu reduzieren. *Keepalived* hat sich einen Ruf für seine Zuverlässigkeit und Geschwindigkeit bei der Fehlererkennung erworben, so bieten viele große Anbieter wie IBM [15], Oracle [16], RedHat [17] und VMWare [18] Dokumentationen, die aufzeigen, wie man *Keepalived* in Kombination mit *HAProxy* nutzt.

Im Gegensatz dazu setzt die DNS basierte Lösung auf die Umleitung des Datenverkehrs bei Ausfällen. Hierbei wird der DNS-Eintrag auf eine andere IP-Adresse umgeleitet, um den Datenverkehr auf eine funktionierende Node zu lenken. Obwohl diese Methode funktioniert, gibt es einige Herausforderungen. So kann die Umleitung des Datenverkehrs durch DNS-Caching und die TTL-Verzögerungen beeinflusst werden. Dies bedeutet, dass die Kontrolle über den Failover-Prozess weniger granular ist als bei *Keepalived*. Trotzdem bietet diese Option den Vorteil einer einfacheren Einrichtung und erfordert in der Regel weniger Konfigurationsaufwand. Dies kann insbesondere für kleinere Unternehmen mit begrenzten Ressourcen attraktiv sein und ist in der Regel kostengünstiger.
Insgesamt sind beide Optionen gültige Ansätze zur Steigerung der Hochverfügbarkeit von Diensten. *Keepalived* mit BGP und Multicast-Health-Checks gilt als robust und zuverlässig, während die DNS-basierte Lösung durch ihre einfache Implementierung und niedrigeren Kosten besticht.

Keepalived benötigt für die in dieser Arbeit angestrebte Infrastruktur zusätzlichen Aufwand, da der Einsatz über interne Netze hinweg nicht vorgesehen ist. So müssten die Netzwerke zweier Cloud-Provider mit einem gemeinsamen internen Netzwerk unterbaut werden. Darüber hinaus spielt die Anforderung an die Verfügbarkeit eine Rolle. Muss eine sehr hohe Verfügbarkeit erreicht werden, dann sollte *Keepalived* bevorzugt werden. Sind hingegen Ausfallzeiten in etwas größerem Ausmaß zu verkraften, stellt ein Failover über DNS eine geeignete Methode dar.
Da es sich bei diesem Projekt um eine Bachelorarbeit handelt, die nicht im Kontext eines Unternehmens durchgeführt wird, wird hier die kostengünstige Variante vorgezogen. Auch im Hinblick auf das Leben der Open-Source-Kultur ist dies die bessere Variante, da im Kontext von BGP und Hochverfügbarkeit in der Cloud-Welt schnell proprietäre Produkte eingesetzt werden, die es hier zu vermeiden gilt. So stehen zwar bestimmte Schnittstellen bereit, diese sind aber allzu oft nicht Open-Source.

2.3.4 Umsetzung DNS

Um nun tiefer in die Umsetzung mit der zuvor ausgewählten Option einzusteigen, müssen zunächst ein paar grundlegende Definitionen erstellt werden. DNS ist ein grundlegender Bestandteil des Internets, der für die Auflösung von Domainnamen in IP-Adressen und

umgekehrt verantwortlich ist. Hier sind einige wichtige Begriffe im Zusammenhang mit DNS und eine Erklärung zu deren Bedeutung:

- **A-Record (Address Record)** Der A-Record ist ein DNS-Eintragstyp, der verwendet wird, um eine Domain mit einer IPv4-Adresse zu verknüpfen. Wenn beispielsweise *www.pr0ject.net* in einen Webbrowser eingeben wird, wird ein DNS A-Record verwendet, um die IP-Adresse des Servers zu finden, auf dem die Webseite gehostet wird [14].
- **Nameserver (NS)** Ein Nameserver ist ein DNS-Server, der für die Speicherung und Bereitstellung von DNS-Einträgen für eine bestimmte Domain verantwortlich ist. Er fungiert als Autorität für die DNS-Zone dieser Domain und kann Anfragen zur Auflösung von Domainnamen beantworten [14].
- **SOA (Start of Authority)** Der Start of Authority (SOA)-Eintrag ist ein spezieller DNS-Datensatz, der Informationen über die Autorität und die grundlegenden Konfigurationseinstellungen für eine DNS-Zone enthält. Dies umfasst Informationen wie den Verantwortlichen (Admin-C), die Seriennummer der Zone, das Refresh-Intervall und den Nameserver für die Zone. Insgesamt gibt der SOA-Eintrag den Startknoten, hier den ersten verantwortlichen Nameserver für eine Domain an [14].

In Bezug auf die Umsetzung der DNS-Infrastruktur wurde sich für eine Kombination aus Google Cloud DNS und Hetzner entschieden. Das Kriterium für die Auswahl der passenden Cloud-Provider ist hier hauptsächlich die Möglichkeit, die TTL der Domain per Application Programming Interfaces (API), oder command-line interface (CLI) selbst und ohne Kontakt zum Support zu konfigurieren.
Google Cloud DNS wird als SOA verwendet, was bedeutet, dass es die zentrale Autorität für die DNS-Zone der Domain *app.pr0ject.net* ist. Wird Google Cloud DNS als einziger Dienst genutzt, entsteht jedoch ein neuer Single Point of Failure. Dies ist auch durch die von uns genutzte TTL verursacht.

Wird die TTL, wie üblich, auf Werte von einer Stunde oder höher gesetzt, sorgt die dezentrale Struktur die DNS zu Grunde liegt dafür, dass die DNS-Resolver weiterhin auf die korrekte IP-Adresse verweisen. Dies verhindert jedoch ein schnelles Ändern der hinterlegten IP-Adresse, da die Änderung erst wirksam wird, wenn die TTL abgelaufen ist. Um dieser Problematik zu begegnen, muss ein zusätzlicher Provider als Nameserver (NS) fungieren.

In der hier beschriebenen Infrastruktur wird Hetzner als zusätzlicher Nameserver fungieren, der Redundanz und Ausfallsicherheit bietet. Bei einem Ausfall der kompletten Google-Cloud ist Hetzner in der Theorie immer noch in der Lage, auf DNS-Abfragen eines potenziellen Clients zu antworten.

Wenn eine DNS-Anfrage gesendet wird, durchläuft sie eine Hierarchie von DNS-Servern, beginnend beim Root-Nameserver und dann zu den Top-Level-Domain (TLD) und schließlich zu den Nameserver, die für die spezifische Domain zuständig sind [14].
Die Verwendung eines zweiten Nameservers bei einem anderen Cloud-Provider wie Hetzner trägt zur Ausfallsicherheit bei, da im Falle eines Ausfalls des einen Providers der andere weiterhin DNS-Abfragen bearbeiten kann. Dies erhöht die Zuverlässigkeit und Verfügbarkeit der DNS-Infrastruktur erheblich.
Die Umsetzung eines zusätzlichen Nameserver mit einem fremden SOA ist oftmals nicht ganz unkompliziert, da wenige Anbieter dies ermöglichen, auch deshalb fiel die Wahl auf Hetzner.

2.3.5 Konfiguration der Domain

Um die DNS-Zone, welche alle möglichen Subdomains und Ressourceneinträge beinhaltet, für die Zone *pr0ject.net* zu konfigurieren, muss die Zone zunächst bei einem entsprechenden Anbieter reserviert werden. In diesem Fall haben wir uns für die Konfiguration des SOA-Records bei Google Cloud DNS entschieden.

```
1 ~$ dig SOA pr0ject.net.
2 ; <<>> DiG 9.18.16-1-Debian <<>> SOA pr0ject.net.
3 ;; ANSWER SECTION:
4 pr0ject.net. 86075 IN SOA ns-cloud-e1.googledomains.com.
  ↪  cloud-dns-hostmaster.google.com. 2023071400 21600 3600 259200 300 UP
  ↪  10.10.0.10/24 10.10.0.9/32 fe80::5054:ff:fe00:a/64
```

Listing 2.2: SOA-Record per dig abfragen

Im Listing 2.2 wird mit der Hilfe des Tools Domain Information Groper (dig) [19] validiert, dass die SOA-Records tatsächlich auf die Nameserver von Google verweisen. Auch hier ist eine zusätzliche Vorsichtsmaßnahme getroffen, da zwei Google Nameserver einen SOA-Record für die Zone konfiguriert haben. Per dig lässt sich mit einer Abfrage auf die NS-Records auch überprüfen, welche Nameserver dafür verantwortlich sind, Informationen über die Zone auszuliefern.

```
~$ dig @a.gtld-servers.net NS pr0ject.net.
; <<>> DiG 9.10.6 <<>> @a.gtld-servers.net NS pr0ject.net.
;; AUTHORITY SECTION:
pr0ject.net.    172800  IN  NS  ns-cloud-e1.googledomains.com.
pr0ject.net.    172800  IN  NS  ns-cloud-e2.googledomains.com.
pr0ject.net.    172800  IN  NS  ns-cloud-e3.googledomains.com.
pr0ject.net.    172800  IN  NS  ns-cloud-e4.googledomains.com.
pr0ject.net.    172800  IN  NS  helium.ns.hetzner.de.
pr0ject.net.    172800  IN  NS  hydrogen.ns.hetzner.com.
pr0ject.net.    172800  IN  NS  oxygen.ns.hetzner.com.
```

Listing 2.3: NS-Record per dig abfragen

In dem Listing 2.3 wird deutlich, dass hierfür sowohl einige Server vom Google Cloud DNS, als auch Server vom deutschen Cloud-Provider Hetzner eingetragen sind. Ohne die Konfiguration der NS-Records wäre es Hetzner nicht möglich, im Falle einer Störung beim Google Cloud DNS, auf Anfragen zu der DNS-Zone zu antworten.

```
~$ dig @ns-cloud-e1.googledomains.com A app.pr0ject.net
; <<>> DiG 9.10.6 <<>> A pr0ject.net
;; ANSWER SECTION:
app.pr0ject.net.    54  IN  A  188.34.181.227
```

Listing 2.4: A-Record per dig abfragen für Google NS

```
~$ dig @helium.ns.hetzner.de. A app.pr0ject.net
; <<>> DiG 9.10.6 <<>> A pr0ject.net
;; ANSWER SECTION:
app.pr0ject.net.    54  IN  A  188.34.181.227
```

Listing 2.5: A-Record per dig abfragen für Hetzner NS

Die verschiedenen Nameserver von Google und Hetzner sind nun aber auch in der Lage, unabhängig voneinander, den A-Record der Domain *app.pr0ject.net* zurückzugeben. Also müssen ebenfalls an beiden Stellen bei einem Umschwenken die A-Records von der

Watcher-Komponente aktualisiert werden. Nur wenn in allen beteiligten Nameserver alle relevanten Records gepflegt werden, sind die Nameserver als unabhängig voneinander zu betrachten [14].
Im Listing 2.4 bzw. 2.5 wird genau dies verdeutlicht. Mit der zusätzlichen Übergabe der Parameter `@ns-cloud-e1-googledomains.com` und `@helium.ns-hetzner.de` wird außerdem sichergestellt, dass exakt der angegebene Nameserver für den Request gefragt wird [19].
Insgesamt wird durch die vorgenommenen DNS-Konfigurationen eine gewisse Stabilität des Einstiegspunktes in die Infrastruktur sichergestellt. Selbst bei einem Ausfall vom Google Cloud DNS können die Nameserver der Hetzner Cloud einspringen und einen störungsfreien Betrieb gewährleisten, da die Domain *app.pr0ject.net* weiter aufgelöst werden kann.

2.4 Schnittstelle zu den Cloud-Providern

Die Erstellung von Infrastrukturkomponenten in der Cloud wird in diesem Projekt grundsätzlich so umgesetzt, dass sie automatisierbar ist. Das bedeutet, dass eine wiederholbare Ausführung per Skript möglich sein muss. Die skriptbasierte Erstellung von Ressourcen ermöglicht bei Bedarf eine schnelle Skalierung, da eine neue Virtual Machine (VM) bei steigender Last problemlos hinzugefügt werden kann.
Da im Rahmen dieses Projektes unter anderem Ausfälle, also das komplette Verschwinden von virtuellen Maschinen und dessen Auswirkung auf die Performance der Infrastruktur untersucht werden sollen, eignet sich die skriptbasierte Erstellung von virtuellen Maschinen hier umso mehr. So ist auch eine gewisse Konsistenz und Fehlerreduzierung sichergestellt. Ressourcen werden immer genau so konfiguriert, wie dies in den Skripten angegeben ist.

2.4.1 Google-Cloud Platform

Um die zuvor genannten Vorteile auch konkret am Beispiel der Google-Cloud umsetzen zu können, verwenden wir das von Google entwickelte CLI-Tool *gcloud*. Dieses Tool bietet einige Funktionen zur Erstellung und Verwaltung von Ressourcen in der Google-Cloud [20].

Die in dieser Arbeit vielfach erwähnte Open-Source Kultur widerspricht grundsätzlich der Nutzung des von *gcloud* bereitgestellten CLI. Als Alternative steht ein Werkzeug wie *Curl* bereit und bietet grundsätzlich den großen Vorteil, dass es frei und Open-Source ist. Bei *Curl* handelt es sich um eine plattformunabhängige Befehlszeilenanwendung, während

sich die CLI eines Cloud-Providers auf bestimmte Betriebssysteme beschränkt und ggf. nicht frei und Open-Source sein könnte und oftmals auch nicht ist.

Mit *Curl* ist es möglich HTTP-Anfragen manuell zusammenzustellen und anzupassen, um genaue Anforderungen zu erfüllen. Das ist ein großer Vorteil, da in manchen Fällen die CLI eines Cloud-Providers möglicherweise nicht alle Funktionen, die von der Cloud Umgebung unterstützt werden, beinhaltet. In den Vorbereitungen zu dieser Arbeit wurde beispielsweise die CLI des europäischen Cloud-Providers Exoscale evaluiert.
Zwar hat Exoscale eine gute CLI, dafür aber eine aus unserer Sicht sehr schlechte Dokumentation der API zur Erstellung von Ressourcen. Wenn dort statt der CLI auf die Verwendung von *Curl* gesetzt wird, ist es oft nötig zu raten, welche Parameter bei einem Request übergeben werden müssen. Das bedeutete sehr viel Aufwand, weshalb wir uns gegen Exoscale und damit gegen deren CLI entschieden haben.

Die schlechte Dokumentation von APIs ist ein gutes Beispiel von Vendor lock-in in der Welt des Cloud-Computings. Denn wenn aus zeitgründen auf *Curl* verzichtet wird, entsteht schnell eine Abhängigkeit die dazu führen kann, dass der Aufwand und die Kosten beim Umzug zu einem anderen Cloud-Provider so hoch werden, dass Kund:innen einfach bei ihrem bisherigen Anbieter bleiben, auch wenn dieser technisch vielleicht nicht optimal zu den Anforderungen passt [21]. Mit *Curl* steht im Vergleich ein Tool zur Verfügung, welches für hohe Portabilität sorgt. Egal welche API, von welchem Cloud-Provider, das Werkzeug ändert sich nicht.

Auch wenn die Vorzüge von *Curl* unabstreitbar sind und von Exoscale als Cloud-Provider abstand genommen wurde, findet sich die grundlegende Problematik der (bewussten) Inkompatibilität zu wirklich offenen Schnitstellen bei fast allen Anbietern. Der beschriebene Versuch mittels *Curl* auf die Schnittstellen zu zugreifen, ist so umfangreich, dass sich hierfür eine eigene Bachelorarbeit verfassen lassen könnte. Aus diesem Grund und dem gelegten Schwerpunkt der vorliegenden Arbeit wird auf die von den Anbietern bereitgestellten CLIs zurückgegriffen.

Nachdem die Installation der *gcloud*-CLI abgeschlossen ist, lässt sich per `gcloud init` die CLI konfigurieren und auf das Projekt bei der Google Cloud entsprechend anpassen. Ohne ein authentifiziertes *gcloud* command-line interface ist sowohl die Erstellung, als auch Bearbeitung von Ressourcen in einem Projekt nicht möglich. Die Authentifizierung erfolgt dabei standardmäßig über den Browser, dies ist im Sinne der Automatisierung hinderlich und kann nur durch die Einrichtung von sog. Serviceaccounts umgangen werden.

```
#!/bin/bash

gcloud compute instances create haproxy-one \
    --project=pr0jects-391714 \
    --zone=europe-west2-c \
    --machine-type=e2-medium \
    --network-interface=network-tier=PREMIUM,stack-type=IPV4_ONLY,subnet=locat
    ↪  ion-one
    ↪  \
    --metadata=ssh-keys=ba:ssh-rsa\ ...\ ba@me.local \
    --can-ip-forward \
    --maintenance-policy=MIGRATE \
    --provisioning-model=STANDARD \
    --service-account=600856898992-compute@developer.gserviceaccount.com \
    --scopes=https://www.googleapis.com/auth/devstorage.read_only,https://www.
    ↪  googleapis.com/auth/logging.write,https://www.googleapis.com/auth/moni
    ↪  toring.write,https://www.googleapis.com/auth/servicecontrol,https://ww
    ↪  w.googleapis.com/auth/service.management.readonly,https://www.googleap
    ↪  is.com/auth/trace.append
    ↪  \
    --tags=http-server,https-server \
    --create-disk=auto-delete=yes,boot=yes,device-name=haproxy-one,image=proje
    ↪  cts/debian-cloud/global/images/debian-12-bookworm-v20230912,mode=rw,si
    ↪  ze=10,type=projects/pr0jects-391714/zones/europe-west2-c/diskTypes/pd-
    ↪  balanced
    ↪  \
    --no-shielded-secure-boot \
    --shielded-vtpm \
    --shielded-integrity-monitoring \
    --labels=goog-ec-src=vm_add-gcloud \
    --reservation-affinity=any
```

Listing 2.6: Bereitstellung einer VM mittels gcloud

Nach der Erstellung (siehe Listing 2.6) können die Instanzen ebenfalls über die CLI des Anbieters verwaltet werden. Es steht eine Vielzahl von möglichen Befehlen bereit, bspw. die Auflistung aller Instanzen eines Projektes (siehe 2.7).

```
~$ gcloud compute instances list
NAME         ZONE             MACHINE_TYPE INTERNAL_IP  EXTERNAL_IP  STATUS
dummy  europe-west2-c  e2-micro     10.214.0.2   34.32.6.155  RUNNING
```

Listing 2.7: Auflisten der Google Cloud VMs

Es ist zwar auch möglich, komplett individualisierte VMs zu erstellen, dies ist im Vergleich jedoch äußerst teuer. Dieser Schritt kann sich dennoch lohnen, wenn eine hoch individualisierte Infrastruktur gebraucht wird, also bspw. mit sehr viel Grafikeinheiten oder viel RAM bei geringer CPU Auslastung. Für die hier angestrebte Infrastruktur reichen die vorgefertigten Lösungen aus.

2.4.2 Hetzner Cloud

Bei der Kommunikation mit der Hetzner Cloud wurde ausschließlich `curl` verwendet, um in dieser Arbeit auch darstellen zu können, wie die Erstellung von Ressourcen unter Verwendung einer REST-API erfolgt. Probleme mit einem freien Werkzeug sämtliche Ressourcen in der Cloud zu verwalten. Hetzner bildet somit eine Ausnahme zu fast allen anderen Anbietern.
Es ist folglich nicht nötig, eine CLI zu installieren und sich über den Browser zu authentifizieren. Hierfür wird ein vorab erstellter Token in Form eines Schlüssels genutzt. Dies hat zur Folge, dass zum Erstellen, Löschen und Hinzufügen von Ressourcen einfache Bash-Skripte (siehe Listing 2.8), wie man sie ab dem ersten Semester an der Hochschule Bremerhaven kennt, genutzt werden können.

```
#!/bin/bash

token="$(cat ../secrets/htz-token)"
networkId=$(curl -s -H "Authorization: Bearer $token"
↪  'https://api.hetzner.cloud/v1/networks' | jq '.networks[] | select(.name
↪  == "location-two") | .id')

printf '{
    "name":"haproxy-two",
    "server_type":"cax11",
    "image":"debian-12",
    "ssh_keys":["ba"],
```

```
    "location":"1",
    "networks":[%s]
}' "$networkId" > payload.json

curl -s -X "POST" "https://api.hetzner.cloud/v1/servers" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $token" \
  -d @payload.json

rm payload.json
```

Listing 2.8: Bereitstellung einer VM bei Hetzner

Nach Aufruf des Skriptes aus Listing 2.9 erfolgt eine sehr umfangreiche als JSON formatierte Antwort von der Hetzner API zurück, in welcher jedes Attribut der verwalteten Server in dem entsprechenden Projekt enthalten ist. Dazu gehören auch detailliertere Informationen, als bspw. mit dem Aufruf der CLI per `hcloud server list`.

```
#!/bin/bash
  token="$(cat htz-token)"
  curl -s \
      -H "Authorization: Bearer $token" \
      'https://api.hetzner.cloud/v1/servers'
```

Listing 2.9: Auflisten der Hetzner Cloud VMs

Insgesamt ist die Verwendung von *Curl* zwar etwas zeitaufwändiger, als lediglich mit der entsprechenden CLI eine VM anzulegen, dafür lässt sich so besser die Erstellung von Ressourcen in verschiedenen Cloud-Umgebungen automatisieren. In diesem Fall ließ sich außerdem erkennen, dass *Curl* das Tool ist, das für mehr Transparenz sorgt. Viele Attribute von zu verwaltenden Ressourcen sind so schneller sichtbar.

2.5 Konfiguration der VMs

Im Laufe des Studiums hat sich in diversen Modulen eine gewisse Standardkonfiguration für die initiale Einstellung von virtuellen Maschinen entwickelt, die auch hier angewendet wurde. Gerade bei der Erstellung von öffentlich erreichbaren VM-Instanzen, also bspw.

virtuellen Maschinen, die direkt über eine IP-Adresse aus dem öffentlichen Netzbereich verfügen, muss besonders auf die Absicherung von Zugängen geachtet werden. Die Konfiguration der Maschine orientiert sich folglich nicht nur an Leistungskriterien und einer gewissen Flexibilität bei der automatisierten Bereitstellung. Vor allem auf die Absicherung der Infrastruktur muss hier besonders geachtet werden. In diesem Kapitel wird diese aufgebaute Standardkonfiguration vorgestellt. Sie betrifft alle VMs die im Rahmen dieser Infrastruktur erstellt werden, da all diese öffentlich erreichbar sind.

2.5.1 Erstellung der virtuellen Maschinen

Für die Virtuelle Maschinen, auf denen die *HAProxys*, die Datenbanken und Testanwendung konfiguriert werden, wurde sich für den Typ `e2-medium` bei Google bzw. `cax11` bei Hetzner entschieden. Die Maschinen verfügen jeweils über zwei Shared CPUs und vier Gigabyte RAM.
Als Standorte dienen London für Location-1 bei Google und Falkenstein für Location-2 bei Hetzner. Als Betriebssystem kommt jeweils *Debian* 12 im stable Release zum Einsatz. Es erhalten alle VMs eine öffentliche und private IP. Ebenfalls befinden sich alle Server an einem Standort in einem gemeinsamen privaten Netz.
Die Server sind dabei jeweils nach ihrer Funktion (app, db, haproxy) und der Location (one, two) benannt. Nach der Erstellung mittels Skript (siehe Listing 2.6 und 2.8) sind die Maschinen über SSH erreichbar, ein SSH-Key wurde bei der Erstellung übergeben.
Der Prozess der Erstellung dauert dabei im Durchschnitt weniger als eine Minute, das Installieren der Standardkonfiguration jeweils wenige Sekunden.

Tabelle 2.1: Auflistung aller VMs mit IP

Hostname	**öffentliche IP**	**private IP**
haproxy-one	35.246.85.138	10.154.0.5
app-one	35.242.191.59	10.154.0.6
db-one	34.89.5.87	10.154.0.7
haproxy-two	188.34.181.227	10.0.1.1
app-two	49.13.74.40	10.0.1.2
db-two	162.55.164.10	10.0.1.3

2.5.2 Konfiguration von Nftables

Grundsätzlich wird weder in der Google-Cloud, noch bei Hetzner die Firewall des jeweiligen Cloud-Providers verwendet. Um auch hier plattformübergreifend eine einheitliche und damit automatisierbare Konfigurationsumgebung zu schaffen, wird *Nftables* als Firewall verwendet. Da die meisten Cloud-Umgebungen jedoch die Verwendung der hauseigenen Firewall erzwingen, wurde dort jeglicher Incoming-Traffic auf die VM-Instanzen erlaubt, um *Nftables* als einzige Komponente zur Zugriffsverwaltung zu sichern.

Wie im Listing 2.10 zu sehen ist, werden in der Input-Chain lediglich Port 22 für SSH-Verbindungen, sowie das ICMP-Protokoll für `ping` erlaubt. Einkommende Verbinungen, für die es keine spezifische Regel gibt, die diese Zugriffe gestatten, werden per `policy drop` verworfen [22].

```
#!/usr/sbin/nft -f
flush ruleset
table inet filter {
 chain input {
  type filter hook input priority filter; policy drop;
  ct state { established, related } counter accept
   ct state invalid drop
   iifname "lo" counter accept
   ip protocol icmp counter accept
   tcp dport {22} counter accept comment "SSH all"
 }
 chain forward {type filter hook forward priority filter; policy drop;}
 chain output {
  type filter hook output priority filter; policy accept;
  tcp dport 25 counter drop comment "Drop SMTP over port 25"
 }
}
```

Listing 2.10: Basiskonfiguration von Nftables

Die Forward-Chain aus dem Listing 2.10 beinhaltet keine besonderen Regeln. Dort ist nur konfiguriert, dass alle Pakete verworfen werden, da in der Infrastruktur bspw. keine speziell konfigurierte Router-Instanz, welche Traffic über verschiedene Netzwerksegmente hinaus verteilt, haben.

Der ausgehende Datenverkehr wird in dieser Konfiguration hingegen grundsätzlich akzeptiert. Lediglich Traffic über den TCP-Port 25 wird verworfen, da dies oft dazu beitragen kann, dass ein System zum Versenden von Spam-Mails missbraucht wird [22].

2.5.3 Konfiguration von Fail2ban

Bei *Fail2ban* handelt es sich ebenfalls um eine Komponente zur Verbesserung der Sicherheitsvorkehrungen auf den virtuellen Maschinen in dieser Infrastruktur. *Fail2ban* ist eine Open-Source-Software, die insbesondere dafür entwickelt wurde, um unautorisierte Zugriffesversuche auf einen Server, anhand verschiedener Gesichtspunkte zu erkennen und zu blockieren.
Speziell im Zusammenspiel mit SSH, kann *Fail2ban* durch eine Analyse der Logdateien des SSH-Daemons, bspw. anhand der Anzahl von fehlgeschlagenen Anmeldeversuche erkennen, dass ein Client als angreifender Dienst zu betrachten ist. *Fail2ban* wird dann die IP-Addresse für einen konfigurierbaren Zeitraum sperren und somit die Loginversuche blockieren [23].

```
1  # JAILS
2  [sshd]
3  enabled = true
4  mode    = ddos
5  port    = ssh
6  logpath = %(sshd_log)s
7  backend = %(sshd_backend)s
8  bantime  = 10m
9  findtime  = 10m
10 maxretry = 5
```

Listing 2.11: Basiskonfiguration von Fail2ban

Die Konfigurationsdatei `jail.local` für *Fail2ban* ist im Listing 2.11 dargestellt. Dort wird lediglich das Jail für den SSH-Daemon aktiviert und mit gewissen Parametern versehen. Spannend ist vor allem die Konfiguration der `bantime` (Anzahl der Sekunden, wie lange ein Host gebannt wird), der `maxretry`-Wert (maximale Anzahl der fehlgeschlagenen Login-Versuche), sowie die `findtime` (Anzahl Minuten in denen *Fail2ban* im Log zurück schaut) [23]. In dieser Infrastruktur werden alle Hosts für zehn Minuten gebannt, die innerhalb von zehn Minuten, fünf fehlgeschlagene Anmeldeversuche aufweisen.

2.6 Aufbau der Netzwerkinfrastruktur

Zur Isolation des Netzwerkverkehrs an einem Cloud-Standort und auch zur Optimierung der Latenzzeiten, wurde an jedem physischen Standort dieser Infrastruktur ein privates Netzwerk bei dem entsprechenden Cloud-Provider eingerichtet, in welches die DB-, App- und *HAProxy*-Instanz hinzugefügt wurden.
Die Erstellung eines solchen Netzwerks funktioniert sehr ähnlich wie die Erstellung von virtuellen Maschinen. Grundsätzlich ist auch ein Netzwerk wieder nur als Ressource zu betrachten, die per REST-API in der entsprechenden Cloud erstellt werden muss.

```
1 gcloud compute networks create location-one \
2     --subnet-mode=auto \
3     --mtu=1460
```

Listing 2.12: Erstellung eines privaten Netzwerks bei Google

Da wir, wie bereits zuvor erwähnt, bei der Interaktion mit der Google-Cloud auf das CLI setzen, ist die Konfiguration des privaten Netzwerkes im Listing 2.12 relativ übersichtlich. Bei der Erstellung müssen lediglich drei Parameter übergeben werden. Der Name des Netzwerks, der `subnet-mode` (automatische Konfiguration der IP über DHCP) und ein Wert für die `mtu` (maximale Übertragungseinheit) [24].

Bei Hetzner (siehe Listing 2.13) wird die Erstellung des privaten Netzwerks wie auch bei der VM-Erstellung per *Curl* umgesetzt. Im Payload werden hier die benötigten Parameter für ein privates Netzwerk definiert.
Neben dem Namen, der auch in diesem Fall angegeben werden muss, wird bei Hetzner noch der Parameter `routes` übergeben, wodurch die Routing Konfiguration genauer definiert wird.

```
#!/bin/bash
token="$(cat ../secrets/htz-token)"

printf '{
"expose_routes_to_vswitch":false,
"ip_range":"10.0.0.0/16",
"labels":{"labelkey":"value"},
"name":"location-two",
"routes":[{"destination":"10.100.1.0/24","gateway":"10.0.1.1"}],
"subnets":[{"ip_range":"10.0.1.0/24","network_zone":"eu-central","type":"cloud
↪  ","vswitch_id":1000}]
}' > payload.json

curl \
  -X POST \
  -H "Authorization: Bearer $token" \
  -H "Content-Type: application/json" \
  -d @payload.json \
  'https://api.hetzner.cloud/v1/networks'
rm payload.json
```

Listing 2.13: Erstellung eines privaten Netzwerks bei Hetzner

Mit dem Argument `ip_range` wird lediglich bestimmt, welches Netzwerk aus dem privaten IPv4-Bereich unseren VM-Instanzen zur Verfügung gestellt werden soll [25]. In diesem Fall wurde dem Netzwerk der Namen `location-two` gegeben und das Subnet 10.0.1.0/24 konfiguriert.

2.7 Integration eines HAProxy

Die erfolgreiche Verwaltung einer Domain erfordert nicht nur eine sorgfältige Konfiguration der DNS-Zone, sondern auch die Implementierung eines Load-Balancers, der in der Lage dazu ist, den HTTP-Traffic, je nach Verfügbarkeit, an den passenden App-Server weiterzuleiten (siehe Abbildung 2.2). Eine bewährte Lösung für diese Aufgabe ist *HAProxy*, ein Open-Source Load-Balancer, der sich insbesondere durch seine Flexibilität und

Skalierbarkeit auszeichnet. Im Folgenden soll beschrieben werden, für welche Anwendungsbereiche *HAProxy* in der Infrastruktur genau genutzt wird und wie dies konkret realisiert wird.

2.7.1 Anwendungsbereich in der Infrastruktur

Beim Einsatz solcher Anwendungskomponenten geht es generell darum, die gesamte Request Verarbeitungskapazität durch eine skalierbare Weise zu erhöhen. Mit der speziellen Implementierung von *HAproxy*, welcher, im Vergleich zu *Apache2* [26], für bestehende Verbindung nicht durchgehend mindestens einen Thread laufen lässt, sondern lediglich auf Callbacks des Clients reagiert [27], entsteht insgesamt eine viel performantere Struktur zur Beantwortung von Requests.

Darüber hinaus kann sich durch das Hinzuschalten weiterer Clients sehr schnell die Last auf weitere Backend-Server einer Anwendung verteilen und zwar ohne, dass der Client in Aktion treten muss [28].
In der in Abbildung 2.2 skizzierten Infrastruktur kommt bei jedem Provider je ein *HAProxy* zum Einsatz. Dies ist im Sinne der Ausfallsicherheit nötig, da ein einzelner *HAProxy* sofort als Single Point of Failure gesehen werden muss. Daraus ergibt sich zusätzlich der Vorteil, dass im Falle eines Ausfalles des App-Server in einer Location, der *HAProxy* auf die andere Location verweisen kann, ohne dass es nötig wird mittels DNS auf eine andere IP-Adresse zu verweisen.

In der angestrebten Architektur werden vorerst keine Loadbalancer Funktionen genutzt, um eine erhöhte Komplexität zu vermeiden. Zur Optimierung von Performance-Kennzahlen, wie bspw. möglicher Requests pro Sekunde, könnte das Loadbalancing von *HAProxy* aktiviert werden.
Somit kommt *HAProxy* lediglich als Reverse Proxy zum Einsatz, um sicherzustellen, dass unsere Backends verfügbar sind, um bei einem Ausfall ggf. umzuschwenken.

Im Gegensatz zu einem Forward-Proxy, arbeitet ein Reverse-Proxy, indem er die Identität des Webservers verbirgt. Ein Client kann also mit einem Reverse-Proxy kommunizieren, ohne direkt über Informationen des eigentlichen Webservers hinter dem Reverse-Proxy zu verfügen. Das hat bspw. den Vorteil, dass ein Reverse-Proxy eine gute Komponente zur Organisation von Load-Balancing-Mechanismen ist.

Jedoch sind auch einige Aspekte hinsichtlich der Performance hier von Bedeutung. Ein Reverse-Proxy kann durch das Zwischenspeichern statischer Inhalte Ladezeiten einer Webanwendung deutlich optimieren. Gleichzeitig ist er auch eine Schlüsselkomponente, um die Ausfallsicherheit einer Anwendung zu optimieren, da ein Client nicht zwangsläufig wissen muss, welcher Server vom Reverse-Proxy konkret angesprochen wird, kann die

Konfiguration des Reverse-Proxy mehrere Backend-Server beinhaltet, womit ggf. ein Single Point of Failure vermieden würde [29].

2.7.2 Konfiguration von HAProxy

Um den *HAProxy* entsprechend der zuvor genannten Anforderungen an einen Proxy in dieser Infrastruktur zu konfigurieren, muss die Standardkonfiguration, die *HAProxy* mitliefert, nicht allzusehr verändert werden.

```
1  global
2    ...
3    frontend https-in
4      bind *:443 ssl crt /etc/letsencrypt/live/certs/
5      acl letsencrypt-acl path_beg /.well-known/acme-challenge/
6      use_backend letsencrypt-backend if letsencrypt-acl
7      default_backend apache-backend
8
9    backend letsencrypt-backend
10   server letsencrypt 127.0.0.1:60001 no-check
11
12   backend apache-backend
13   mode http
14     option httpchk GET /testdb.php
15        http-check expect status 200
16        http-check expect string dbworking
17       server apache1 10.154.0.6:80 check inter 1000 rise 1 fall 1
18       server apache2 49.13.74.40:80 check inter 1000 rise 1 fall 2 backup
```

Listing 2.14: HAProxy Konfiguration

Zu Beginn wird das Frontend `http-in` definiert. Hier wird lediglich eingestellt, dass *HAProxy* auf jeder verfügbaren Netzwerkschnittstelle auf dem Port 80 lauscht und somit für Verbindungen von Clients zur Verfügung steht. Des Weiteren wird hier angegeben, dass alle Requests von Clients an das Default-Backend `apache-backend` weitergeleitet werden. Das Default-Backend wird im weiteren Verlauf des Listings 2.14 konfiguriert.
Zentral ist vor allem der Parameter `option httpchk`. Damit wird eine Option aktiviert, durch welche *HAProxy* regelmäßig Health-Checks nach dem HTTP-Protokoll an die

konfigurierten Backend-Server schickt, um zu überprüfen, ob diese noch so funktionieren wie erwartet.

An dieser Stelle lassen sich auch bestimmte Strings in der Respsonse eines Backend-Servers einstellen, die von einem funktionierenden Backend-Server erwartet werden.
Als erster Backend-Server ist `apache1` mit der IP-Adresse 10.154.0.6 und dem Anwendungsport 80 konfiguriert. Mit dem Parameter `check inter 1000` erreicht man, dass *HAProxy* alle 1000 Millisekunden überprüfen wird, ob dieses Backend noch healthy ist, d.h. für Requests von Clients zur Verfügung steht. Der Standardwert hierfür beträgt zwei Sekunden. Der `rise` Parameter beschreibt, wie viele erfolgreiche Requests *HAProxy* erwartet, bis der Backend-Server wieder als healthy betrachtet werden darf. In der Konfiguration für dieses Projekt wird der Wert auf eins gesetzt.
Der `fall` Parameter beschreibt schließlich, wie viele fehlgeschlagene Requests dazu führen, dass *HAProxy* ein Backend als unhealthy betrachtet. Auch dieser Wert ist in unserere Konfiguration auf eins gesetzt worden [30].

Um zu überprüfen, ob die Datenbankserver nach wie vor erreichbar sind, wird auf den App-Servern ein *PHP*-Skript erstellt, welches lediglich eine Datenbankverbindung öffnet bzw. dies versucht. Sollte die Verbindung nicht erfolgreich zustande gekommen sein, liefert das Skript den String `error` zurück, in dessen Folge der *HAProxy* alle Verbindung an das Failover-System weiterleitet.

Um die Komplexität des Traffics möglichst gering zu halten, wird an dieser Stelle keine Lastverteilung des Traffics, bspw. per Roundrobin-Verfahren umgesetzt. Da der Backend-Server `apache2` mit der Flag `backup` markiert wird, wird der gesamte Traffic zunächst an den ersten Backend-Server weitergeleitet. Nur wenn die zuvor erläuterten Health-Checks an diesen Server fehlschlagen, wird der Traffic umgeleitet [31].

2.8 Aufbau der Watcher-Infrastruktur

Wie bereits in Abbildung 2.2 dargestellt, benötigt die Verwendung von DNS als Failover-Mechanismus eine Watcher-Instanz, welche das System überwacht und den A-Record bei den Providern ändern kann.
Gleichzeitig muss sich die Frage gestellt werden, wo dieser Watcher implementiert wird, da sichergestellt sein muss, dass dieses Programm mit hoher Ausfallsicherheit läuft. Um für dieses Problem eine angemessene Lösung zu finden, musste die Entscheidung getroffen werden, welchen Grad der Automatisierung bzw. welche Aufgaben der Watcher übernehmen soll. Kann der Watcher sowohl von dem als primär genutzten System zum sekundären System schwenken, als auch andersherum, dann muss der Watcher in einer vollkommen autarken Infrastruktur laufen.

Wird die Funktionalität auf einen Wechsel von primärem zu sekundärem System beschränkt, dann kann der Watcher in der Infrastruktur des sekundären Systems implementiert werden, da wenn dieses Ausfällt, die Funktionalität des Watchers nicht mehr benötigt wird.

Wie aus Abbildung 2.2 zu entnehmen, wurde sich für die letztere Variante entschieden. Zum einen, um nicht noch eine dritte Infrastruktur bereitstellen zu müssen und zum anderen, weil unseres Erachtens nach, nach einem Failover kein vollautomatisierter Mechanismus in die Infrastruktur mehr eingreifen darf. Im schlimmsten Fall wird sonst auf eine defekte Infrastruktur zurückgeschwenkt.

Um den Watcher zu überwachen, nutzen wir das primäre System, auf welchem ein Programm läuft, welches lediglich überprüft, ob die Infrastruktur und der Watcher noch läuft und sonst eine entsprechende Meldung erzeugt.

```
#!/bin/bash

ip_primary=35.246.85.138
ip_secondary=188.34.181.227
while true; do
  if ! ping -c19 "$ip_primary" > /dev/null; then
    echo "System nicht erreichbar"
    ./change_arecord_gcloud.sh $ip_secondary; echo "Changed IP by Google"
    ./change_arecord_hetzner.sh $ip_secondary; echo "Changed IP by Hetzner"
    echo "send alert to IT"
    sleep 10 ; ./test_dns.sh > testresult.txt
    echo "tested dns entrys, look at testresult.txt"
    exit 0
  else
    echo "System erreichbar"
  fi
done
```

Listing 2.15: DNS-Watcher

Wie im obigen Listing dargestellt, müssen die statischen IP-Adressen des primären und sekundären Systems fest im Programm hinterlegt werden. Daraufhin wird in einer Endlosschleife geprüft, ob die primäre IP-Adresse auf einen Ping reagiert und somit erreichbar ist. Ist das Programm nicht erreichbar, wird eine Änderung der DNS-Einträge angestoßen (siehe digitalen Anhang) und nach einer zehnsekündigen Wartezeit, eine Überprüfung der

DNS-Einträge bei verschiedenen Resolvern vorgenommen. Im Anschluss beendet sich das Skript, es ist also genau einmal dazu in der Lage, die DNS-Einträge zu ändern.

Die Überprüfung der Erreichbarkeit des primären Systems erfolgt dabei mit 20 Pings über 20 Sekunden (ein Ping je Sekunde), wobei lediglich ein einziger Ping erfolgreich beantwortet werden muss. Dies dient als Absicherung, da es durchaus möglich ist, dass einzelne Pings nicht beantwortet werden, obwohl das System erreichbar ist.
Die Anzahl von 20 Pings ist dabei zu einem gewissen Grad willkürlich gewählt. Ebenso könnte mehr als ein Ping je Sekunde durchgeführt werden. Im Zweifel würde dies die Ausfallzeit minimieren. Da eine Anpassung jederzeit möglich ist, wird dieser Wert bis auf Weiteres beibehalten.
Die im Listing 2.15 gezeigte Ausgabe der Status über einzelne `echo` Kommandos würde in der finalen Infrastruktur über sauberes Logging und die Benachrichtigung der IT über ein Kommunikationsmedium wie Matrix oder E-Mail erfolgen, für die dargestellte Logik und Funktionalität spielt dies hier keine Rolle.

```
#!/bin/bash

while true; do
  if ping -c20 188.34.181.227 > /dev/null; then
    result=$(ssh haproxy-two 'ps aux | grep watcher.sh | grep -v grep')
    if test "$result" = ""; then
      echo "no watcher running"
      echo "alert IT"
      exit 0
    else
      echo "watcher running"
    fi
  else
    echo "secundary system semse offline"
    echo "alert IT"
    exit 0
  fi
done
```

Listing 2.16: Health-Check

Um zu überprüfen, ob der Watcher auf dem sekundären System durchgehend läuft, dient ein Programm auf dem primären System.

Dieses nutzt SSH, um auf dem sekundären System nach einer laufenden Instanz des Skriptes zu suchen und im Falle eines negativen Ergebnisses die IT zu informieren.
In der gewählten Implementierung des DNS-Watcher darf der Health-Check des primären Systems auf keinen Fall einen Neustart des Watchers triggern, da dies sonst zu einer mehrfachen Änderung der DNS-Einträge und immer wieder Neustarten des Skriptes führen würde.

Wie in Listing 2.16 zu sehen, folgt das Health-Check Skript der Logik des DNS-Watchers. Es prüft, ob die sekundäre Infrastruktur überhaupt noch erreichbar ist, informiert ggf. die IT oder sucht nach einer laufenden Watcher Instanz.
Auch in diesem Fall würde die tatsächliche Umsetzung in der Infrastruktur durch sauberes Logging und eine Nachricht an die IT via E-Mail oder ähnliches erfolgen.

2.9 Einrichtung der Anwendungsserver

Innerhalb der Infrastruktur finden sich in beiden Locations je eine virtuelle Maschine, auf welcher die Anwendung gehostet wird. Dafür kommt wie bei allen Servern derselbe Maschinentyp zum Einsatz. Innerhalb der jeweiligen Location wird die VM zum internen Netz hinzugefügt. Im Folgenden soll kurz beschrieben werden, welche Besonderheiten zur Inbetriebnahme der Anwendung nötig waren.

2.9.1 Vorbereitung der VMs für die Anwendung

Für die Nutzung als Anwendungsserver werden einige Pakete benötigt, die mittels `apt` installiert werden können. Die benötigten Pakete sind `apache2`, `mariadb-client`, `php` und `php-mysql`.
Durch die Basiskonfiguration sind bereits *Nftables* und *Fail2ban* installiert. Die Konfiguration der `nftables.conf` muss jedoch noch so angepasst werden, dass der Traffic nur von den jeweiligen *HAProxy*-Servern erlaubt wird.

Durch die Einstellung aus Listing 2.17 wird sichergestellt, dass nur die *HAProxy*-Server beider Locations den Port 80 und 443 erreichen können. Auf dem Anwendungsserver der Location-2 werden die entsprechend passenden IP-Adressen hinterlegt.

```
#!/usr/sbin/nft -f

flush ruleset
table inet filter {
  set allowed_ips {
    type ipv4_addr
    elements = {35.246.85.138,10.154.0.5,188.34.181.227}
  }
  chain input {
    type filter hook input priority filter; policy drop;
  ...
    ip saddr @allowed_ips tcp dport {80, 442} counter accept
  ...
```

Listing 2.17: nftables.conf des App-Servers

Da als Standardeinstellung `/var/www/html` dem User `root` gehört, wird diese Einstellung wie in Listing 2.18 überschrieben und das Verzeichnis dem User `ba` und der Gruppe `www-data` zugeordnet. Darüber hinaus wird das Verzeichnis so eingestellt, dass alle neuen Dateien immer der Gruppe `www-data` zugeordnet werden. Diese Einstellung ist der aus der Informatikinfrastruktur nachempfunden.

```
sudo chown -R ba:www-data /var/www/html/
sudo chmod g+s /var/www/html/
```

Listing 2.18: Ändern der User- und Gruppenzuweisung von html

Durch die Installation der übrigen Pakete ist es nun möglich *PHP*-Skripte auszuführen und mit einer *MariaDB* Datenbank zu kommunizieren. Eine Einrichtung bzw. Konfiguration auf den Anwendungsservern ist nicht nötig.

2.9.2 Übertragung der Sessions zwischen den Anwendungsservern

Da die gewählte Anwendung Sessions verwendet, muss über die Replikation dieser zwischen den Anwendungsservern betrachtet werden, da sonst Anwender:innen bei einem

Failover über keine gültige Session verfügen und somit aus dem System ausgeloggt werden. Hierfür wurde eine Lösung mittels Bash-Skript (Listing 2.19) angestrebt und umgesetzt. Dabei werden die Session-Dateien mittels `scp` an das Failover-System, den zweiten Anwendungsserver, übertragen.

```
#!/bin/bash

cd /tmp
touch transffered.txt
find /tmp/ -maxdepth 1 -type f -name 'sess_*' | xargs basename -a 2>/dev/null | sort -u > to_transfer.txt
diff -u to_transfer.txt transffered.txt | grep -E "^-[a-zA-Z]" | sed -E 's/^(-)(.*)/\2/g' > transfer.tmp

sleep 1

if  test -s transfer.tmp; then

  sudo scp $(cat transfer.tmp | tr "\n" " ") app-two:tmp/
  sudo ssh app-two "sudo chown www-data:www-data tmp/ && sudo -u www-data mv tmp/* /tmp/"
  cat transfer.tmp >> transffered.txt
  sort -u transffered.txt -o transffered.txt

fi
```

Listing 2.19: Tranfser der Sessions zwischen den Anwendungsservern

So wird ermittelt, welche Session Dateien existieren und gegen eine Liste von Dateinamen abgeglichen, welche bereits übertragen wurde. Dies dient zur Minimierung der Übertragungszeit, da nicht jedes Mal alle Dateien kopiert werden müssen.
Zwischen Ermittlung der zu übertragenden Sessions und dem eigentlichen Kopiervorgang wird eine Sekunde gewartet. Dies erhöht zwar die Wahrscheinlichkeit im Ausfall Sessions nicht rechtzeitig zu übertragen, sichert aus unserer Sicht jedoch, dass die Sessions vollständig von *PHP* geschrieben wurden.
Da die Dateien normalerweise erst gelöscht werden, wenn der Server heruntergefahren wird, existiert ein weiteres Skript, welches abgelaufene Sessions automatisch löscht.

```
#!/bin/bash

verzeichnis="/tmp"

find $verzeichnis -maxdepth 1 -type f -name 'sess_*' -mmin +31 | xargs rm
↪  2>/dev/null
```

Listing 2.20: Löschen alter Sessions

Beide Skripte kommen kombiniert in einem Skript zum Einsatz, welches durchgehend ausgeführt wird und die Replikation und den Löschvorgang sicherstellt. Zu beachten ist, dass die Skripte von `www-data` ausgeführt werden müssen, da die Dateien diesem User gehören. Hierbei muss der Handlungsspielraum des Users auf dem Zielserver beschränkt werden, da dies sonst ein Sicherheitsrisiko darstellt.

2.10 Aufbau des Datenbankmanagementsystems

Die Theorie besagt, dass die einzelnen Komponenten unserer Architektur in völliger Unabhängigkeit voneinander entwickelt werden könnten. Dennoch ist es in den meisten Fällen so, dass Anwendungen, selbst wenn sie parallel und an verschiedenen Standorten betrieben werden, auf dieselbe Datenbank zugreifen. Die Robustheit und Zuverlässigkeit unserer gesamten Infrastruktur hängt daher in hohem Maße von der Verfügbarkeit des Datenbanksystems ab.

Eine Datenbank muss stets dem Anspruch gerecht werden, allen Benutzer:innen gleichzeitig dieselben Daten zur Verfügung stellen zu können [32]. Um sicherzustellen, dass das gebaute Datenbanksystem ständig erreichbar ist, wurden in der Infrastruktur zwei Datenbank-Server aufgebaut.
Diese Server arbeiten mithilfe der klassischen Binärlog-Replikation, um ihre Transaktionen synchron zu halten und die Daten standortübergreifend zur Verfügung zu stellen. Bei dem Datenbankmanagementsystem welches hier aufgebaut werden soll, wird auf *MariaDB* gesetzt, da dieses frei und Open-Source ist und somit die naheliegende Wahl für den Aufbau einer cloudbasierten Infrastruktur darstellt.

2.10.1 Grundlagen einer Datenbankreplikation

Bevor der praktische Aufbau des Datenbankmanagementsystems mit verschlüsselter Replikation erläutert wird, ist es zunächst wichtig, die theoretischen Grundlagen der Replikation zu verstehen.
Insgesamt ist Replikation als eine Funktion zu betrachten, welche es ermöglicht, die Inhalte von einem oder mehreren Servern (Primaries), auf andere *MariaDB*-Server (Replicas) zu spiegeln [33]. *MariaDB* bietet verschiedene Möglichkeiten, bzw. Mechanismen eine Replikation technisch umzusetzen. Die wohl bekannteste basiert auf dem sogenannten Binärprotokoll.
Durch ein aktiviertes Logging der Datenbanktransaktionen im Binärformat von *MariaDB* werden alle Aktualisierungen der Datenbank, also bspw. Datenmanipulationen, als Ereignis in das Binärlog geschrieben. Die Replicas lesen das Binärprotokoll von jedem Primary, um auf die zu replizierenden Daten zuzugreifen.
Auf dem Replikat wird schließlich ein Relay-Log erstellt, das auf demselben Format wie das Binärlog basiert und zur Durchführung der Replikation verwendet wird [33].

Ein Replikationsserver verfolgt die Position des letzten auf dem Replica angewendeten Ereignisses im Binärprotokoll des Primarysystems. Dadurch kann der Replikationsserver nach einer vorübergehenden Unterbrechung der Replikation die Verbindung wiederherstellen und an der Stelle fortsetzen, an der aufgehört wurde. Ein weiterer Vorteil an einem Replikationssetup ist somit, dass ein Replica durchaus, bspw. vor Wartungsarbeiten auf der VM geklont und ggf. durch das neue Replikat ersetzt werden kann. Die Replikation wird durch das persistierte Binärlog dann sofort wieder aufgenommen [33]. Primaries und Replicas müssen folglich nicht ständig miteinander kommunizieren.
Das besondere an der Konfiguration in dieser Infrastruktur ist, dass keine klassische Primary-Secondary Replikation gebaut wird, sondern eine Primary-Primary Replikation.

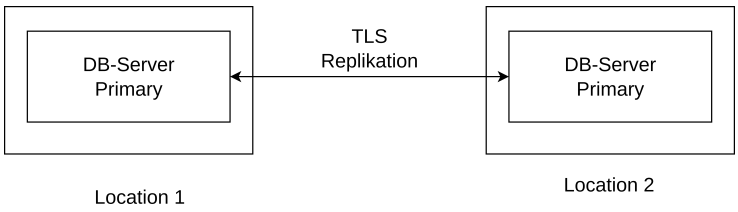

Abbildung 2.3: Datenbankreplikation zwischen zwei Primary-Nodes

Damit wird erreicht, dass an beiden Standorten der Infrastruktur eine beschreibbare Datenbank existiert. Würden in diesem Fall eine Primary-Replica Replikation aufgebaut werden, hätte der Standort mit der Replica-Node nur eine Read-Only Datenbank und müsste für Transaktionen eine Verbindung mit der Datenbank am anderen Standort aufbauen.

In diesem Fall sind also beide *MariaDB*-Server sowohl als Primary, als auch Replica zu betrachten.
Beide erstellen Binär- und Relay-Logs, um sowohl eigene Transaktionen zu sichern, als auch die Transaktionen der jeweils anderen Datenbank über das Relay-Log einzuspielen.

Beim Aufbau einer Datenbankreplikation existieren, vor allem durch die hohe Komplexität, diverse Herausforderungen, die hier vor der konkreten Konfiguration zunächst dargestellt werden sollen [33]:

- **Datenkonsistenz:** Die Wahrung der Datenkonsistenz zwischen den beiden Primary-Nodes ist von entscheidener Bedeutung für ein verteiltes Datenbankmanagementsystem. Es darf zu keinem Zeitpunkt eine Situation entstehen, in der beide *MariaDB*-Nodes inkonsistente Ergebnisse liefern.
- **Konfliktauflösung:** Wenn auf beiden Primary-Nodes gleichzeitig Änderungen an denselben Daten vorgenommen werden, können Konflikte auftreten. In der Infrastruktur muss also durch geeignete Maßnahmen sichergestellt werden, dass es, bspw. durch eine angepasste Konfiguration in den Datenbankzugriffen der Testanwendung nicht zu solchen Konflikten kommen kann.
- **Latenz:** Eine Replikation hat eine gewisse Latenz. Je nach Netzwerkgeschwindigkeit und dem Setup der ausgewählten Cloud-Provider kann dies zu Verzögerungen führen. Der Aufbau eines sinnvollen Monitorings der Latenz ist allerdings ein Thema für sich und stellt aus unserer Sicht eine Herausforderung dar.
- **Überwachung und Wartung:** Die laufende Überwachung der Replikation durch Skripte ist unerlässlich um sicherzustellen, dass die Replikation ordnungsgemäß funktioniert. In diesem Setup müssen Fehler schnell erkannt werden um Datenverlust, oder ggf. einen Verlust der Datenintegrität zu verhindern.
- **Sicherheit:** Die Replikation zwischen den Primary-Nodes kann ein Sicherheitsrisiko darstellen, da sensible Daten in der Standardkonfiguration unverschlüsselt über das Netzwerk übertragen werden. Um die weitere Komplexität eines VPN-Tunnels zu vermeiden, haben wir daher die Replikation per TLS verschlüsselt.

Trotz der vielen Herausforderungen lohnt sich aus unserer Sicht der Aufbau einer Replikation im Zusammenhang einer Multi-Cloud-Infrastruktur. Kritische Informationen an mehr als einer Stelle aufzubewahren ist ein klarer Vorteil, der die Stabilität der gesamten Infrastruktur positiv beeinflusst.

2.10.2 Installation von MariaDB

MariaDB kann auf einem *Debian*-System klassich per `apt` installiert werden, da es in den Standardpaketquellen enthalten ist.
Die Installation von *MariaDB* (siehe Listing 2.21) muss sowohl auf `db-one` und `db-two` erfolgen.

```
1 $~ apt update
2 $~ apt install mariadb-server
3 $~ mysql_secure_installation
```

Listing 2.21: Installtion von MariaDB

Nach der Installation wurde eine `mysql_secure_installation` durchgeführt, wodurch gewisse Sicherheitskonfigurationen der Datenbankserver vorgenommen wurden. Gerade da in dieser Infrastruktur kein VPN-Tunnel existiert, ist eine korrekte Konfiguration der *MariaDB*-Server von großer Bedeutung.

2.10.3 Vorbereitung der TLS-Verschlüsselung

Da, wie bereits erwähnt, die Replikationsdaten zwischen den zwei *MariaDB*-Servern per TLS verschlüsselt werden sollen, müssen hier zunächst ein paar Vorbereitungen getroffen werden. Konkret wird per `openssl` eine CA erstellt, welche in der Lage dazu ist, die Zertifikate der zwei *MariaDB*-Server zu verifizieren.
Da in dieser Infrastruktur eine Primary-Primary Replikation aufgebaut wird, muss auch jeder Server über ein eigenes X509-Zertifikat und einen Private-Key verfügen (Two-Way TLS) [34].

```
1 mkdir -p /etc/mysql/ssl/
2 chown -Rvv mysql:root /etc/mysql/ssl/
3 openssl genrsa 2048 > ca-key.pem
4 openssl req -new -x509 -nodes -days 730 -key ca-key.pem -out ca-cert.pem
```

Listing 2.22: Erstellung der CA mit openssl

Im Listing 2.22 wird dargestellt, wie zunächst auf beiden *MariaDB*-Servern ein Verzeichnis für die von `openssl` generierten Ausgabedateien angelegt wird. Schließlich wird

basierend auf dem erstellten Schlüssel `ca-key.pem` das Zertifikat `ca-cert.pem` erstellt. Das Zertifikat hat eine Gültigkeit von 730 Tagen und ist vom Typ X509 [35].

```
1 openssl req -newkey rsa:2048 -days 730 -nodes -keyout server-key.pem -out
  ↪ server-req.pem
2 openssl rsa -in server-key.pem -out server-key.pem
3 openssl x509 -req -in server-req.pem -days 730 -CA ca-cert.pem -CAkey
  ↪ ca-key.pem -set_serial 01 -out server-cert.pem
```

Listing 2.23: Erstellung des db-one Zertifikats

Mit dem erstellten X509-Zertifikat `ca-cert.pem` können nun, wie in den Listings 2.23 und 2.24 dargestellt, Zertifikate erstellt werden, die von der im ersten Schritt erstellten CA signiert wurden. Die Zertifikate sind auch hier wieder vom Typ X509 und 730 Tage gültig [35]. Somit können die *MariaDB-Server* untereinander die Daten entschlüsseln. Zusätzlich ist hiermit die Datenintegrität sichergestellt, da die Transaktionen nachweisbar von der spezifizierten Node stammt.

```
1 openssl req -newkey rsa:2048 -days 730 -nodes -keyout server-two-key.pem -out
  ↪ server-two-req.pem
2 openssl rsa -in server-two-key.pem -out server-two-key.pem
3 openssl x509 -req -in server-two-req.pem -days 730 -CA ca-cert.pem -CAkey
  ↪ ca-key.pem -set_serial 01 -out server-two-cert.pem
```

Listing 2.24: Erstellung des db-two Zertifikats

Per `openssl` lassen sich die erstellten Zertifikate für `db-one` und `db-two` verifizieren (siehe Listing 2.25). Somit ist sichergestellt, dass die erstellten Zertifikate für die Replikation von der selbst signierten CA erstellt wurden [35].

```
1 root@db-one:/etc/mysql/ssl# sudo openssl verify -CAfile ca-cert.pem
  ↪ server-cert.pem server-two-cert.pem
2 server-cert.pem: OK
3 server-two-cert.pem: OK
```

Listing 2.25: Testen der Zertifikate

2.10.4 Einrichtung der DB-Replikation mit TLS

Auch für die eigentliche Konfiguration der Datenbankreplikation in *MariaDB* müssen einige Konfigurationen vorgenommen werden. Zu Beginn muss für eine funktionierende Replikation ein User erstellt, sowie mit passenden Rechten ausgestattet werden. Wie im Listing 2.26 dargestellt, wird dies mit einem klassischen *SQL*-Create-User Statement umgesetzt.

Schließlich werden dem erstellten User `replicator` noch die Rechte `grant replication slave` zugeordnet. An dieser Stelle wird mit `REQUIRE SSL` sichergestellt, dass es keine funktionierende Replikation über diesen User geben kann, die nicht per TLS abgesichert ist [33].

```
1 CREATE USER 'replicator'@'%' IDENTIFIED BY 'feuerzeug';
2 GRANT REPLICATION SLAVE ON *.* TO 'replicator'@'%' REQUIRE SSL;
3 FLUSH PRIVILEGES;
```

Listing 2.26: Erstellung eines Replication Users

Anschließend wird der *MariaDB*-Server für die Replikation konfiguriert (siehe Listing 2.27). Wichtig ist hier zunächst für alle Server eine individuelle `server-id` zu setzen, sowie die `bind-address` auf `0.0.0.0` anzupassen. Somit ist der *MariaDB*-Daemon auf allen konfigurierten IP-Adressen erreichbar und kann bspw. auch vom App-Server aus erreicht werden.
Der Parameter `max_binlog_size` wird hier auf `100M` gesetzt. Damit werden die Binärlogs nicht größer als 100 Megabyte und andernfalls rotiert.

```
1 bind-address            = 0.0.0.0
2 server-id               = 1
3 log_bin                 = /var/log/mysql/mysql-bin.log
4 max_binlog_size         = 100M
5 relay_log = /var/log/mysql/mysql-relay-bin
6 relay_log_index = /var/log/mysql/mysql-relay-bin.index
7 ssl-ca=/etc/mysql/ssl/ca-cert.pem
8 ssl-cert=/etc/mysql/ssl/server-cert.pem
9 ssl-key=/etc/mysql/ssl/server-key.pem
```

Listing 2.27: 50-server.conf von db-one

Die Konfiguration des Servers `db-two` findet sich im digitalen Anhang.
Bevor nun im finalen Schritt die Replikation auf den zwei *MariaDB*-Servern aktiviert werden kann, muss die aktuelle Position des Binärlogs per `show master status` ermittelt werden. Die Position ist hier ein Verweis auf die letzte im Binärlog persistierte Datenbanktransaktion [33].

```
1 CHANGE MASTER TO MASTER_HOST = '<db-two_ip>' , MASTER_USER =
  ↪ 'replication_user' , MASTER_PASSWORD = 'feuerzeug' , MASTER_LOG_FILE =
  ↪ 'mysql-bin.000001', MASTER_LOG_POS = <position_binary_log_A>, MASTER_SSL =
  ↪ 1;
2 START SLAVE;
```

Listing 2.28: Konfiguration der Replikation auf db-one

Abschließend kann die Replikation per `change master to` angepasst werden. Unter der Angabe der IP-Adresse des jeweils anderen Datenbankservers, dem Binärlog, der aktuellen Position des Binärlogs, an der die Replikation beginnen soll, sowie den Userdaten des Replikationsusers wird durch das `change master to` die Datenbankreplikation hinreichend konfiguriert.
Auch hier wird durch das Argument `MASTER_SSL` TLS bei der Replikation sichergestellt. Das Starten der Replikation erfolgt schließlich per `start slave` (siehe Listing 2.28).

```
1 CHANGE MASTER TO MASTER_HOST = '<db-one_ip>' , MASTER_USER =
  ↪ 'replication_user' , MASTER_PASSWORD = 'feuerzeug' , MASTER_LOG_FILE =
  ↪ 'mysql-bin.000001', MASTER_LOG_POS = <position_binary_log_B>, MASTER_SSL =
  ↪ 1;
2 START_SLAVE;
```

Listing 2.29: Konfiguration der Replikation auf db-two

Im Listing 2.29 wird dasselbe Statement für `db-two` durchgeführt. Hier unterscheiden sich lediglich die IP-Adresse und ggf. die Position des Binärlogs. Nachdem auf beiden Primaries `start slave` ausgeführt wurde, lassen sich per `show slave status` diverse Informationen über die Replikation auslesen.
Um die Funktionalität der Replikation zu überprüfen, sind bspw. `Slave_IO_Running` und

`Slave_SQL_Running` genauer zu betrachten. Sie geben an ob die Replikation auf IO- und *SQL*-Ebene funktioniert [33].

```
# Test db-one
create database test_db;
create table data(id int, name varchar(20));
insert into data values(1,"test");
# Watch auf db-two
watch -n 1 'mysql -e "select * from test_db.data"'
```

Listing 2.30: Testen der Replikation

Am besten lässt sich die Replikation aber mit dem Anlegen einer Datenbank testen (siehe Listing 2.30). Nachdem in einer Test-Tabelle Daten hinzugefügt wurden, sollten diese sofort per `watch` auf dem jeweils anderen *MariaDB*-Server sichtbar sein.
Der oben beispielhaft dargestellte Test verlief zufriedenstellend, so konnten geschriebene Daten in der zweiten Location erfolgreich abgefragt werden.

2.11 Gesamtübersicht der aufgebauten Infrastruktur

Die folgende Abbildung 2.4 zeigt final die in den vorherigen Schritten beschriebenen Komponenten der Multi-Cloud-Infrastruktur. Von der Implementierung über diverse Health-Checks zum Triggern automatisierter Umschwenk-Mechanismen, über die klassiche Kompontente *HAProxy* und Applikationssserver, bis zum Datenbankmanagementsystem mit verschlüsselter Replikation, existiert zumindest in der Theorie kein einziger Single Point of Failure innerhalb der Infrastruktur.

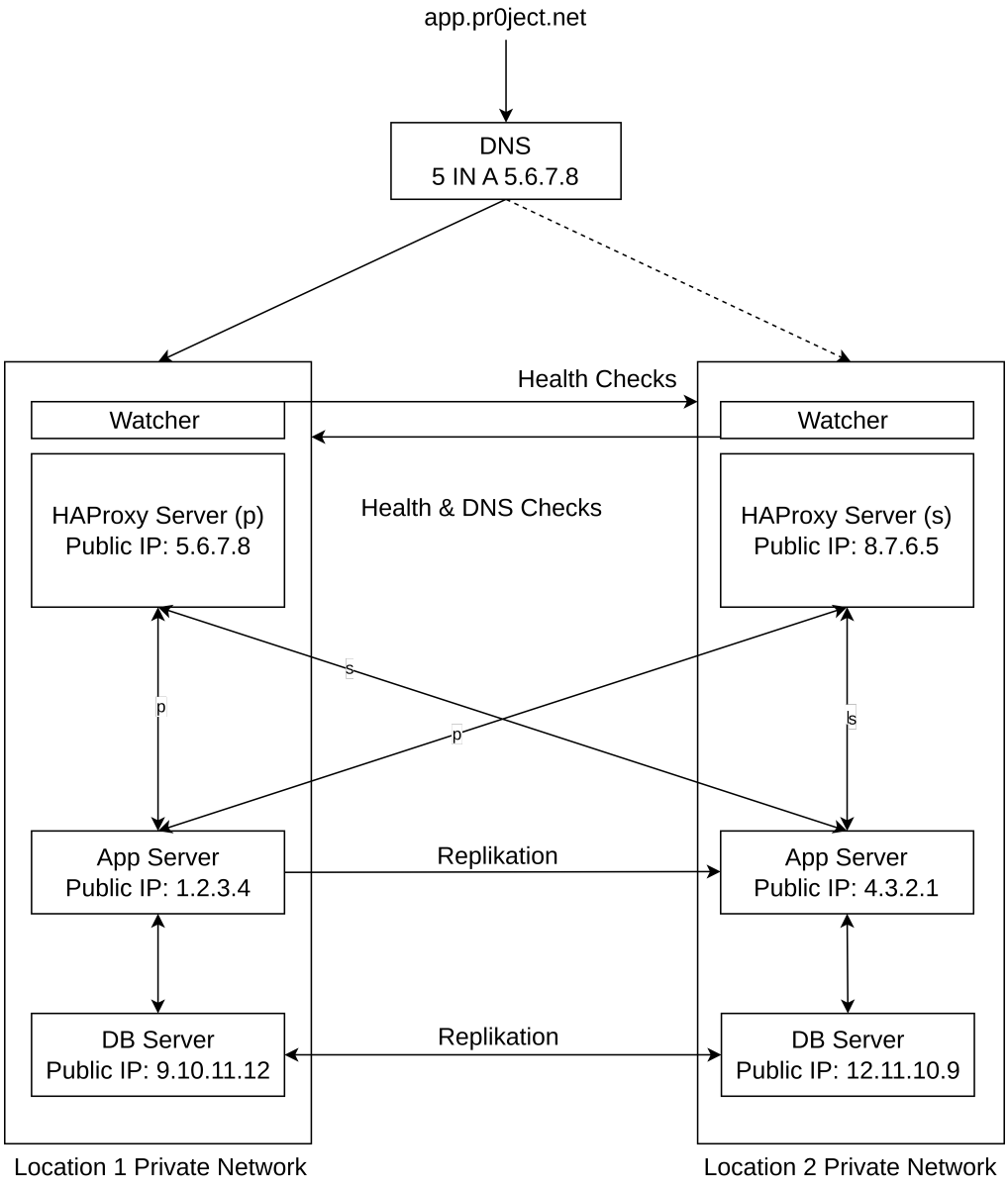

Abbildung 2.4: Gesamtübersicht der aufgebauten Infrastruktur

Dennoch sieht die aufgebaute Architektur, zumindest aus unserer Sicht, in dieser Abbildung nicht über die Maßen kompliziert aus. Obwohl die Komponenten in der Cloud mit vielen komplexen Tools ausgestattet wurden, um die aufgestellten Anforderungen erfüllen zu können, folgt das Architekturdesign einem simplen Prinzip. In dem folgenden Kapiteln soll mit diversen Tests überprüft werden, ob die Infrastruktur wirklich so robust ist, wie geplant oder ob es noch weitere Punkte zu beachten gilt.

3 Entwicklung einer Testanwendung

Um die Infrastruktur sinnvoll und realitätsnah zu testen, braucht es eine Testanwendung, welche die, aus unserer Sicht, grundlegenden Funktionalitäten bereitstellt. Gleichzeitig soll die Anwendung nur als Hilfsmittel dienen, bestimmte Failover-Mechanismen umzusetzen und zu testen. Im Folgenden werden die konkreten Anforderungen an die Anwendung dargestellt, wie diese Implementiert wurden und wie die Anwendung in der Infrastruktur bereitgestellt wird.

3.1 Anforderungen an die Anwendung

Die Anwendung sollte über zwei grundlegende Funktionalitäten verfügen, eine Validierung des Logins über alle Anwendungsebenen hinweg mittels Sessions und die Möglichkeit mit einer Datenbank zu interagieren.
Diese beiden Funktionen sind, aus unserer Sicht, deshalb so relevant, weil sie die größten Herausforderungen für eine verteilte Infrastruktur darstellen. Andere Inhalte, ob nun *HTML*-Seiten, *Java*-Code oder *PHP*-Skripte lassen sich relativ problemlos übertragen, gerade in eine Failover-Infrastruktur, welche, im Normalfall, keine Aufrufe registriert. Datenbankeinträge und Sessions müssen aber in Echtzeit repliziert werden, um das Failover-System sofort zur Verfügung zu stellen. Gleichzeitig sind sie von der Interaktion der Nutzenden mit dem System abhängig und besonders anfällig für Inkonsistenzen, wenn z. B. das System während des Erstellens einer Session oder einer Datenbanktransaktion ausfällt.

Da die Anwendungsebene so simpel wie möglich gehalten werden soll, wurde sich für die Implementierung einer CRUD-Anwendung entschieden. Die Nutzung der CRUD-Schnittstelle erfordert, sowohl im Browser, als auch mittels *Curl* eine vorherige Authentifizierung über Username und Passwort.
Als Programmiersprache dient dabei *PHP*, diese Entscheidung wurde getroffen, da *PHP* direkt hinter einem *Apache* laufen kann und als Programmiersprache an der Hochschule Bremerhaven gelehrt wird. Ebenso verfügt *PHP* über interne Schnittstellen zur Kommunikation mit einem Datenbankmanagementsystem und den Umgang mit Sessions.
Auch wenn das Frontend für die Anwendung eine untergeordnete Rolle spielt, soll die Anwendung als Single-Page-Anwendung fungieren. Damit wird entsprechend die Möglichkeit benötigt, Inhalte, im Speziellen die Datenbankoperationen, dynamisch zu laden. Die Umsetzung erfolgt in diesem Fall mittels *JavaScript* und Ajax-Calls.
Ebenso soll ein Mechanismus implementiert werden, welcher das Backend auf Erreichbarkeit hin überprüft und im Zweifel das Frontend zum Umschwenken ins Failover-System zwingt.

Sowohl auf *PHP* als auch auf *JavaScript* Ebene verzichten wir auf die Verwendung von Frameworks. Die Umsetzung soll auf einem möglichst einfachen und für viele verständlichen Niveau erfolgen.

3.2 Implementierung der Testanwendung

Für die Loginmaske, welche den Einstiegspunkt in unsere Anwendung darstellt, wird ein *PHP*-Skript verwendet (siehe Listing 3.1), welches die eingegebenen Daten zur Validierung weiterleitet. Die Validierung selbst erfolgt über eine Datenbankabfrage des Hash und Salt.

```php
<?php

include ('createDBConnection.php');

function getHashAndSaltByUsername($conn, $username) {
  $stmt = $conn->prepare("SELECT hash, salt FROM login WHERE username = ?");
  $stmt->bind_param("s", $username);
  $stmt->execute();
  $result = $stmt->get_result();

  if ($result->num_rows > 0) {
    $row = $result->fetch_assoc();
    $hash = $row['hash'];
    $salt = $row['salt'];
    return array('hash' => $hash, 'salt' => $salt);
  } else {
    return null; // Benutzername nicht gefunden
  }
  $stmt->close();
}
```

Listing 3.1: Abfrage der Logindaten aus der Datenbank

Die aus der Datenbank abgefragten Werte werden gegen die Eingabe des Users, wie in Listing 3.2 dargestellt, geprüft, dazu wird die Eingabe mit dem Salt aus der Datenbank im selben Verfahren gehashed.

```php
<?php
function checkHashAndSalt($username, $password, $conn) {
  $result = getHashAndSaltByUsername($conn,$username);
  if ($result === null) {
    header('Location: login.php?status=failed');
    exit();
  }
  $hash = $result['hash'];
  $salt = $result['salt'];
  $hashtocheck = hash('sha256', $password . $salt);
  if ($hashtocheck == $hash) {
    return True;
  }
  else {
    return False;
  }
}
```

Listing 3.2: Prüfen der Logindaten

Sollte der User sich erfolgreich authentifiziert haben, wird eine Session erstellt und in diese eine zufällige ID und der Username gespeichert. Daraufhin wird der User an die Anwendung weitergeleitet (siehe Listing 3.3).

```php
<?php
function startRandomSessionAndRedirect($username) {
  session_set_cookie_params(1800);
  $new_path = '/tmp';
  session_save_path($new_path);
  session_start();
  $_SESSION['session_id'] = bin2hex(random_bytes(16));
  $_SESSION['username'] = $username;
  header('Location: main.php');
  exit();
  }
```

Listing 3.3: Erstellen der Session

Die Datenbankverbindung wird dabei in allen Fällen in eine gesonderte Datei, siehe Listing 3.4 ausgegliedert und von den weiteren *PHP*-Skripten eingebunden.

```php
<?php
function openDatabaseConnection() {
  $servername = "10.154.0.7";
    $dbusername = "ba";
    $dbpassword = "feuerzeug";
    $dbname = "bachelorarbeit";
  $conn = new mysqli($servername, $dbusername, $dbpassword, $dbname);

  if ($conn->connect_error) {
    die("Verbindung zur Datenbank fehlgeschlagen: " . $conn->connect_error);
  }
  return $conn;
}
?>
```

Listing 3.4: Öffnen der Datenbankverbindung

Auf der Hauptseite der Anwendung steht die CRUD-Funktionalität über simple *HTML*-Forms bereit. Alle hinter dem Login befindlichen Funktionalitäten durchlaufen vorab eine Validierung über die Session-ID, wie in Listing 3.5 dargestellt.

```php
<?php
if (!isset($_COOKIE["PHPSESSID"])){
  header('Location: login.php?case=0');
  exit();
}
session_start();
if (!isset($_SESSION['session_id']) || empty($_SESSION['session_id'])) {
  header('Location: login.php?case=1');
  exit();
}
```

Listing 3.5: Validerung der Session

Die Datenbankoperationen erfolgen in jeweils separaten Skripten und folgenden alle grundsätzlich demselben Schema, wie bei der Abfrage der Logindaten (siehe Listing 3.1). Die Abfrage an die Datenbank erfolgt dabei stets mittels sog. Prepared Statements, damit soll verhindert werden, dass simple *SQL*-Injektions möglich werden.

Es gilt zu betonen, dass die Umsetzung relativ rudimentär ist. Es werden nicht für alle Eventualitäten Fehler ausgegeben oder entsprechende Rückmeldungen erzeugt. Da jedoch nur die Funktionalitäten gebraucht werden, ist es aus unserer Sicht auch nicht zwingend nötig die Anwendung für alle möglichen Eingaben und Rückmeldungen abzusichern.
Es soll vorerst genügen, die Seiten gegen unbefugte Nutzung und die gängigsten *SQL*-Injektions abzusichern, dies haben wir aus unserer Sicht erreicht.

Für die Nutzung im Browser ist zusätzlicher *JavaScript*-Code implementiert. Es handelt sich hauptsächlich um Ajax-Calls zum Abrufen der Inhalte im Hintergrund und Aktualisieren der Seite. Um im Frontend den Ausfall der Infrastruktur zu bemerken und entsprechend zu reagieren, wird jede Sekunde geprüft, ob das Skript `main.php` noch erreichbar ist. Falls dies nicht der Fall ist, erfolgt eine Meldung. (Listing 3.6).

```
1  var redirectEvent = false;
2  function redirectFunction(){
3    if (!redirectEvent){
4    alert("Es ist etwas schiefgelaufen: Bitte warte 1 Minute");
5    redirectEvent = true;
6    }
7  }
8  function checkServerStatus() {
9    var req = new XMLHttpRequest()
10   function statechange(){
11   console.log("location: one, state:"+req.readyState+"  "+req.status);
12   }
13   req.onreadystatechange = statechange
14   req.onerror = redirectFunction
15     req.open("GET","https://pr0ject.net/main.php")
16     req.send()
17 }
18 setInterval(checkServerStatus,1000);
```

Listing 3.6: Statusabfrage an den Server

Die Anwendung besteht somit aus einer kleinen Zahl *PHP*-Skripte und einer *JavaScript*-Datei. Entsprechend schnell lässt sie sich auf die Anwendungsserver deployen und ausführen.

3.3 Testen der Anwendungsimplementierung

Die Anwendung wurde sowohl mit verschiedenen Browsern (Firefox, Chrome, Safari) als auch auf korrekte Funktionalität geprüft. Hierfür wurden Einträge über die Weboberfläche hinzugefügt und anschließend die Anzeige im Browser und der Datenbankeintrag auf dem Server geprüft. Selbes gilt für das Löschen von Einträgen anhand der ID und dem Ändern von Einträgen.
Diese manuellen Tests haben gezeigt, dass die Implementierung grundsätzlich gut funktioniert, in keinem Fall kam es zu Fehlern. Wie durch die Implementierung zu erwarten, fehlen bestimmte Mechanismen zum Fehlerhandling, so gibt es keine Rückmeldung, wenn versucht wird eine fremde (nicht dem User zugeordnete) ID zu löschen oder einen fremden Inhalt zu bearbeiten. Dies liegt schlicht daran, dass diese Funktionalität nicht implementiert wurde.

Auf die manuellen Tests folgte eine Überprüfung mittels `Curl`, in der alle Funktionen einmal überprüft wurden. Hierfür wurde zuerst der Login vorgenommen und die Session abgespeichert (Listing 3.7).

```
1 curl -c cookie.jar -d "user=demo&password=1234"
  ↪ https://pr0ject.net/checkLoginData.php
```

Listing 3.7: Automatisierter Login und Speichern der Session mit curl

Die gespeicherte Session kann nun genutzt werden, um Einträge anzulegen. In einem ersten Schritt erfolgt ein einzelner Eintrag (Listing 3.8).

```
1 curl -b cookies.txt -d "content=Manueller Eintrag"
  ↪ https://pr0ject.net/addEntry.php
```

Listing 3.8: Automatisiertes Erstellen eines Eintrages mit curl

Dieser Schritt kann natürlich auch automatisiert erfolgen, um so sehr viele Einträge anzulegen und darauffolgend die Einträge zu prüfen (siehe Listing 3.9 und 3.10).

```
for i in {1..100}; do curl -b cookies.txt -d "content=automatischer Eintrag
↪  $i" https://pr0ject.net/addEntry.php; done
```

Listing 3.9: Automatisiertes Erstellen vieler Einträges mit curl

```
curl -b cookies.txt https://pr0ject.net/getAllEntrys.php | sed -E
↪  's/\|\|\|/\n/g'
```

Listing 3.10: Automatisiertes Abfragen aller Einträge mit curl

```
Eintrag: 21 Manueller Eintrag
Eintrag: 23 automatischer Eintrag 1
Eintrag: 24 automatischer Eintrag 2
Eintrag: 25 automatischer Eintrag 3
...
Eintrag: 122 automatischer Eintrag 100
```

Listing 3.11: Rückgabe zur Abfrage aller Einträge mit curl (1)

Anschließend können Einträge abgeändert und entfernt werden. Hierzu werden die Einträge mit den IDs 23 bis 30 überschrieben (Listing 3.12).

```
for i in {23..30}; do curl -b cookies.txt -d "id=$i&content=Neuer Eintrag $i"
↪  https://pr0ject.net/updateEntry.php; done
```

Listing 3.12: Automatisiertes Ändern bestimmter Einträge mit curl

```
Eintrag: 21 Manueller Eintrag
Eintrag: 23 Neuer Eintrag 23
...
Eintrag: 30 Neuer Eintrag 30
Eintrag: 31 automatischer Eintrag 9
Eintrag: 32 automatischer Eintrag 10
...
```

Listing 3.13: Rückgabe zur Abfrage aller Einträge mit curl (2)

Abschließend wird das Löschen der Einträge geprüft, hierfür werden alle Einträge mit den IDs 70 bis 80 gelöscht (Listing 3.14).

```
for i in {70..80}; do curl -b cookies.txt -d "id=$i"
↪  https://pr0ject.net/deleteEntry.php; done
```

Listing 3.14: Automatisiertes Löschen bestimmter Einträge mit curl

```
...
Eintrag: 68 automatischer Eintrag 46
Eintrag: 69 automatischer Eintrag 47
Eintrag: 81 automatischer Eintrag 59
Eintrag: 82 automatischer Eintrag 60
...
```

Listing 3.15: Rückgabe zur Abfrage aller Einträge mit curl (3)

Die direkte Datenbankabfrage auf dem Server spiegelt die Ausgabe der Webanwendung wider. Dennoch gilt es zu betonen, dass die Anwendung die vorgesehenen Funktionalitäten korrekt umsetzt, jedoch sie keines Falls vollständig getestet oder unempfindlich gegen bestimmte Eingaben ist. So wäre z. B. bei der Eingabe des Trennzeichens (|||) die Darstellung im Browser fehlerhaft. Auch wenn an dieser Stelle die Anwendung nur im Wissen um die Implementierung und lediglich mit erwarteten bzw. gewünschten Eingaben getestet wurde, reicht dies unseres Erachtens für die angestrebten Überprüfungen im Kontext der gesamten Infrastruktur aus. Es muss jedoch angemerkt werden, dass diese Anwendung weder vollständig getestet, noch so produktiv eingesetzt werden sollte.

4 Untersuchung der Verfügbarkeit

In diesem Kapitel soll die zuvor aufgebaute Infrastruktur nun im Hinblick auf die Verfügbarkeit untersucht werden. Das Ziel ist es herauszufinden, wie performant die verschiedenen Failover-Mechanismen tatsächlich sind und ob die entwickelte Infrastruktur den zu Beginn aufgestellten Anforderungen an die Verfügbarkeit gerecht wird. Hierzu werden verschiedene Aspekte und Metriken betrachtet, die im Folgenden auch grafisch dargestellt werden sollen.

4.1 Definition der Testszenarien

Wie in Abbildung 2.4 dargestellt, besteht die entworfene Infrastruktur aus verschiedenen Komponenten. Für alle Komponenten sollen im folgenden Testszenarien entworfen werden, in denen der Ausfall bewusst herbeigeführt wird. Dabei werden vor allem die Zeiten ermittelt, welche die Failover-Tools bzw. die Replikationsmechanismen benötigen.
Hierfür werden diese Mechanismen und die entsprechenden Testszenarien kurz dargestellt. Dabei wird eine Unterteilung in verschiedene Schichten, im Folgenden als Layer bezeichnet, vorgenommen (siehe Abbildung 4.1). Jeder von uns kontrollierbare Layer ist dabei durch ein Failover abgesichert.

- **Layer 0**: Dieser Layer liegt außerhalb des kontrollierbaren Bereichs. Er steht für Ausfallszenarien, die beide Locations gleichzeitig betreffen und diese unerreichbar machen. Großflächige Ausfälle des Internets, der Stromversorgung oder ähnliches können mögliche Szenarien darstellen. Gegen diese kann kein Failover-Mechanismus entworfen werden, entsprechend wird dieser Layer nicht weiter betrachtet.
- **Layer 1**: Unter diesem Layer wird der Ausfall einer der DNS-Resolver verstanden, welcher für die Infrastruktur verantwortlich ist. Konkret, wenn Googles oder Hetzners DNS-Service vollständig ausfällt.
- **Layer 2**: Auf dieser Ebene bedeutet ein Ausfall, dass der *HAProxy*-Server in Location-1 ausfällt. So können eingehende Requests nicht mehr entgegengenommen werden.
- **Layer 3**: Ein Ausfall auf diesem Layer heißt, dass der App-Server aus Location-1 ausgefallen ist, somit können eingehende Requests noch entgegengenommen, jedoch nicht mehr an die Anwendung weitergeleitet werden.

- **Layer 4**: Der Ausfall auf Layer 4 ist gleichbedeutend mit einem Ausfall des Datenbankservers aus Location-1, dabei ist die Anwendung erreichbar, wird aber nicht wie vorgesehen funktionieren, da keine Datenbankverbindung mehr zustande kommen kann.

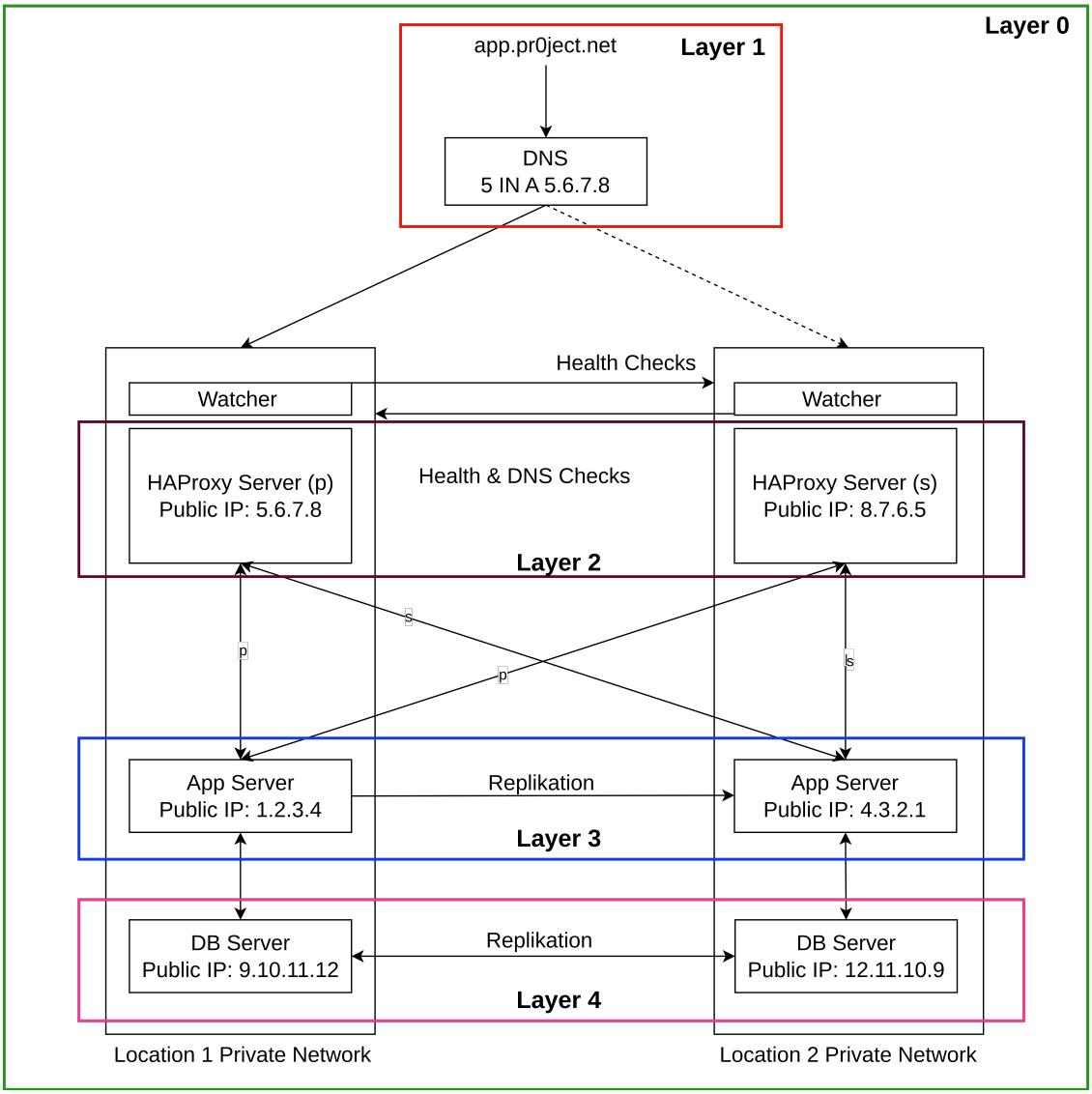

Abbildung 4.1: Unterteilung der Infrastruktur in Layer

4.2 Testen des Failover auf Layer 1

Fällt einer der für die Domain verantwortlichen DNS-Resolver aus, dann erfolgt bei Anfrage an die entsprechenden Nameserver keine Antwort und die Domain kann keiner IP zugeordnet werden.
Wie in Abschnitt 2.3.5 beschrieben, besteht der Failover-Mechanismus darin, dass nicht nur ein Resolver für die Namensauflösung verantwortlich ist. Durch die Verteilung auf Nameserver verschiedener Anbieter, sollte der Ausfall des jeweils anderen keine Auswirkungen auf die Verfügbarkeit der Infrastruktur haben.
Lediglich wenn der SOA für einen Zeitraum ausfällt, in dem alle Nameserver diesen für ihre eigene Aktualiserung abfragen, wird auch der zweite Resolver keine Einträge mehr ausliefern. Aus diesem Grund wurde für alle Nameserver eine abweichende TTL gewählt, um diesen Zeitraum möglichst lang zu gestalten.

Es ist nicht möglich, einen tatsächlichen Ausfall eines Resolvers zu erzeugen, da diese nicht von uns kontrolliert und verwaltet werden. Somit ist es nicht möglich an dieser Stelle einen Test durchzuführen. Es ist jedoch möglich zu überprüfen, ob tatsächlich alle Nameserver die Domain korrekt auflösen. Dies ist bereits in Listing 2.3 erfolgt, welches aus Gründen der Übersichtlichkeit nochmals dargestellt wird.

```
1  ~$ dig @a.gtld-servers.net NS pr0ject.net.
2  ; <<>> DiG 9.10.6 <<>> @a.gtld-servers.net NS pr0ject.net.
3  ;; AUTHORITY SECTION:
4  pr0ject.net.    172800  IN  NS  ns-cloud-e1.googledomains.com.
5  pr0ject.net.    172800  IN  NS  ns-cloud-e2.googledomains.com.
6  pr0ject.net.    172800  IN  NS  ns-cloud-e3.googledomains.com.
7  pr0ject.net.    172800  IN  NS  ns-cloud-e4.googledomains.com.
8  pr0ject.net.    172800  IN  NS  helium.ns.hetzner.de.
9  pr0ject.net.    172800  IN  NS  hydrogen.ns.hetzner.com.
10 pr0ject.net.    172800  IN  NS  oxygen.ns.hetzner.com.
```

Listing 4.1: NS-Record per dig abfragen (2)

4.3 Testen des Failover auf Layer 2

Der Ausfall auf Layer 2, also die nicht Erreichbarkeit des *HAProxy*, führt zum praktischen Ausfall der ganzen Location, da ohne Proxy keine Requests mehr verarbeitet werden können. Als Failover-Mechanismus muss der DNS-Eintrag der Domain angepasst werden,

damit dieser auf den *HAProxy* der zweiten Location zeigt und Request wieder verarbeitet werden können. Für die darunter liegenden Ebenen hat dies keine Auswirkung, da beide *HAProxy* den App-Server der Location-1 ansprechen, sollte dieser erreichbar sein.

Die Änderung der DNS-Einträge bei Hetzner und Google wird im Ausfall durch einen Watcher (siehe Listing 2.15) vorgenommen. Im Folgenden soll betrachtet werden, wie schnell das gesamte Failover, also vom Moment der Abschaltung von *HAProxy* zur Wiedererreichbarkeit, abläuft. Zusätzlich soll die Zeit die jede Komponente benötigt gemessen werden, um darzustellen, welche Mechanismen den größten Einfluss auf das Failover haben.

Hierfür wird der *HAProxy* abgeschaltet und der Failover-Prozess beobachtet. Für die Darstellung wird der Zeitpunkt des Ausfalls festgehalten, wann der Watcher diesen bemerkt, wann die ersten Resolver die neue IP ausliefert und wie lange es braucht, bis der letzte Resolver die korrekte Auflösung vornimmt.

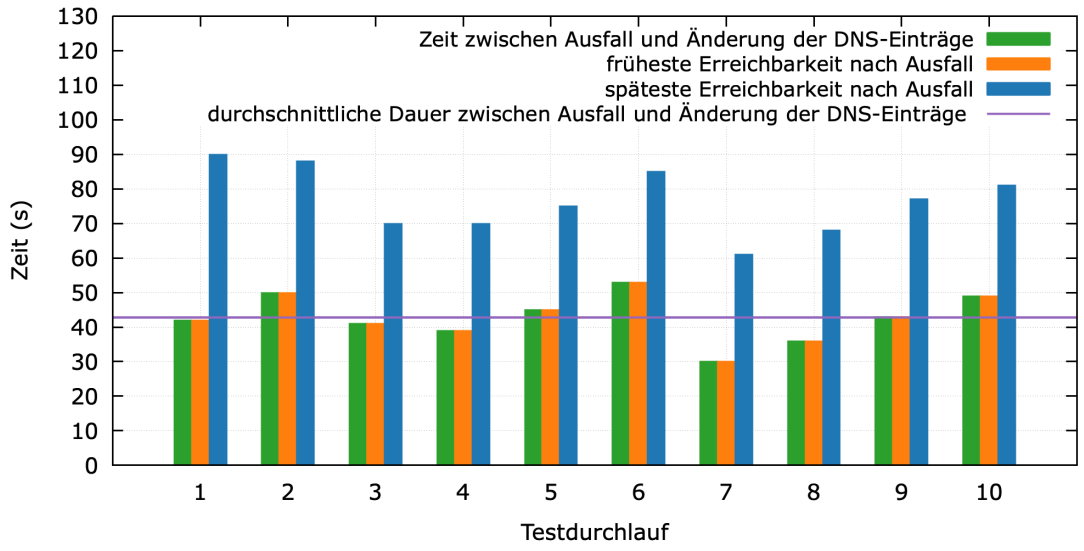

Abbildung 4.2: Zeitmessung der Änderung der DNS-Einträge

Wie in Abbildung 4.2 zu sehen, braucht der gesamte Prozess zwischen 30 und 90 Sekunden. Wobei der niedrigste Wert erreicht wird, wenn man die schnellsten DNS-Resolver betrachtet und der höchste Wert die letzte Aktualisierung eines Resolvers darstellt.
Die Watcher-Komponente hat dabei einen Anteil von 30 bis 53 Sekunden am gesamten Prozess. Durchschnittlich braucht der Watcher 42,8 Sekunden, um den Ausfall zu registrieren und die IP-Adresse in den DNS-Einträgen zu ändern.

Etwa ein Drittel der DNS-Resolver registriert die Änderung dabei unverzüglich bzw. in unter einer Sekunde. Die Tabelle 4.1 zeigt die Aktualisierungszeit der Test-Resolver in

fünf Sekunden Schritten. Diese Werte sind dabei nicht absolut, sondern aus einer Vielzahl von Testdurchläufen errechnet.

Tabelle 4.1: Änderung des DNS-Eintrags nach Sekunden

Sekunden	**0**	**5**	**10**	**15**	**20**	**25**
Skydns			X			
FortinetInc				X		
OpenDNS	X					
Sprint	X					
AliyunComputing-Co.Ltd			X			
OskarEmmenegger			X			
VeriSignGlobalRegis-tryServices			X			
Google		X				
DanielCid					X	
nemox.net			X			
ProbeNetworks	X					
Cloudflare	X					
TeleDanmark	X					
Prioritytelecom-SpainS.A	X					
Quad9			X			
4DDataCentresLtd	X					
LiquidTelecommuni-cationsLtd						X
AssociationGitoyen					X	
CLOUDITYNetwork				X		
TeknetYazlim			X			
MarcatelCom	X					
TTDotcomSdnBhd	X					

Fortsetzung auf nächster Seite

Tabelle 4.1: Änderung des DNS-Eintrags nach Sekunden

Sekunden	**0**	**5**	**10**	**15**	**20**	**25**
AT&T			X			
SiteHost						
LGDacomCorporation	X					
DigitalOceanLLC				X		
MangoTeleservicesLimited	X					
AirDesignBroadcastMediaPvtLtd	X					
UniversoOnlineS.A			X			
CMPakLimited			X			
PacificInternet				X		
KPN			X			

Die Resolver liegen dabei auf verschiedenen Kontinenten und Ländern. Der Resolver *SiteHost* hat über den gesamten Testzeitraum keine Aktualisierung vorgenommen und wurde aus den Werten der obigen Abbildungen rausgerechnet.

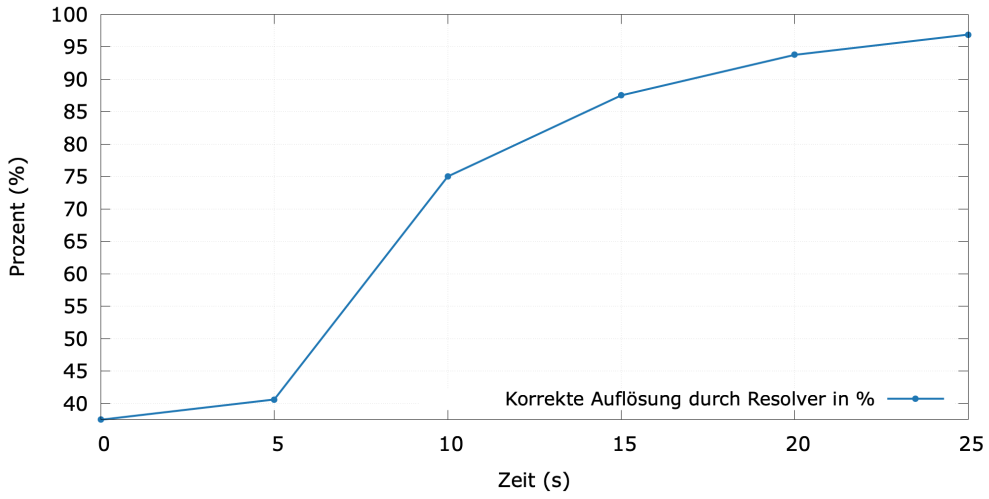

Abbildung 4.3: Aktualisierung des DNS-Eintrages nach Zeit

Wird der prozentuale Verlauf aus Abbildung 4.3 betrachtet, zeigt sich, dass nach fünf Sekunden bereits 40 Prozent der Resolver den neuen Eintrag korrekt auflösen. Nach 20 Sekunden wird eine Abdeckung von 90 Prozent erreicht. Der größte Anstieg ist im Bereich von fünf zu zehn Sekunden erreicht. Da *SiteHost* keine Aktualiserung innerhalb des Testzeitraums (60 Minuten) vorgenommen hat, wird eine Abdeckung von 100 Prozent nicht erreicht.

Die Änderung des DNS-Eintrags und die darauffolgende Aktualisierung bei den Test-Servern, stellt den Abschluss des Failover dar. Somit liegt die benötigte Zeit für den Failover in Layer 2 in einem Bereich von 30 bzw. 90 Sekunden, je nach zugrundeliegendem DNS-Resolver des Clients.
Im Browser wird der entstandene Timeout des Systems durch eine Meldung an den Client dargestellt. In dieser wird der Client aufgefordert eine Minute zu warten, bevor weitere Aktionen vorgenommen werden können.

4.4 Testen des Failover auf Layer 3

Ein Ausfall von Layer 3 bedeutet die Nichterreichbarkeit des Anwendungsservers auf Location-1. In diesem Fall schwenkt *HAProxy* auf den Anwendungsserver auf Location-2 um, wodurch dieser für die Beantwortung von Requests verantwortlich ist. In der Abbildung 4.4 sind die Ergebnisse eines Tests dargestellt, welcher untersucht, wie schnell das Umschwenken durch den *HAProxy* erfolgt.

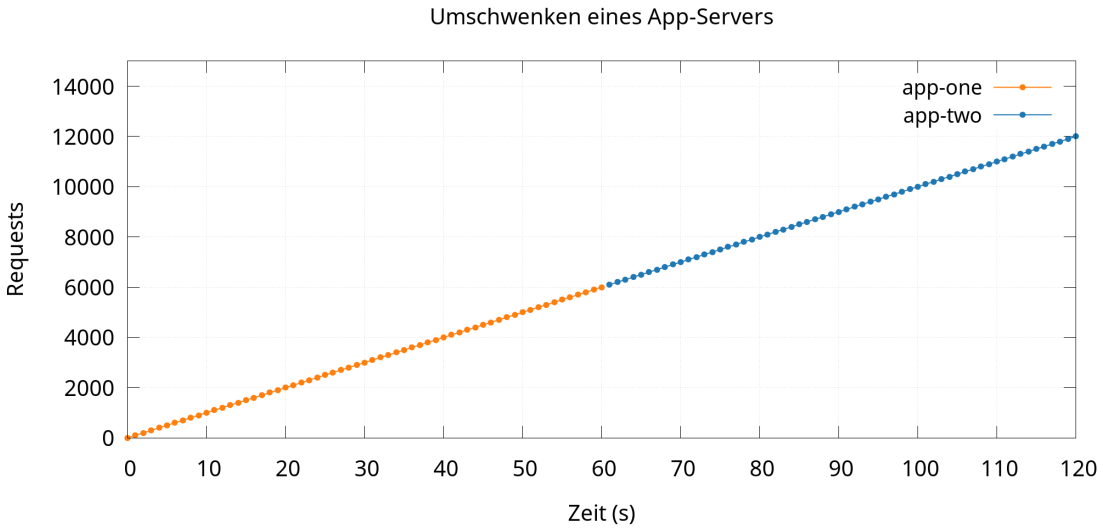

Abbildung 4.4: Umschwenken eines Applikationsservers

In diesem Test wurden über einen Zeitraum von 120 Sekunden jeweils 100 Requests pro Sekunde an die Anwendung geschickt und ausgewertet, zu welchem Zeitpunkt, welcher Anwendungsserver auf die Requests antwortet.
Nach 60 Sekunden wurde der Ausfall von Layer 3 simuliert, indem der *Apache2*-Server auf dem Anwendungsserver auf Location-1 ausgeschaltet wurde. In der Abbildung 4.4 ist schließlich durch den Farbwechsel erkennbar, dass kurz nach 60 Sekunden der Anwendungsserver auf Location-2 die folgenden Requests beantwortet hat. *HAProxy* schwenkt dabei in weniger als einer Sekunde auf den Anwendungsserver auf Location-2 um. Es sind im Zuge des Tests keine Request verloren gegangen bzw. nicht von einer Location beantwortet worden.
Durch eine erhöhte Parallelisierung der Abfragen war zu beobachten, dass einzelne Request während des Failover verloren gehen. Es war in keinem Fall mehr als einen fehlerhaften Request je Testdurchlauf.

Auch wenn die Requests in über 99,99 Prozent der Fälle an das Failover-System weitergeleitet werden und die Anwendung als Programmcode als exakte Kopie vorliegt, finden sich die Sessions nicht automatisch auf beiden Locations wieder.
Aus diesem Grund wurde getestet, in welcher Zeit Sessions zwischen den Anwendungsservern übertragen werden können. Hierfür wurden zwischen 100 und 1000 Sessions übertragen und die Zeit gemessen. Zusätzlich wurde die durchschnittliche Übertragungszeit je Session in Millisekunden angegeben.

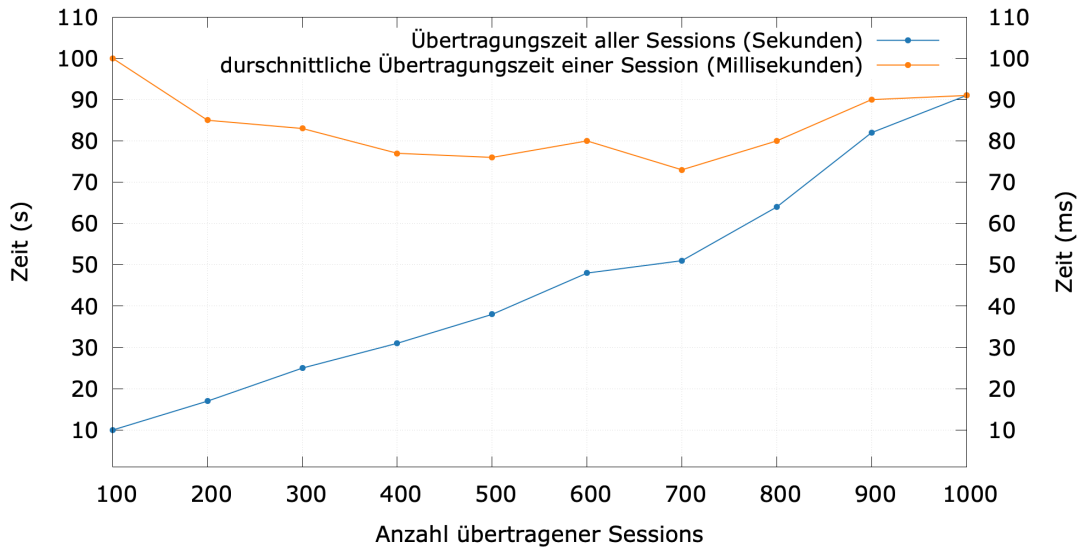

Abbildung 4.5: Übertragungsdauer der Sessions

Wie zu sehen, steigt die Dauer der Übertragung mit der Anzahl der Sessions stetig an. Die durchschnittliche Übertragungsdauer je Session (Datei) nimmt dabei geringfügig ab.

Zu Beginn ist dieser Abfall besonders stark zu beobachten, während er mit Zunahme der Dateien immer weiter abnimmt. Insgesamt dauert die Übertragung je Session zwischen 75 und 100 Millisekunden.

4.5 Testen des Failover auf Layer 4

Ein Ausfall von Layer 4 beschreibt den plötzlichen Verlust des Datenbankservers aus Location-1. Da für das Datenbanksystem, wie bereits erwähnt, eine Replikation zwischen Location-1 und Location-2 aufgebaut wurde, sollen im Folgenden einige Parameter bestimmt werden, um den Failover-Prozess auf Location-2 besser untersuchen zu können.

```
1  #!/bin/bash
2  server="49.13.74.40"
3  num_pings=100
4  ping_result=$(ping -c $num_pings $server)
5
6  min_latency=$(echo "$ping_result" | grep "min/avg/max" | cut -d " " -f4  | cut
   ↪  -d'/' -f1)
7  avg_latency=$(echo "$ping_result" | grep "min/avg/max" | cut -d " " -f4  | cut
   ↪  -d'/' -f2)
8  max_latency=$(echo "$ping_result" | grep "min/avg/max" | cut -d " " -f4  | cut
   ↪  -d'/' -f3)
9
10 echo "Minimale Latenz: $min_latency ms"
11 echo "Durchschnittliche Latenz: $avg_latency ms"
12 echo "Maximale Latenz: $max_latency ms"
```

Listing 4.2: Untersuchung der Netzwerklatenz

Im ersten Schritt wird hierfür das Netzwerk genauer betrachtet.
Im Listing 4.2 wurde ein Bash-Skript implementiert, das mit `ping` die Netzwerklatenz zwischen Location-1 und Location-2 ermittelt. Das Skript wurde auf dem Applikationsserver aus Location-1 gestartet und hat eine minimale Latenz von 35,245 ms, eine durchschittliche Latenz von 46,176 ms und eine maximale Latenz von 138,432 ms ermittelt.
Es dauert also durchschittliche 46,176 ms, um ein Datenpaket von Location-1 aus nach Location-2 zu senden.
Ein weiterer Paramter der im Kontext der Netzwerkperformance bestimmt werden soll, ist

der Datendurchsatz. Mit `iperf3` lässt sich hierfür eine TCP-Verbindung zwischen zwei Servern aufbauen und die Performance des Netzwerks genauer untersuchen.

```
Connecting to host 35.242.191.59, port 5201
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   138 MBytes   116 Mbits/sec  577             sender
[  5]   0.00-10.04  sec   135 MBytes   113 Mbits/sec                  receiver
```

Listing 4.3: Untersuchung des Datendurchsatzes

Im Listing 4.3 ist das Ergebnis dieses Tests dargestellt. In einem 10 Sekunden andauernden Intervall wurden dabei 138 MBytes bei einer Bitrate von 116 Mbits pro Sekunde vom Sender übertragen. Dabei gab es 577 Retransmissions, was bedeutet, dass in 577 Fällen ein Datenpaket erneut verschickt werden musste, da es zu Fehlern bei der Übertragung gekommen ist.

Ein weiterer Messwert, der bei der Untersuchung des Failover-Prozesses durch die Replikation von Bedeutung ist, ist die CPU-Auslastung des Datenbankservers während der Durchführung von *SQL*-Statements.

```
#!/bin/bash
count="25"
while true; do
    for i in {1..$count};do
        mysql -e "insert into bachelorarbeit.diary (username,entry) values
        ↪  ('demo','Testeintrag');" &
    done
sleep 1
done
```

Listing 4.4: Ausführung von 25 Insert-Statements pro Sekunde

Im Listing 4.4 wurde für diesen Test ein Bash-Skript entwickelt, das einmal pro Sekunde eine bestimmte Anzahl von Insert-Statements durchführt. Das Ergebnis dieses Versuchs ist in der Darstellung 4.6 visualisiert.
Es wird hier deutlich, dass bei fünf Insert-Statements pro Sekunde die CPU-Auslastung unter 25 Prozent und bei zehn Insert-Statements bereits zwischen 45 und 53 Prozent liegt. Die Kurve für 15 Insert-Statements pro Sekunde zeigt an manchen Stellen bereits eine Last von fast 80 Prozent.

Die Last bei 20 Insert-Statements befindet sich im Bereich von 95 bis 100 Prozent, während bei 25 Statements pro Sekunde die Auslastung nahezu bei 100 Prozent liegt.
Die erzeugte CPU-Last durch 25 Statements pro Sekunde reicht also, um den Datenbankserver einer Location vollständig auszulasten.

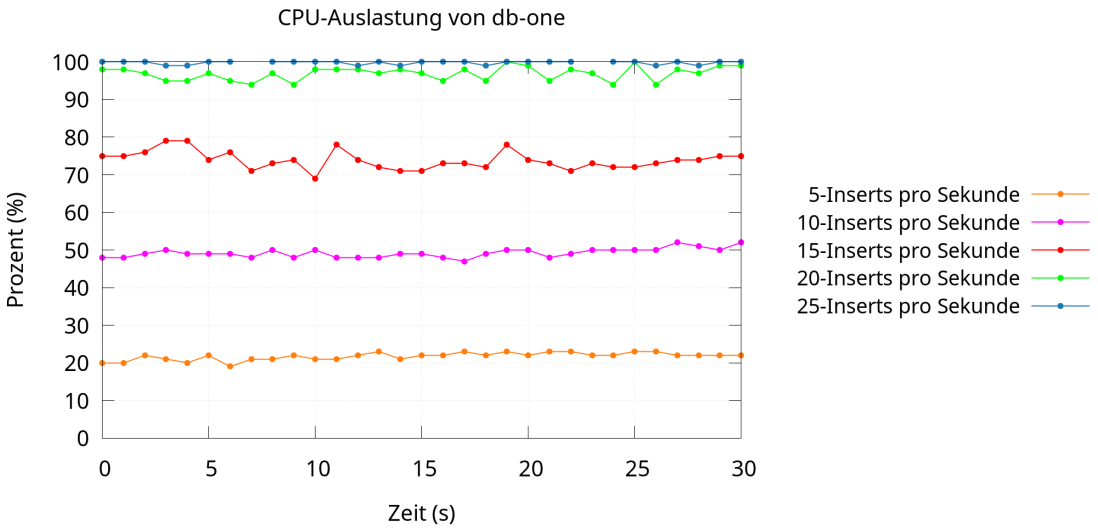

Abbildung 4.6: CPU-Last des Datenbankservers

Neben dem Test mit der Ausführung der verschiedenen Insert-Statement-Prozesse pro Sekunde wird im Folgenden die Performance der Replikation genauer untersucht.
Hierfür wird insbesondere der Parameter `seconds_behind_master` überwacht. Dieser beschreibt die Differenz zwischen dem auf dem Primary protokollierten Zeitstempel und dem Replica-Zeitstempel in Sekunden, für die Transaktion, welche gerade durch den Replica-Server verarbeitet wird [36].

```
1 while true; do
2 echo "$(date +%H:%M:%S) $(mysql -e "show slave status\G" |grep "Seconds" | cut
  ↪  -d " " -f 11)" >> seconds-behind-master.log
3 sleep 1
4 done
5
```

Listing 4.5: Auswertung des Parameters Seconds_behind_master

Da das Datenbanksystem eine maximale Anzahl von 25 Insert-Statements pro Sekunde verarbeiten kann, bevor die CPU-Auslastung zu hoch wird, ist dies auch der Grenzwert für

die maximale Anzahl durchgeführter Statements bei der Analyse der Replikation.
In keinem Szenario ist dieser Wert dabei größer als null geworden. Es ist also von einer Replikationslatenz unter einer Sekunde auszugehen, da durch das Netzwerk eine Latenz von durchschittlich 46.176 ms existiert.

Ein Ausfall des Datenbankservers wirkt sich außerdem nicht anders auf Requests an die Testanwendung aus, als bei einem Ausfall des aktiven Anwendungsservers.

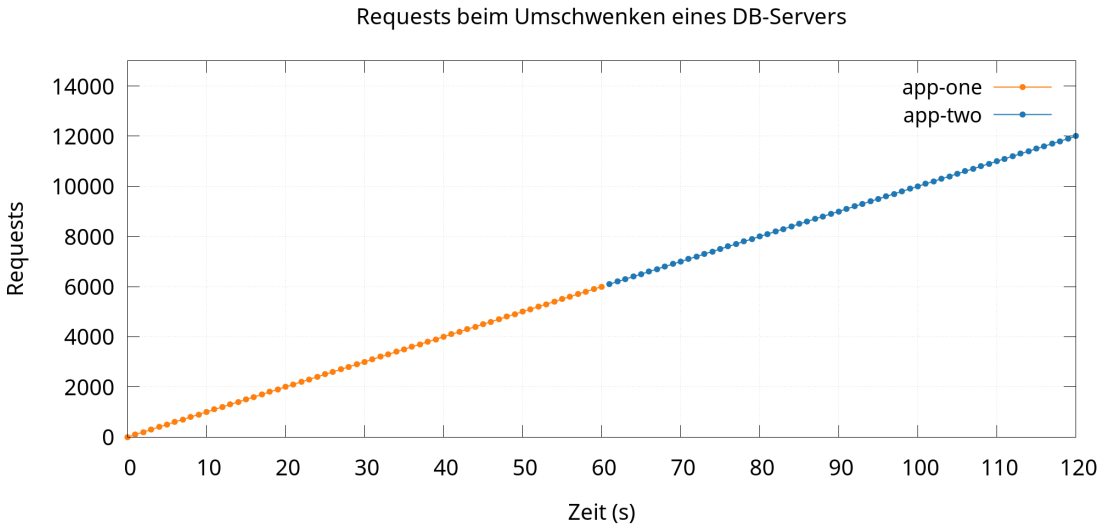

Abbildung 4.7: Umschwenken eines Datenbankservers

Folglich entsteht in der Abbildung 4.7 eine ähnliche Darstellung wie in der Abbildung 4.4. Durch den bereits erläuterten Endpunkt der Anwendung zur Überprüfung der Datenbankverbindung, ändert auch hier der *HAProxy* in unter einer Sekunde das Backend wodurch kein Request in der Abbildung verloren geht.

5 Diskussion der Ergebnisse

Aus den verschiedenen Tests zur Untersuchung der Verfügbarkeit der gebauten Infrastruktur geht hervor, dass die *HAProxy*-Server, bzw. der Failover-Prozess über DNS grundsätzlich als kritischste Komponenten zu betrachten sind.
Der Failover-Prozess bei einem Ausfall dauert hier mit 30 bis 90 Sekunden mit Abstand am längsten. Positiv zu bewerten ist, dass die verschiedenen DNS-Resolver zum Großteil sehr schnell die Änderung des A-Records übernehmen.
Bereits nach fünf Sekunden haben 40 Prozent der Resolver den aktualisierten A-Record eingepflegt und liefern diesen entsprechend bei DNS-Anfragen aus.
Da die TTL von der Domain der Testanwendung fünf Sekunden beträgt, ist hier vor allem spannend, wieso gerade 40 Prozent der getesteten Resolver die Änderung nach genau fünf Sekunden umsetzten.
Es könnte sich bspw. positiv auf die benötigte Zeit beim Ausrollen der Änderungen auswirken, wenn die Zone *pr0ject.net* noch bei weiteren Nameservern verwaltet wird, da einige Resolver ggf. verschiedene Nameserver in ihren Systemen konfiguriert haben. Unter Umständen ist es so möglich die Änderungen schneller bei den verschiedenen Resolvern zu verteilen und die Ausfallzeit so noch weiter zu reduzieren.

Insgesamt ist der Failover-Prozess über DNS jedoch sehr performant. Dies war in den ersten manuellen Tests nicht unbedingt zu erwarten, da dort mit einzelnen Resolvern auch deutlich schlechtere Ergebnisse erzielt wurden. Dies zeigt, dass die Ausfallzeit in diesem Fall stark vom jeweiligen Resolver des Clients abhängig ist. Darüber hinaus spielt bei der Übernahme einer neuen IP-Adresse vom jeweiligen Resolver des Clients, die Browser-Cache Konfiguration eine nicht unerhebliche Rolle, wozu es noch offene Fragestellungen gibt.
Im Hinblick auf die Optimierung des Failover-Prozesses bei einem Ausfall eines *HAProxy*-Servers kann auch durch weitere Funktionen im *JavaScript*-Code des Frontends die Stabilität verbessert werden. So wäre es möglich, eingegebene Daten eines Clients in einem lokalen Cache zu sichern um diese ggf. bei einem Ausfall einer gesamten Location während der Downtime zu sichern. Folglich könnte ein Client bei einem Ausfall nicht so leicht seine Daten verlieren, was die Stabilität der Anwendung ebenfalls verbessern würde.

Ein Punkt, an dem die Zeit für den Failover reduziert werden könnte, ist die Konfiguration der Watcher-Infrastruktur. Das Skript, welches den Failover-Prozess initiiert und damit die IP-Adresse bei den zuständigen Nameservern aktualisiert, überwacht die Verfügbarkeit der *HAProxy*-Server per `ping` und geht erst nach 20 fehlgeschlagenen *Ping*-Prozessen in 20 Sekunden von dem Ausfall eines *HAProxy* aus.
Dieser Wert lässt sich noch reduzieren, wodurch zumindest der Failover-Prozess schneller

angestoßen werden könnte, was den Prozess insgesamt beschleunigen würde. Dabei muss jedoch überprüft werden, ob es zu einem ungewollten Failover kommen könnte, weil bspw. Pings verloren gehen, obwohl die Infrastruktur noch wie geplant funktioniert.
Insgesamt funktioniert der Failover-Prozess aber trotzdem viel schneller als erwartet. DNS ist als sehr stabiles System zu betrachten und hat im Rahmen der Tests überzeugt. Grundsätzlich ist die Wahrscheinlichkeit, dass die Nameserver von Hetzner und Google gleichzeitig ausfallen, als sehr gering einzustufen. An dieser Stelle kann man also von einer hohen Verfügbarkeit durch die Anbieter ausgegeangen werden, die so bspw. mit *Keepalived* nur schwer zu erreichen wäre.

Das Umschwenken auf der Ebene von Layer 3 durch den *HAProxy* ist sehr schnell. Selbst bei 100 Requests pro Sekunde an die Testanwendung lässt sich bei dem Failover-Prozess vom Anwendungsserver aus Location-1 auf Location-2 kein Zeitpunkt feststellen, an dem das Failover durch *HAProxy* mindestens eine Sekunde dauert.
Der Bottleneck an dieser Stelle ist die Replikation der Sessions, da hier schnell viele Dateien entstehen können. Die Session-Dateien werden über SCP übertragen, wodurch allein beim Verbindungsaufbau schon ein gewisser Zeitaufwand entsteht.
Fällt ein Anwendungsserver aus, während die Replikation der Session-Dateien aktiv ist, kann schnell eine Situation entstehen, in der bei einem Failover nicht alle Sessions auf dem neuen Anwendungsserver vorhanden sind. So können ggf. Daten verloren gehen, da sich ein Client neu am System anmelden muss und bspw. ein Request zur Anlegung eines Datensatzes in der Testanwendung nicht erfolgreich durchgeführt werden kann.

Eine Möglichkeit zur Optimierung wäre die Verwendung von Archives, da die Übertragung von vielen kleinen Dateien nicht so performant ist wie von einer Großen. Allerdings gehen in dieser Überlegung alle Sessions verloren, wenn die Übertragung fehlschlägt.
Optimal wäre hier der Einsatz von *Redis*, da es vor allem für den Einsatz als Caching-System entwickelt wurde. Dies würde jedoch wieder eine höhere Komplexität mit sich bringen, da auch hier keine einzelne *Redis*-Instanz ausreicht. Es müsste ein standortübergreifendes Cluster aufgebaut werden, bei dem auch Faktoren wie Backup, Monitoring und Failover bedacht werden müssten. Es muss an dieser Stelle wieder ein ausgewogenes Verhältnis zwischen Komplexität und Performance gefunden werden, wodurch die Verwendung eines *Tar* in Kombination mit SCP oft ausreichend sein könnte.
Generell ist aber davon auszugehen, dass es immer einen Punkt gibt, an dem ein Request genau zum Failover eintrifft und nicht sauber beantwortet werden kann. Dieser Zeitraum lässt sich nur minimieren. Wichtig ist, dass für die zu erwartende Last an eine Anwendung eine hinreichende Stabilität sichergestellt ist.

Ein Ausfall eines Datenbankservers einer Location ist, wie bereits erwähnt, genau wie der Ausfall eines Applikationsservers zu betrachten. Hier übernimmt die *HAProxy*-Instanz ebenfalls den Failover-Prozess in deutlich weniger als einer Sekunde.

Anders als zuvor ist die Replikation hier nicht als Bottleneck zu betrachten, da diese für die maximale mögliche Anzahl von 20 Insert-Statements pro Sekunde eine Latenz von weniger als einer Sekunde aufweist und damit zu vernachlässigen ist. Ein Risiko im Hinblick auf die Verfügbarkeit sind hier lediglich Szenarien, die dazu führen könnten, dass die Replikation nicht mehr wie vorgesehen funktioniert und einen inkonsistenten Datenstand erzeugt. Im Rahmen diverser Tests konnte so ein Szenario allerdings nicht erzeugt werden.
Im Bereich der Performance des Datenbanksystems, also der Anzahl von Statements pro Sekunde existiert allerdings Optimierungspotenzial.
Lediglich 20 Insert-Statements pro Sekunde durchführen zu können, erscheint auf den ersten Blick doch etwas wenig. Durch eine angepasste Konfiguration des *MariaDB*-Servers könnte dies eventuell noch verbessert werden.
Auch durch Skalierung, bspw. durch Load-Balancing der Anfragen an das Datenbanksystem nach dem Round-Robin-Prinzip mit einer Kombination aus *HAProxy* als TCP-Loadbalancer und einem ganzen Datenbankcluster an einem Standort ließe sich die Performance vermutlich auch ohne ein Hardware-Upgrade der VMs verbessern.

Die verschiedenen Untersuchungen zur Verfügbarkeit der gebauten Infrastruktur sind insgesamt sehr positiv verlaufen. Anhand der Testergebnisse lässt sich sagen, dass die Infrastruktur bei einem Ausfall, im schlimmsten Fall, zwischen 30 und 90 Sekunden nicht erreichbar sein wird.

6 Resümee und Ausblick

Abschließend kann gesagt werden, dass die in dieser Thesis aufgestellten Anforderungen an eine hochverfügbare Multi-Cloud-Infrastruktur erfüllt werden konnten. Es ist gelungen die Infrastruktur so aufzubauen, dass sie keinen Single Point of Failure besitzt. Darüber hinaus wurden, mit Ausnahme der Google-Cloud-CLI, auf proprietäre Lösungen verzichtet.
Gleichzeitig zeigt die Umsetzung aus unserer Sicht eine gewisse Einfachheit. Es kommen Lösungen wie *Bash*- bzw. *PHP*-Skripte, *HAProxy* als Reverse-Proxy, *Apache2* als Webserver und *MariaDB* als Datenbankmanagementsystem zum Einsatz. Lediglich die Datenbankreplikation weist eine gewisse Komplexität auf, die auch durch die TLS-Verschlüsselung verursacht wird. Im Allgemeinen werden jedoch keine Technologien eingesetzt, die nicht im Laufe des Studiums Teil des Lehrplans sind.
Eine solche Einfachheit kann unseres Erachtens nur dadurch erreicht werden, indem die Planung für eine solche Infrastruktur gut vorbereitet und immer wieder evaluiert wird. So wurden verschiedene Technologien und Software in der Planung betrachtet und Entscheidungen immer wieder durchdacht oder revidiert.

Die Ergebnisse aus den Failover-Tests haben gezeigt, dass die Failover-Mechanismen nicht nur technisch gut funktionieren, sondern auch teilweise die Erwartungen übertroffen haben. In den Vorbesprechungen vor Beginn dieser Thesis kamen immer wieder Werte im Bereich von fünf bis zehn Minuten für ein Failover zur Sprache. Die erreichten Werte von 30 bis 90 Sekunden haben die von uns prognostizierten Zeiten deutlich unterboten. Besonders die schnelle Reaktion der getesteten DNS-Resolver hat uns positiv überrascht. Hier wurde zu Beginn mit deutlich längeren Zeiträumen bis zur Aktualisierung der Einträge gerechnet.

Es hat sich wiederum gezeigt, dass es noch Verbesserungspotenziale gibt. Der Übertrag der Sessions kann durch unterschiedliche Mechanismen sehr wahrscheinlich deutlich beschleunigt werden. Ebenso zeigt die Umsetzung des DNS-Watchers, dass dieser optimiert werden kann. Hierfür sind aber in beiden Fällen weitere Überlegungen und Tests nötig. Dennoch liegt in diesen beiden Punkten das größte Verbesserungspotenzial, da gerade diese Mechanismen die längste Zeit für ein Failover benötigen.

Bei allen Test und Überlegungen sollte dennoch betont werden, dass es sich um ein kleines Testsystem aus insgesamt nur sechs Servern handelt. Unseres Erachtens sollten sich die meisten Lösungen gut skalieren lassen, allerdings ergeben sich dabei auch Grenzen. So könnte sich bei der Datenbankreplikation eine Übertragungszeit ergeben, die bei einer hohen Anzahl von Transaktionen nicht mehr weiter reduzierbar und dennoch zu groß für ein so schnelles Failover sein könnte.
Auch die Testanwendung ist sowohl in der Funktionalität als auch der Interaktion mit dem Server auf einen engen Bereich begrenzt. So würde bspw. eine Anwendung, welche über

einen Dateiupload verfügt, die Komplexität deutlich erhöhen. Gerade durch die Aufteilung in unterschiedliche geografische Zonen, bei unterschiedlichen Anbietern, könnte die Replikation großer Datenmengen deutlich erschweren.

Ebenso stellt sich die Frage, inwieweit die aufgebaute Infrastruktur im wirtschaftlichen Sinne praxistauglich ist. Im gezeigten Entwurf wird die gesamte Location redundant vorgehalten. Dabei müssen alle Server durchgehend in Betrieb sein und somit verdoppeln sich die Kosten für die gesamte Anwendung gegenüber einer Infrastruktur ohne diese Architektur.
In diesen Überlegungen spielen natürlich auch die entgangenen Kosten bei Ausfall eine Rolle und welche Ausfallzeiten im wirtschaftlichen Sinne als akzeptabel gelten. So könnte die Infrastruktur so umgebaut werden, dass einige der Komponenten lediglich nach dem Ausfall in Betrieb genommen werden können, um so Kosten zu sparen, wobei sich die Ausfallzeit jedoch erhöhen würde.

Trotz dieser Punkte und dem noch vorhandenen Verbesserungspotenzial kann das Ziel der vorliegenden Arbeit aus unserer Sicht als erfüllt angesehen werden. Dieser Entschluss ergibt sich vor allem aus den Resultaten der Testergebnisse, welche uns teilweise überrascht haben.
Wir sind auch persönlich mit dem gezeigten Ergebnis überaus zufrieden. Nach neun Wochen intensiver Auseinandersetzung konnten wir, trotz der uns schon vorab bekannten Technologien, eine Vielzahl von neuem Wissen und Ansätzen gewinnen. Gerade die aus unserer Sicht hohe Geschwindigkeit in welcher *HAProxy* die Failover vornehmen konnte, die Latenzen der Datenbankreplikation und die Aktualisierung der DNS-Resolver, haben wir so nicht erwartet.
Durch den von uns gewählten Studienschwerpunkt der Systemintegration war das Thema der Bachelorthesis von besonderem Interesse, welches sich über den gesamten Verlauf der Arbeit gehalten bzw. noch verstärkt hat. Gerne hätten wir uns mit einigen Aspekten aus dieser Arbeit noch intensiver auseinandergesetzt, dies war durch den begrenzten Rahmen und die vorgegebene Zeitspanne leider nicht möglich.

Literaturverzeichnis

[1] N. Lewanczik. „Die 50 meistbesuchten Websites der Welt: Search, Social Media und Sex.“ de. (Jan. 2023), Adresse: https://onlinemarketing.de/cases/50-meistbesuchte-websites-welt-search-social-media-sex (besucht am 23. 05. 2023).

[2] J. Brien. „Azure, AWS, Google: Das sind die 10 größten Cloud-Provider der Welt.“ de. (Juni 2023), Adresse: https://t3n.de/news/azure-aws-google-cloud-provider-ranking-1558923/ (besucht am 17. 06. 2023).

[3] „Amazon Web Services: Das (fast) unsichtbare Rückgrat des Internets.“ de. (Apr. 2019), Adresse: https://upload-magazin.de/33219-amazon-web-services/ (besucht am 24. 06. 2023).

[4] „Die Sonne hinter Amazons Wolken.“ de. (März 2019), Adresse: https://www.faz.net/pro/d-economy/cloudsparte-aws-die-sonne-hinter-amazons-wolken-16067635.html?printPagedArticle=true#pageIndex_0 (besucht am 03. 07. 2023).

[5] „Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region.“ en. (März 2017), Adresse: https://aws.amazon.com/de/message/41926/?nc1=h_ls (besucht am 07. 07. 2023).

[6] S. Grüner. „Tippfehler führt zu zehnstündigem Cloud-Ausfall.“ de. (Mai 2023), Adresse: https://www.golem.de/news/microsoft-azure-tippfehler-fuehrt-zu-zehnstuendigem-cloud-ausfall-2306-174700.html (besucht am 06. 07. 2023).

[7] S. Grüner. „Google-Rechenzentrum nach Wasserschaden und Brand offline.“ de. (Apr. 2023), Adresse: https://www.golem.de/news/paris-google-rechenzentrum-nach-wasserschaden-und-brand-offline-2304-173764.html (besucht am 06. 07. 2023).

[8] B. für Sicherheit in der Informationstechnik, „Hochverfügbarkeit eine herausfordernde Aufgabenstellung für ein professionelles IT-Service Management,“ 2013.

[9] M. Kleppmann, *Designing data-intensive applications : the big ideas behind reliable, scalable, and maintainable systems*, First edition. Beijing: O’Reilley, März 2017, 1 Online-Ressource (xix, 590 Seiten) : Illustrationen, Diagramme, Karten, ISBN: 9781491903117 and 1491903112 and 9781491903117. Adresse: https://ebookcentral.proquest.com/lib/suub/detail.action?docID=4825244.

[10] M. Schwartzkopff, *Clusterbau: Hochverfügbarkeit mit Linux*, 3. Aufl. O’Reilly Media Germany, 2012, 1 volume, ISBN: 9783868993585. Adresse: https://learning.oreilly.com/library/view/~/9783868998764/?ar.

[11] „Keepalived User Guide.“ en. (n.a), Adresse: https://www.keepalived.org/doc/index.html (besucht am 18. 10. 2023).

[12] I. van Beijnum, *BGP: Building Reliable Networks with the Border Gateway Protocol*. O'Reilly Media, 2002, ISBN: 9781449390822. Adresse: https://books.google.de/books?id=9zZShNhlB6oC.

[13] Google. „BGP-Sitzungen erstellen.“ de. (Aug. 2023), Adresse: https://cloud.google.com/network-connectivity/docs/router/how-to/configuring-bgp?hl=de (besucht am 18. 10. 2023).

[14] C. Liu und P. Albitz, *DNS and BIND*, 5th ed. O'Reilly, 2006, xxi, 616, ISBN: 0596100574 and 9780596100575 and 0596550006. Adresse: https://learning.oreilly.com/library/view/~/0596100574/?ar.

[15] IBM. „Keepalived and HAProxy.“ en. (Juli 2023), Adresse: https://www.ibm.com/docs/en/linux-on-z?topic=available-keepalived-haproxy (besucht am 18. 10. 2023).

[16] Oracle. „Load Balancing and High Availability Configuration.“ en. (n.a), Adresse: https://docs.oracle.com/en/operating-systems/oracle-linux/6/admin/ol6-loadbal.html (besucht am 18. 10. 2023).

[17] A. Critelli. „Setting up a Linux cluster with Keepalived: Basic configuration.“ en. (Mai 2020), Adresse: https://www.redhat.com/sysadmin/keepalived-basics (besucht am 18. 10. 2023).

[18] VMWare. „Configure HAProxy with Keepalived.“ en. (Okt. 2022), Adresse: https://docs.vmware.com/en/vRealize-Operations/8.10/vrops-manager-load-balancing/GUID-EC001888-776B-42D5-9843-719EF08AB940.html (besucht am 18. 10. 2023).

[19] I. S. Consortium, *dig - DNS lookup utility*, Aug. 2023.

[20] „Übersicht über die gcloud CLI.“ de. (Sep. 2023), Adresse: https://cloud.google.com/sdk/gcloud?hl=de (besucht am 09. 09. 2023).

[21] C. Ewerhart und P. W. Schmitz, „Der lock in effekt und das hold up problem,“ 1997.

[22] Nftables. „NFT Manpage.“ en. (Juni 2023), Adresse: https://www.netfilter.org/projects/nftables/manpage.html (besucht am 25. 10. 2023).

[23] Fail2ban Community. „Fail2ban Wiki.“ en. (2023), Adresse: https://github.com/fail2ban/fail2ban/wiki (besucht am 25. 10. 2023).

[24] „VPC-Netzwerke.“ de. (Sep. 2023), Adresse: https://cloud.google.com/vpc/docs/vpc?hl=de (besucht am 25. 09. 2023).

[25] Hetzner Cloud GmbH. „Hetzner Cloud API.“ en. (2023), Adresse: https://docs.hetzner.cloud/ (besucht am 19. 10. 2023).

[26] Apache Foundation. „Apache Performance Tuning.“ en. (n.a), Adresse: https://httpd.apache.org/docs/2.4/misc/perf-tuning.html (besucht am 18. 10. 2023).

[27] HAProxy. „Management Guide.“ en. (Mai 2023), Adresse: https://docs.haproxy.org/2.5/management.html (besucht am 18. 10. 2023).

[28] „Haproxy Starter Guide.“ en. (Aug. 2023), Adresse: https://docs.haproxy.org/2.8/intro.html#2 (besucht am 15. 08. 2023).

[29] Mozilla Foundation. „Proxy servers and tunneling.“ en. (n.a), Adresse: https://developer.mozilla.org/en-US/docs/Web/HTTP/Proxy_servers_and_tunneling (besucht am 18. 10. 2023).

[30] N. Ramirez. „How to Enable Health Checks in HAProxy (Guide).“ en. (Sep. 2021), Adresse: https://www.haproxy.com/blog/how-to-enable-health-checks-in-haproxy (besucht am 16. 09. 2023).

[31] „HAProxy Configuration Manual.“ en. (Sep. 2023), Adresse: https://docs.haproxy.org/2.8/configuration.html (besucht am 16. 09. 2023).

[32] R. Elmasri und S. Navathe, *Grundlagen von Datenbanksystemen*. Pearson Deutschland GmbH, 2009, ISBN: 9783863263348.

[33] MariaDB. „MariaDB Replication.“ en. (2023), Adresse: https://mariadb.com/kb/en/standard-replication/ (besucht am 08. 11. 2023).

[34] MariaDB. „Replication with Secure Connections.“ en. (2023), Adresse: https://mariadb.com/kb/en/replication-with-secure-connections/ (besucht am 08. 11. 2023).

[35] I. Ristic, *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. Feisty Duck, 2013.

[36] MariaDB. „Show Slave Status.“ en. (2023), Adresse: https://mariadb.com/kb/en/show-replica-status/%5D (besucht am 11. 11. 2023).

Listingverzeichnis

2.1 VIP-Konfiguration Keepalived 11
2.2 SOA-Record per dig abfragen 17
2.3 NS-Record per dig abfragen 18
2.4 A-Record per dig abfragen für Google NS 18
2.5 A-Record per dig abfragen für Hetzner NS 18
2.6 Bereitstellung einer VM mittels gcloud 21
2.7 Auflisten der Google Cloud VMs 22
2.8 Bereitstellung einer VM bei Hetzner 23
2.9 Auflisten der Hetzner Cloud VMs 23
2.10 Basiskonfiguration von Nftables 25
2.11 Basiskonfiguration von Fail2ban 26
2.12 Erstellung eines privaten Netzwerks bei Google 27
2.13 Erstellung eines privaten Netzwerks bei Hetzner 28
2.14 HAProxy Konfiguration 30
2.15 DNS-Watcher 32
2.16 Health-Check 33
2.17 nftables.conf des App-Servers 35
2.18 Ändern der User- und Gruppenzuweisung von html 35
2.19 Tranfser der Sessions zwischen den Anwendungsservern 36
2.20 Löschen alter Sessions 37
2.21 Installtion von MariaDB 40
2.22 Erstellung der CA mit openssl 40
2.23 Erstellung des db-one Zertifikats 41
2.24 Erstellung des db-two Zertifikats 41
2.25 Testen der Zertifikate 41
2.26 Erstellung eines Replication Users 42
2.27 50-server.conf von db-one 42
2.28 Konfiguration der Replikation auf db-one 43
2.29 Konfiguration der Replikation auf db-two 43
2.30 Testen der Replikation 44
3.1 Abfrage der Logindaten aus der Datenbank 47
3.2 Prüfen der Logindaten 48
3.3 Erstellen der Session 48
3.4 Öffnen der Datenbankverbindung 49
3.5 Validerung der Session 49
3.6 Statusabfrage an den Server 50
3.7 Automatisierter Login und Speichern der Session mit curl 51
3.8 Automatisiertes Erstellen eines Eintrages mit curl 51

3.9 Automatisiertes Erstellen vieler Einträges mit curl 52
3.10 Automatisiertes Abfragen aller Einträge mit curl 52
3.11 Rückgabe zur Abfrage aller Einträge mit curl (1) 52
3.12 Automatisiertes Ändern bestimmter Einträge mit curl 52
3.13 Rückgabe zur Abfrage aller Einträge mit curl (2) 53
3.14 Automatisiertes Löschen bestimmter Einträge mit curl 53
3.15 Rückgabe zur Abfrage aller Einträge mit curl (3) 53
4.1 NS-Record per dig abfragen (2) 56
4.2 Untersuchung der Netzwerklatenz 62
4.3 Untersuchung des Datendurchsatzes 63
4.4 Ausführung von 25 Insert-Statements pro Sekunde 63
4.5 Auswertung des Parameters Seconds_behind_master 64

Abbildungsverzeichnis

2.1 Einstiegspunkt der Infrastruktur mit Keepalived 12
2.2 Einstiegspunkt der Infrastruktur mit DNS 13
2.3 Datenbankreplikation zwischen zwei Primary-Nodes 38
2.4 Gesamtübersicht der aufgebauten Infrastruktur 45
4.1 Unterteilung der Infrastruktur in Layer 55
4.2 Zeitmessung der Änderung der DNS-Einträge 57
4.3 Aktualisierung des DNS-Eintrags nach Zeit 59
4.4 Umschwenken eines Applikationsservers 60
4.5 Übertragungsdauer der Sessions 61
4.6 CPU-Last des Datenbankservers 64
4.7 Umschwenken eines Datenbankservers 65

Tabellenverzeichnis

2.1 Auflistung aller VMs mit IP 24
4.1 Änderung des DNS-Eintrags nach Sekunden 58

Abkürzungsverzeichnis

API	Application Programming Interfaces	16
AWS	Amazon Web Services	5
Ajax	Asynchronous JavaScript and XML	46
BGP	Border Gateway Protocol	11
CA	Certificate Authority	40
CLI	command-line interface	16
CRUD	Create Read Update Delete	46
DHCP	Dynamic Host Configuration Protocol	27
DNS	Domain Name System	5
HTTP	Hypertext Transfer Protocol	12
ICMP	Internet Control Message Protocol	25
IO	Input Output	44
IP	Internet Protocol	9
IP	Internet Protocol	9
ISP	Internet Service Provider	14
NS	Nameserver	16
PID	Process identifier	10
REST	Representational State Transfer	22
SCP	Secure Copy	67
SOA	Start of Authority	16
SSH	Secure Shell Protocol	24
TCP	Transmission Control Protocol	11
TLD	Top-Level-Domain	17
TLS	Transport Layer Security	39
TTL	Time to live	14
VIP	Virtual IP-Address	10
VM	Virtual Machine	19
VPN	Virtual Private Network	12
VRRP	Virtual Router Redundancy Protocol	10
dig	Domain Information Groper	17