anexia

DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

# Route Injection

# Projekt T3_2000

für die Prüfung zum

Bachelor of Science

des Studienganges Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Leon Louis Schoch**

Abgabedatum 18. September 2023

| | |
|---|---|
| Bearbeitungszeitraum | 27 Wochen |
| Matrikelnummer | 1015290 |
| Kurs | TINF21B5 |
| Ausbildungsfirma | Anexia Deutschland GmbH<br>Karlsruhe |
| Betreuer der Ausbildungsfirma | Stephan Peijnik-Steinwender (B.Sc.) |
| Gutachter der Studienakademie | Prof. Dr. Markus Strand |

## Erklärung

Ich versichere hiermit, dass ich meine Projekt T3_2000 mit dem Thema: Route Injection selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Unterschrift

## Sperrvermerk

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung vom Dualen Partner vorliegt.

**Zusammenfassung**

Ein Distributed Denial of Service (DDoS)-Angriff kann eine starke Auslastung der betroffenen Systeme verursachen. Dies kann einen Absturz zur Folge haben, oder den Zugriff auf die Systeme verhindern. Um dieses Problem zu lösen wurde der Route Injection Service entwickelt, mit welchem ein Nutzer in der Lage ist, Netzwerkroute über Border Gateway Protocol (BGP)-Communities zu manipulieren. Ein DDoS-Angriff kann daher in ein Blackhole geroutet, und eine Belastung der Zielsysteme verhindert werden.

*A DDoS-Attack can cause a high load on the attacked systems. As a result, the systems might be inaccessible or crash. To solve this problem, we developed the route injection service, which enables a user to manipulate network routes via BGP-Communities. A DDoS-Attack can then be routed into a blackhole, and a strain on the target systems can be avoided.*

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Liste der Code Snippets

# Abkürzungsverzeichnis

| | | |
|---|---|---|
| **API** | Application Programming Interface | I |
| **DRF** | Django Rest Framework | 4 |
| **REST** | Representational State Transfer | 4 |
| **BGP** | Border Gateway Protocol | I |
| **DDoS** | Distributed Denial of Service | I |
| **TTL** | Time to live | 25 |
| **SQL** | Structured Query Language | 4 |
| **AS** | Autonome Systeme | 1 |
| **VM** | Virtuelle Maschine | 4 |
| **TCP** | Transmission Control Protocol | 6 |
| **RIP** | Routing Information Protocol | 6 |
| **OSPF** | Open Shortest Path First | 6 |
| **ASN** | Autonome System Nummer | 6 |
| **JSON** | JavaScript Object Notation | 11 |
| **SSH** | Secure Shell | 26 |
| **TTL** | Time to live | 25 |
| **DRY** | Don't repeat yourself | 31 |

# Kapitel 1

# Einleitung

Die zunehmende Abhängigkeit von digitalen Kommunikationsnetzwerken und die kontinuierliche Weiterentwicklung der globalen Infrastruktur haben zu einer signifikanten Steigerung des Datenverkehrs im Internet geführt. Während diese Fortschritte zahlreiche Vorteile für die Gesellschaft mit sich bringen, eröffnen sie auch neue Herausforderungen im Hinblick auf die Sicherheit und Stabilität des Netzwerkbetriebs. In diesem Zusammenhang gewinnt die Fähigkeit, den Datenverkehr effektiv zu leiten und gleichzeitig gegen potenzielle Bedrohungen zu schützen, zunehmend an Bedeutung. Das BGP, als das fundamentalste Routing-Protokoll im Internet, spielt eine kritische Rolle bei der Bestimmung der optimalen Routen für den Datenverkehr zwischen Autonomen Systemen (ASen). Allerdings hat die BGP-Protokollsuite bisher nur begrenzte Möglichkeiten zur gezielten Beeinflussung des Datenverkehrs in Ausnahmesituationen oder bei Sicherheitsvorfällen geboten. Eine solche Ausnahmesituation tritt beispielsweise auf, wenn ein Netzwerkressourcen-Engpass aufweist oder wenn bösartige Akteure versuchen, den Datenverkehr abzufangen oder zu manipulieren. Die vorliegende Forschung widmet sich daher der Entwicklung eines innovativen Ansatzes, der es ermöglicht, Internet-Routen über BGP gezielt in sogenannte „Blackholes“zu lenken. Dieses Konzept zielt darauf ab, den Datenverkehr von bestimmten Quellen oder zu bestimmten Zielen hinzuleiten, indem die betreffenden Routen im Netzwerk auf Blackholes abgebildet werden. Diese Blackholes repräsentieren Pfade im Netzwerk, die keinen tatsächlichen Datenaustausch ermöglichen, sondern den Verkehr effektiv abfangen und isolieren. Durch die Einrichtung dieser Blackholes wird eine maßgeschneiderte Methode zur Verteidigung gegen DDoS-Angriffe sowie zur effizienten Nutzung von Ressourcen in Überlastsituationen geschaffen. Die Motivation für dieses Projekt liegt darin,

die Flexibilität und die Sicherheitsaspekte von BGP-Routings zu erweitern, um den heutigen Anforderungen an die Netzwerksicherheit und -stabilität gerecht zu werden. Durch die Schaffung eines Mechanismus zur Blackhole-Routing kann das Risiko von Datenverkehrsumleitung durch bösartige Einflüsse minimiert und die Möglichkeit zur gezielten Netzwerkressourcenlenkung maximiert werden. Die Ergebnisse dieses Projekts haben das Potenzial, die bestehenden Ansätze zur Netzwerkverwaltung und -sicherheit zu erweitern und somit einen bedeutenden Beitrag zur Aufrechterhaltung der Integrität und Effizienz globaler Kommunikationsnetzwerke zu leisten.

# Kapitel 2

# Grundlagen

## 2.1 Einführung in die Problematik

Um im Falle eines DDoS Angriffs schnell reagieren zu können, muss es eine bequeme und einfache Möglichkeit geben, Routen zu manipulieren. Hierfür wurde das Projekt Remote Triggered Blackholing gestartet. Im Falle eines DDoS-Angriffs könnten somit IP Präfixe des Angreifers gezielt in ein Blackhole geroutet werden. Eine Belastung der Zielsysteme könnte somit verhindert werden, da die boshaften Pakete des Angreifers somit nicht beim Zielsystem ankommen würden, sondern in das schwarze Loch (Blackhole) weitergeleitet werden. Um die Routen in Routern manipulieren zu können, müssen diese über Injektoren in die Router injiziert werden. Im Verlaufe dieser Projektarbeit wird die Entwicklung der Injektoren Komponente und der Aufbau einer Staging(Testing) Umgebung genauer dargelegt. Der Aufbau und die Entwicklung der API Komponente wurde bereits zu einem großteil in der T1000 erläutert, jedoch wurde im Rahmen der T2000 diese um einen Delete-Endpunkt erweitert. [Schoch 2022]

## 2.2 Technologie Selektion

### 2.2.1 Django Rest Framework

„Django ist ein Web-Framework, dessen Ziel es ist, die Entwicklung von Web Applikationen schnell, einfach und übersichtlich zu machen. Das Django Representational State Transfer (REST) Framework, hier nachfolgend als Django Rest Framework (DRF) bezeichnet, ist ein REST Framework welches auf Django basiert. Mit DRF lassen sich REST-ful APIs schnell und einfach gestalten. Hierfür bietet Django eine Reihe an vorgefertigten Hilfestellung an, welche im Verlaufe dieser Projektarbeit näher erläutert werden. Datenbankmodelle werden hier einfach programmatisch deklariert und anschließend von Django automatisch verwaltet. Über Objekte können somit einzelne Werte aus der Datenbank entnommen werden, ohne sich mühsam mit Structured Query Language (SQL) Queries auseinandersetzen zu müssen. Sowohl Django als auch DRF basieren auf der Programmiersprache Python.“ [Vgl. Schoch 2022, S. 8]

### 2.2.2 Hashicorp Consul

„Consul, entwickelt von Hashicorp, ist eine Netzwerk Service Lösung, welche eine sichere Kommunikation zwischen Services und Applikation erlaubt. Consul kann sowohl redundant mit mehreren Nodes, als auch standalone genutzt werden. Für diese Projektarbeit, wird eine standalone Lösung eingesetzt und es wird lediglich die Key-Value Store Funktion genutzt. Mit dieser Funktion können Key-Value [...] Paare über das Netzwerk in Consul gespeichert werden.“ [Schoch 2022]

### 2.2.3 Docker

Docker ist Platform zur Containerisierung von Anwendungen. Hierdurch wird die Möglichkeit geschaffen eine isoliertes und leichtgewichtige Umgebung zu schaffen, welche sonst lediglich mittels Virtuellen Maschinen (VMs) möglich wäre. Durch Docker wird auf produktiven System durch die zusätzliche Isolationsschicht der Containerisierung eine weitere Sicherheitsstufe hinzugefügt, welche potenziellen Angreifern den Zugriff auf das Hostsystem erschwert.

### 2.2.4 Bird

Der Bird Internet Routing Daemon (Bird) ist eine Open-Source-Routing-Software, die als Router fungiert. Bird implementiert unter anderem BGP,

um Routing-Informationen zwischen Routern auszutauschen und optimale Routenentscheidungen zu treffen. Bird arbeitet neben anderen BGP-Routern, um BGP-Sessions aufzubauen, Routing-Updates auszutauschen und Routing-Informationen zu speichern. Bird kann BGP-Routen exportieren und an andere Router weitergeben, indem es BGP-'Update'-Messages verwendet und Exportregeln in seiner Konfigurationsdatei folgt. Diese Regeln definieren, welche Routen exportiert werden sollen und können durch Filter und Richtlinien gesteuert werden. Durch den Export von BGP-Routen ermöglicht Bird eine effiziente und zuverlässige Kommunikation und Weiterleitung in großen Netzwerken.

## 2.3 Stand der Technik

Das BGP ist ein Protokoll des Internet-Routings, das die *besten* Wege für den Datenverkehr zwischen ASen bestimmt. Im ursprünglichen Sinne war mit einem AS eine Organisation mit einem Standort gemeint, welche intern über ein internes routing Protokoll verfügte. Mit der Zeit hat sich die Bedeutung eines AS abgewandelt und eine Autonome System Nummer (ASN) kann von einer Organisation Standortübergreifend verwendet werden bzw. eine Organisation kann über mehrere ASNs verfügen. Es verwendet Peering-Verbindungen zwischen Routern, um Informationen über erreichbare Netzwerke auszutauschen und die optimalen Pfade für den Datenaustausch zu ermitteln. Anders als bei herkömmlichen Routing Protokollen wie dem Routing Information Protocol (RIP) oder Open Shortest Path First (OSPF), wird hier eine direkte Transmission Control Protocol (TCP) Verbindung zwischen Routern(Neighbours/Nachbarn) hergestellt. Eine weitere Unterscheidung besteht darin, dass es sich bei BGP um 'Policy'-basiertes Routing, im Vergleich zu 'Metrik' basierten Routing handelt. Konkret bedeutet dies, dass ein AS selbst bestimmen kann, wie Datenverkehr geroutet werden soll, sollte das AS über mindestens zwei Uplinks verfügen.

Wenn zwei BGP Nachbarn eine TCP Verbindung aufgebaut haben, beginnen diese BGP Informationen in Form von Nachrichten auszutauschen. Jede Nachricht besteht aus einem Header, und dem tatsächlichen Inhalt. [Vgl. Beijnum 2002, S. 19 f.] Um eine BGP Verbindung herzustellen, müssen sich Router über eine 'Open'-Message verbinden. Diese wird direkt nach dem Aufbau der TCP Verbindung ausgetauscht. [Vgl. Beijnum 2002, S. 20 f.]

| Version | My AS | Hold time | Identifier | Parlen | Optional parameters |
|---|---|---|---|---|---|
| 1 byte | 2 bytes | 2 bytes | 4 bytes | 1 byte | 0-255 bytes |

Tabelle 2.1: Aufbau der 'Open'-Message
Quelle: [RFC4271 Rekhter, Hares und Li 2006] in Anlehnung an [Beijnum 2002, S. 20]

Sollte die Open-Message erfolgreich vom Gegenstück angenommen worden sein, sendet dieser eine 'Keepalive'-Message zurück. Anschließend wird die BGP-Routentabelle über 'Update'-Messages ausgetauscht. [Vgl. Beijnum 2002, S. 20]

| UR length | Withdrawn routes | PA length | Path attributes | NLRI |
|---|---|---|---|---|
| 2 bytes | Variable | 2 bytes | Variable | Variable |

Tabelle 2.2: Aufbau der 'Update'-Message
Quelle: [RFC4271 Rekhter, Hares und Li 2006] in Anlehnung an [Beijnum 2002, S. 20]

Durch die 'Update'-Message werden die eigentlichen Informationen übertragen. Hierdurch können neue Routen hinzugefügt, oder alte Routen zurückgezogen werden. Ein nicht optionales Attribute ist der 'AS_PATH', welcher beschreibt, über welche AS bestimmte Präfixe zu erreichen sind.

BGP-Communities sind ein Mechanismus, mit welchem Netzwerkbetreiber spezifische Gruppen oder Kategorien von Präfixen markieren können. Diese Markierungen, als „Communities“ bezeichnet, können verwendet werden, um Routen zu identifizieren und zu beeinflussen, wie sie von anderen ASen interpretiert werden. Durch die Verwendung von Communities können Netzwerkbetreiber das Routing auf feinere Weise steuern und anpassen, ohne die Kernstruktur des BGP-Netzwerks zu verändern. Die Manipulation von Routen mittels BGP Communities erfolgt, indem einem bestimmten Präfix eine oder mehrere BGP-Communities zugewiesen werden. Andere AS können dann diese Community-Markierungen verwenden, um spezifische Aktionen auszuführen, wie z.B.:

- Pfadwahl beeinflussen: Durch das Zuweisen von Communities zu bestimmten Präfixen können Netzwerkbetreiber festlegen, wie andere AS ihre Routen interpretieren sollen. Dies kann dazu verwendet werden, den bevorzugten Weg für den Datenverkehr zu beeinflussen.

- Traffic-Engineering: Netzwerkbetreiber können Communities verwenden, um den Datenverkehrsfluss zu steuern. Durch Markieren von Präfixen können sie bestimmte AS dazu anleiten, den Datenverkehr auf bestimmten Wegen zu leiten, um Netzwerkressourcen effizienter zu nutzen.

- Blackhole-Routing: BGP Communities können dazu genutzt werden, bestimmte Präfixe zu markieren und den Datenverkehr über Blackholes zu lenken, um Angriffe oder Überlastungen zu bewältigen. Spezielle für Blackholing wurde eine eigene Community definiert: `65535:666` [Vgl. King u. a. 2016]

- Routenfilterung: AS können Community-Markierungen verwenden, um präzise Routenfilterung durchzuführen. Damit können sie bestimmte Routen von bestimmten Quellen oder für bestimmte Zwecke filtern oder akzeptieren.

Die Verwendung von BGP Communities ermöglicht eine flexiblere und zielgerichtete Steuerung des Internet-Routings. Netzwerkbetreiber können so gezielt auf unterschiedliche Anforderungen reagieren und gleichzeitig die Integrität und Stabilität des BGP-Netzwerks aufrechterhalten.

# Kapitel 3

# Architekur

Die Architektur des Route Injection Service besteht aus drei wesentlichen Bestandteilen, welche entweder direkt verbunden sind oder mittels Hashicorp Consul Daten austauschen können.

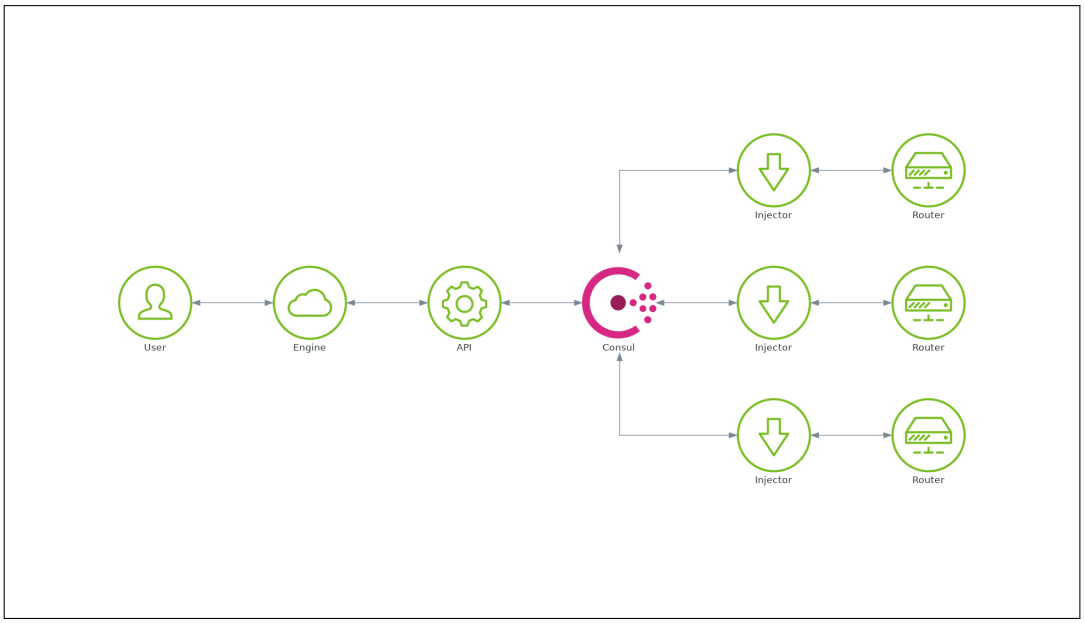


Abbildung 3.1: Route Injection Architektur
Quelle: Firmenintern

## 3.1 API

Die API ist dafür verantwortlich die Eingaben des Users, welche über die Engine übermittelt wurden zu überprüfen und zu validieren. Sind die Eingaben nicht korrekt, so gibt die API eine entsprechende Fehlermeldung zurück. In der Zukunft wird die API auch dafür verantwortlich sein entsprechende Monitoring Endpunkte zur Verfügung zu stellen, sodass der allgemeine Status des Service überwacht werden kann.

## 3.2 Hashicorp Consul

Hashicorp Consul, im weiteren Verlauf nur ‘Consul‘ genannt, wird als Zwischenspeicher für Routen und deren injizierte BGP-Communities verwendet. Des Weiteren können Injectoren hier Ihren ‘Heartbeat‘ abspeichern.

## 3.3 Injector

Der Injector bezieht periodisch(alle 5 Sekunden) die in Consul gespeicherten Routen. Sollte es hier eine Änderung gegeben haben, wird eine Konfigurationsdatei für den Bird Routingdaemon neu erstellt. Anschließend wird über das ’Bird Controlsocket’ der Befehl zum Neuladen der Konfiguration gegeben.

## 3.4 Router

Als Router wird der Bird Routingdaemon eingesetzt. Dieser stellt eine BGP-Session mit einem physischen Router her, welcher die von Bird zu Verfügung gestellten Router importiert und innerhalb des BGP-Netzwerks weitergibt.

# Kapitel 4

# API Komponente

## 4.1 Aufgaben

Die API ist die Schnittstelle des Service und außen stehenden Technologien wie der Anexia Engine. Ihre Hauptaufgabe besteht darin, eine strukturierte Interaktionsmöglichkeit zu bieten, die es internen Benutzern über Systeme wie der Anexia Engine ermöglicht, BGP-Routen mit zugehörigen BGP-Communities in das Netzwerk zu injizieren. Dies geschieht durch die Annahme von JavaScript Object Notation (JSON)-Anfragen, die spezifische Informationen enthalten, nämlich IPv4- oder IPv6-Präfixe und die entsprechenden BGP-Communities. Die API führt eine umfassende Validierung der eingehenden Daten durch, um sicherzustellen, dass die bereitgestellten Informationen korrekt und im erwarteten Format vorliegen. Diese Validierung umfasst die Überprüfung der Richtigkeit der IP-Adressbereiche sowie die syntaktische Korrektheit der zugeordneten BGP-Communities. Durch diesen Schritt wird gewährleistet, dass nur gültige Informationen in das System eingebracht werden. Die validierten Daten werden anschließend an Consul, über dessen eigene API übermittelt. Die Daten werden so abgelegt, dass der Injector einen erleichterten Zugriff hat.

## 4.2 Umsetzung

Da die Konzeption und Implementierung der API schon umfassend in der Projektarbeit T1000 erläutert wurde, wird auf eine Wiederholung dessen verzichtet. In diesem Bericht wird lediglich die Implementierung des ‘Delete‘-Endpunkts dargestellt, da dieser aus zeitlichen Gründen nicht mehr in den ersten beiden Praxisphase implementiert werden konnte, jedoch ein Grundbestandteil des entwickelten Service ist.

Die Implementierung eines ‘Delete‘-Endpunkts in der API, mittels des Django Rest Frameworks, ermöglicht das Löschen von Routen aus dem System.

```
class BaseRouteViewSet(
    CreateModelMixin,
    ReadOnlyModelViewSet,
    BaseRequestViewSet,
):
    @action(detail=False, url_path=r"([A-Za-z-_/]*)status/(?P<task_info_id>[0-9a-z-]+)")
    def status(self, request, task_info_id):
        route_object = get_object_or_404(
            self.serializer_class.Meta.model, task_info_id=task_info_id
        )
        propagate_status(route_object)
        return super().status(request, task_info_id)
```

Code Snippet 4.1: BaseRouteViewset Klasse

Der in Snippet 4.1 gezeigte Code stellt eine Mutterklasse dar, von welcher sowohl der ‘Create‘, als auch ‘Delete‘-Endpunkt erben. Durch diese Klasse wird die Möglichkeit gegeben, von der Anexia Engine erwartete Endpunkte einfach zu implementieren, ohne dass sich ein Entwickler mit den Feinheiten dessen auseinandersetzen muss. Da hier die `CreateModelMixin` Klasse geerbt wird, stellt sich das DRF automatisch ein ’POST’-Requests für diesen Endpunkt zu akzeptieren.

```
class DeleteRouteViewSet(BaseRouteViewSet):
    queryset = DeleteRoute.objects.all()
    serializer_class = DeleteRouteSerializer

    def perform_create(self, serializer):
        super().perform_create(serializer)
        delete_route(serializer.instance)
```

Code Snippet 4.2: DeleteRouteViewset Klasse

Die tatsächliche Implementierung fällt durch das Erben von der 'BaseRouteViewSet' Mutterklasse sehr simpel aus. Durch das Überschreiben der `perform_create` Methode, welche vom DRF zur Verfügung gestellt wird, kann diese als Hook benutzt werden um eigenen Code ausführen zu lassen. Mit der Super Methode wird sichergestellt, dass die nicht überschriebene Ursprungsmethode von `perform_create` ausgeführt wird. Das DRF erstellt dann einen Datenbankeintrag mit den vom Nutzer eingegeben Werten. Vor dem Ende des Kontextes der Methode wird noch eine weitere Methode `delete_route` aufgerufen.

```
def delete_route(instance):
    consul_instance = prepare_consul(os.getenv("CONSUL_HOST"),
    os.getenv("CONSUL_PORT"))
    prefix = str(instance.prefix)
    prefix_encoding = get_prefix_encoding(prefix)
    consul_instance.kv.delete(
        f'v1/route/global/{prefix_encoding}/{prefix.replace
    ("/", "_")}'
    )
    update_active_injectors(instance)
```

Code Snippet 4.3: delete_route Methode

Hier findet nun das eigentliche Übermitteln der Daten an Consul statt.

# Kapitel 5

# Injector Komponente

## 5.1 Aufgaben

Der Injector ist der zentrale Baustein des Route Injection Service, der die Möglichkeit bietet, mittels BGP Communities, Routen in das Netzwerk zu injizieren. Der Injector erfüllt dabei eine Reihe von wesentlichen Aufgaben:

Zuallererst ist der Injector für die Konvertierung der von der API empfangenen Routen in eine für den Router (Bird) verständliche Konfigurationsdatei verantwortlich. Diese Konvertierung ist von entscheidender Bedeutung, um die Weiterleitung der Routen an den Router in einem kompatiblen Format sicherzustellen. Während die Validierung der Präfixe und Communities von der API Komponente übernommen wird, hat der Injector eine eigene Validierung für Routen, welchen über den Emergency-Mode angegeben werden, da hier die API Komponente überbrückt wird. Bei auftretenden Konflikten oder Unstimmigkeiten kann der Injector angemessene Maßnahmen ergreifen, um die Integrität der anderen Komponenten und schlussendlich des Netzwerks, zu gewährleisten. Ein wichtiger Aspekt ist auch die aktive Kommunikation des Injectors mit dem Router (Bird). Diese Kommunikation erfolgt, um die generierten Konfigurationsänderungen effektiv zu übertragen und sicherzustellen, dass die injizierten Routen nahtlos in das Routing-Protokoll des Routers integriert werden. Schließlich stellt der Injector durch präzises loggen sicher, dass im Falle eines Fehlers, oder im schlimmsten Fall, bei einem Absturz der Komponente, Ereignisse festgehalten werden. Zusammenfassend fungiert der Injector als entscheidende Schnittstelle, die die Funktionen der API und des Routers miteinander verbindet. Mit seiner intelligenten Konvertierung und Verwaltung

von Routen durch BGP Communities gewährleistet er, dass die gewünschten Routing-Änderungen präzise und effizient im BGP-Netzwerk implementiert werden.

## 5.2 Umsetzung

### 5.2.1 Generieren der Config Files für Bird

Für den einfachen Umgang mit Routen wurde für die `routes` Variable eine Python Dataclass angelegt, welche das IP-Präfix, die IP-Version und eine Liste der gesetzen BGP-Communities enthält.

```
@dataclass
class Route:
    prefix: str
    encode: str
    communities: list[str]

    def __init__(self, prefix="", encode="", communities=[]):
        self.prefix = prefix
        self.encode = encode
        self.communities = communities

    def __str__(self):
        return f"{self.encode} {self.prefix} {self.communities}
    "

    def decode_prefix(self):
        self.prefix = self.prefix.replace("_", "/")

    def encode_prefix(self):
        self.prefix = self.prefix.replace("/", "_")

    def get_communities(self):
        communities = self.communities
        return list(map(lambda com: com.replace(":", ","),
    communities))
```

Code Snippet 5.1: Python Dataclass für Routen Objekte

Neben den genannten Feldern, kann die Dataclass auch noch Methoden zur verarbeitung der Felder, ähnlich wie eine normale Klasse beinhalten. Besonders zu betonen ist hier die `get_communities` Methode, welche die Communities in ein von Bird akzeptiertes Format umwandelt.

Die in Consul gespeicherten Routen werden dann periodisch von Consul über dessen eigene API abgefragt. Hierfür ist die `parse_consul_values` Methode zuständig.

```
def parse_consul_values(values, watched_prefix) -> (list[Route
    ], list[Route]):
     if not values:
         return [], []

     v4_routes = []
     v6_routes = []

     for entry in values:
         route_entry = Route()
         route_entry.prefix = entry["Key"].split(watched_prefix)
    [1]
         json_communities = entry["Value"].decode("utf-8")
         route_entry.communities = json.loads(json_communities)[
    "communities"]

         route_entry.encode, route_entry.prefix = route_entry.
    prefix.split("/")
         route_entry.decode_prefix()

         if route_entry.encode == "IPv4":
             v4_routes.append(route_entry)

         elif route_entry.encode == "IPv6":
             v6_routes.append(route_entry)
     return v4_routes, v6_routes
```

Code Snippet 5.2: parse_consul_values Methode

Im Laufe der Methode werden die abgefragten Einträge in Routenobjekte umgewandelt. Um herauszufinden um welche IP Version es sich bei der Route handelt, wird der entsprechende Key des Pfades ausgelesen, da dieser die IP Version mit im Namen trägt. Als Resultat gibt die Methode ein Tupel zurück, wobei eines die IPv4 Routen und das andere die IPv6 Routen sind. Diese Aufteilung ist notwendig, da die in Debian 11 mitgelieferte Version von Bird eine klare Auftrennung dieser fordert. Neuere Versionen von Bird können auch mit beiden IP Versionen gleichzeitig umgehen.

Durch die Auftrennung der beiden IP-Versionen, muss auch die Konfigurationsdatei für Bird, zweimal generiert werden. Mit der Methode `generate_bird_files`

werden Umgebungsvariablen geladen, welche für die Generierung der Konfiguration benötigt werden. Neben diesen übergibt die Methode auch die Routen als Parameter weiter.

```
def generate_bird_files(v4_routes, v6_routes, pybird, pybird6):
    gen = BirdConfigGenerator()
    click.echo("Generating and committing config files")
    gen.bird_config(
        v4_routes,
        "route_template.j2",
        "../config/bird",
        "v4.conf",
        os.getenv("ROUTER_ID"),
        os.getenv("LOCAL_AS"),
        os.getenv("REMOTE_AS"),
        os.getenv("BGP_NEIGHBOR"),
        "ANEXIA Route Injection v4",
    )

    gen.bird_config(
        v6_routes,
        "route_template.j2",
        "../config/bird",
        "v6.conf",
        os.getenv("ROUTER_IDv6"),
        os.getenv("LOCAL_AS"),
        os.getenv("REMOTE_AS"),
        os.getenv("BGP_NEIGHBORv6"),
        "ANEXIA Route Injection v6",
    )

    pybird.commit_config()
    pybird6.commit_config()
    click.echo("Done")
```

Code Snippet 5.3: generate_bird_files Methode

Die Methode `bird_config`, welche um eine gute Struktur zu wahren zu einer gesonderten Datei und Klasse angehört, ruft die `get_communities` Methode der Route Dataclass auf, um die Routen in einer von Bird lesbares Format zu wandeln. Des Weiteren wird hier die tatsächliche Konfiguration auf das Dateisystem geschrieben.

```
def bird_config(
    self,
    unconverted_routes,
    route_template,
    config_path,
    config_name,
    router_id=None,
    local_as=None,
    remote_as=None,
    bgb_neighbor=None,
    description=None,
):
    converted_routes = []
    for route in unconverted_routes:
        route.communities = route.get_communities()
        converted_routes.append(route)

    target_file = self.__prepare_config_path(config_path,
    config_name)
    self.__write_config_file(
        target_file,
        converted_routes,
        route_template,
        router_id,
        local_as,
        remote_as,
        bgb_neighbor,
        description,
    )
```

Code Snippet 5.4: bird_config Methode

Um die Routen an den Bird Routing Daemon übermitteln zu können, müssen diese erst in eine für Bird verständliche Konfigurationsdatei umgewandelt werden. Zur Realisierung wird die Jinja2 Templating Engine verwendet, da diese die Möglichkeit schafft, alle Eigenschaften des Injectors dynamisch zu konfigurieren. Somit kann der tatsächliche Code aller Injectoren identisch sein, und Variable Eigenschaften wie z.B. die `router_id` oder der BGP-Peering

Nachbar können beim Ausrollen festgelegt werden.

```
protocol static injected_routes {
    {% for route in routes %}
        route {{route.prefix}} via {{router_id}} {
        {% for community in route.communities %}
        bgp_community.add(({{community}}));
        {% endfor %}
        };
    {% endfor %}
}

protocol bgp Route_Injection {
description "{{description}}";
    local as {{local_as}};
    neighbor {{bgp_neighbor}} as {{remote_as}};
next hop self;
export filter {
    if proto = "injected_routes" then accept;
    reject;
    };
}
```

Code Snippet 5.5: Jinja Template zur Konfigurationsgenerierung

In diesem Template finden sich einige Variablen

- `routes` (Python Liste mit Routen Elementen)
- `router_id` (IPv4 Addresse des Injectors)
- `local_as` (Lokales ASN)
- `remote_as` (Nachbar ASN)
- `bgp_neighbor` (Nachbar BGP-Router IPv4 Adresse)
- `description` (Beschreibung des Protokolls)

wieder, welche entweder im Code oder dynamisch beim Ausrollen, also ausrollen des Injectors gesetzt werden müssen. Jinja kann auch mit Listen und verschachtelten Listen umgehen, was bei der `routes` Variable zum Einsatz kommt. Jinja kann dann über die Liste der Routenobjekte iterieren und für jede Route einen gesonderten Eintrag mit den jeweiligen BGP-Communities erstellen. Folglich ein Beispiel einer möglichen Konfiguration. ASNs und `router_id`s

können hier entweder als Umgebungsvariable oder von einer .env Datei geladen werden. Die Routen werden dynamisch während der Programmlaufzeit angegeben, konvertiert und konfiguriert.

```
protocol static injected_routes {
        route 1.1.1.1/32 via 172.20.0.5 {
        bgp_community.add((47147,3200));
        bgp_community.add((12345,12345));
        };
}

protocol bgp Route_Injection {
description "ANEXIA Route Injection v4";
    local as 64701;
    neighbor 172.20.0.6 as 65001;
next hop self;
export filter {
    if proto = "injected_routes" then accept;
    reject;
    };
```

Code Snippet 5.6: Beispiel einer Konfigurationsdatei

Somit können nun Konfigurationsdateien für Bird erstellt werden. Sollte jedoch während dem Rendern des Templates ein Fehler auftreten, kann es passieren, dass eine inkorrekte oder gar keine Konfigurationsdatei generiert wird. Dies könnte einen negativen Einfluss auf die Operation des Service haben und muss somit verhindert werden.

Um das Problem zu verhindern, wird die Konfiguration erst in eine temporäre Datei geschrieben. Wenn dies erfolgreich war, dann wird die temporäre Datei umbenannt und in das echte Konfigurationsverzeichnis geschoben. Da hier eine Datei überschrieben statt angepasst wird, gehen Dateiberechtigungen verloren und müssen neu gesetzt werden.

```
def __write_config_file(
    self,
    target_path,
    routes,
    template,
    router_id=None,
    local_as=None,
    remote_as=None,
    bgp_neighbor=None,
    description=None,
):
    template = self.env.get_template(template)
    with NamedTemporaryFile(delete=False, mode="w") as conf:
        conf.write(
            template.render(
                routes=routes,
                router_id=router_id,
                local_as=local_as,
                remote_as=remote_as,
                bgp_neighbor=bgp_neighbor,
                description=description,
            )
        )
    try:
        os.chmod(conf.name, 0o660)
        shutil.move(conf.name, target_path)
    except Exception as e:
        echo(e)
        os.remove(conf.name)
```

Code Snippet 5.7: ___write_config_file Methode

### 5.2.2 Status der Routen von Bird abfragen

#### Evaluation der pybird Bibliothek

Da die entwickelte API über einen Status Endpunkt verfügt, welcher letztendlich von der Anexia Engine abgerufen wird, muss auch der Injektor die benötigten Statusinformationen zur Verfügung stellen. Hierfür wurde evaluiert, welche Python Bibliothek sich am besten zu diesem Zwecke eignet.

Die Entscheidung für die Verwendung der ‘pybird‘ Bibliothek wurde aus folgenden Gründen getroffen:

1. Funktionalität: Die ‘pybird‘ Bibliothek wurde speziell dafür entwickelt mit dem Bird Routing Daemon zu interagieren.

2. Direkte Socket Anbindung: ‘pybird‘ unterstützt die direkte Kommunikation mit dem Bird Control Socket, was eine erleichterte Kommunikation ermöglicht.

3. Aktualisierung und Wartung: Da die ‘pybird‘ Bibliothek aktiv gepflegt wird, kann sichergestellt werden, dass sie auch mit zukünftigen Versionen des Bird Routing Daemons kompatibel sein wird. Des Weiteren kann so auch sichergestellt werden, dass das Route Injection Project sich auch in der Zukunft noch auf diese Bibliothek verlassen kann.

4. Open-Source: Durch den offenen Quellcode, kann sichergestellt werden, dass der Code keine Malware/Spyware enthält. Sollte es nötig sein, kann der Quellcode der Bibliothek geforked, und auf die Bedürfnisse der Anexia angepasst werden.

Über die Methode `get_routes` der `PyBird` Klasse können die von Bird übernommenen Routen abgefragt werden. Als Parameter kann das Präfix der Route angegeben werden, sodass die Ausgabe auf nur dieses Präfix beschränkt wird. Pybird gibt die Ausgabe dann in folgendem Format zurück:

```
[{'community': '65535:65281',
  'prefix': '1.2.3.4/32',
  'peer': '172.20.0.3',
  'interface': 'eth0',
  'source': 'injected_routes',
  'time': '13:37:47'}]
```

Code Snippet 5.8: Unverarbeitete Ausgabe von Pybird

Von dieser Ausgabe wird jedoch nur der Teil, welcher die Communities betrifft benötigt. Folglich muss die Ausgabe noch im Code angepasst werden.

```
def respond_state_to_consul(
    consul: ckv, pybird: PyBird, route: Route, injector_id: str
) -> None:
    state = pybird.get_routes(prefix=route.prefix)
    route.encode_prefix()
    try:
        actual_communities = state[0].get("community", "").
    split(" ")
    except IndexError:
        actual_communities = []
    expected_communities = list(route.communities)
    state = get_bird_communities(expected_communities,
    actual_communities)
    state = json.dumps({"communities": state})
    try:
        consul.kv.put(
            f"v1/state/{injector_id}/{route.encode}/{route.
    prefix}",
            state,
        )
    except requests.exceptions.ConnectionError:
        click.echo("Lost consul while reporting route :c")
        return
    route.decode_prefix()
    click.echo(f"Route {route.prefix} with state {
    actual_communities} pushed to consul")
```

Code Snippet 5.9: respond_state_to_consul Methode

Um den Status zu bestimmen, werden die Communities, welche im Routenobjekt abgespeichert sind, mit den Communities welche von Bird zurückgegeben wurde verglichen. Stimmen diese überein, so kann davon ausgegangen werden, dass Bird alle Communities akzeptiert hat und an den Nachbar Router übermitteln kann. Sollte es Abweichungen zwischen den Communities geben, bedeutet dies, dass noch nicht alle Communities von Bird akzeptiert wurden. Als Folge dessen werden auch nur die aktuell in Bird eingetragen Communities zurück an Consul übermittelt. Die API-Komponente des Route Injection Service fragt dann den in Consul eingetragenen Status ab und bestimmt dann selbst, ob der gesamte Prozess erfolgreich, noch im Gange oder fehlerhaft war. Dies wird dann von der Anexia Engine interpretiert und ist für den Nutzer sichtbar.

### 5.2.3 Realisierung des Heartbeats

Um sicherzustellen, dass die API den aktuellen Status der online verfügbaren Injektoren erfassen kann, verwenden die Injektoren ein sogenanntes ‘Heartbeat‘-System, das seine Aktivität in Consul signalisiert. Dieses Heartbeat wird in Form eines Wertes (Value) in Consul gemeldet. Dieser Prozess ermöglicht es der API, den Zustand der einzelnen Injektoren zu überwachen und sicherzustellen, dass sie ordnungsgemäß funktionieren.

Jeder Injektor meldet seinen Status durch das Schreiben eines Wertes (Value) in einen spezifischen Schlüssel-Wert-Pfad in Consul. Dieser Pfad lautet: `v1/state/<injector_id>/heartbeat`. Hierbei steht `<injector_id>` für die eindeutige Kennung des Injektors. Der Wert (Value), der in den oben genannten Schlüssel-Wert-Pfad geschrieben wird, hat den Inhalt ‘{}‘, was auf ein leeres JSON-Objekt hinweist. Dieses leere Objekt dient als Platzhalter und signalisiert der API, dass der Injektor aktiv ist und seinen Heartbeat meldet. Der gemeldete Wert (Value) hat eine Time to live (TTL) von 10 Sekunden. Dies bedeutet, dass nachdem der Injektor seinen Heartbeat gemeldet hat, der Wert für 10 Sekunden in Consul bestehen bleibt. Wenn innerhalb dieses Zeitraums keine weiteren Heartbeats gemeldet werden, wird der Wert automatisch aus Consul entfernt.

Durch das Heartbeat-System kann die API regelmäßig aktualisierte Informationen erhalten, welche Injektoren online und funktionsfähig sind.

Um den Heartbeat im Programmcode möglichst modular zu realisieren wurde hierfür eine eigene Methode erstellt.

```
1 def create_heartbeat(consul, injector_id):
2     session_id = consul.session.create(behavior="delete", ttl
    =10)
3     consul.kv.put(
4         key=f"v1/state/{injector_id}/heartbeat", value="{}",
    acquire=session_id
5     )
6     return session_id
```

Code Snippet 5.10: create_heartbeat Methode

Nach dem initialen Anlegen des Heartbeateintrages wird dieser alle fünf Sekunden erneuert und sicherzustellen, dass die TTL des Eintrages nicht abläuft.

### 5.2.4 Emergency-Mode

Um sicherzustellen, dass der Route Injection Service auch in Szenarien von Netzwerkproblemen zwischen der API und den Injektoren effizient arbeiten kann, sei es für das Hinzufügen, Ändern oder Löschen von Routen, wurde eine maßgebliche Funktion eingeführt, die als Emergency-Mode, bzw. Notfallmodus bekannt ist. Diese Funktion wurde entwickelt, um direkten Zugriff auf die Injektoren zu ermöglichen und Routenverwaltungsvorgänge über die Kommandozeile durchzuführen. Der Emergency-Mode fungiert als eine Art Sicherheitsvorkehrung, die sicherstellt, dass die Verfügbarkeit und Funktionalität des Dienstes aufrechterhalten werden kann, selbst wenn die übliche Kommunikation zwischen der API und den Injektoren temporär gestört ist. Der Namensteil 'Mode' lässt vermuten, dass es sich um einen tatsächlichen Operationsmodus handelt. Dies ist allerdings nicht ganz korrekt. Der Emergency-Mode ist eher als Funktionalitätserweiterung zu sehen und kann selbst dann aktiviert werden, wenn die Kommunikation zwischen API und Injector intakt ist. Dies ist insbesondere nützlich, wenn dringende Änderungen an den Routingeinstellungen erforderlich sind, die nicht auf die normale Kommunikation warten können. Um Zugriff auf die Kommandozeile zu erhalten, muss ein Nutzer sich über Secure Shell (SSH) auf den Injector einloggen. Firmeninterne Automatismen stellen sicher, dass nur befugte Nutzer Zugriff auf das System haben.

Zur Gewährleistung der Integrität des Service müssen die vom Nutzer eingegebene Routen validiert und auf Ihre Korrektheit überprüft werden. In der Regel wird dies von der API übernommen, jedoch werden im Notfallmodus die Routen direkt in den Injector eingespeist, und die Validierung der API wird umgangen. Daher muss diese vom Injector selbst durchgeführt werden.

```
def validate_route(prefix: str, communities=None) -> Route:
    route = Route()
    try:
        route.prefix = str(ipaddress.ip_network(prefix))
    except ValueError:
        raise click.exceptions.BadParameter("Route prefix is
    invalid")
    if ":" in route.prefix:
        route.encode = "IPv6"
    else:
        route.encode = "IPv4"
    if communities:
        communities = communities.split(",")
        for community in communities:
            community_parts = community.split(":")
            if len(community_parts) != 2:
                raise click.exceptions.BadParameter(
                    f"{community} is not a valid BGP community"
                )
            try:
                if not int(community_parts[0]) in range(1,
    65535) or not int(
                    community_parts[1]
                ) in range(1, 65535):
                    raise click.exceptions.BadParameter(
                        f"{community} is not a valid BGP
    community"
                    )
            except ValueError:
                raise click.exceptions.BadParameter(
                    f"{community} contains invalid integer
    value"
                )
        route.communities = list(communities)
    return route
```

Code Snippet 5.11: Methode zur Validierung von Routen

Der Zweck ist, BGP-Routen, primär in Bezug auf deren Präfixe und Communities zu validieren. Die Methode akzeptiert ein Präfix als obligatorisches Argument und optional eine Liste von Communities als Zeichenfolge. Das Hauptziel dieser Funktion ist es, sicherzustellen, dass die angegebenen Informationen den BGP-Anforderungen entsprechen und gültig sind.

Zuerst wird ein neues Routenobjekt erstellt, das als Container für die validierten Daten dient. Die Funktion versucht dann, den übergebenen Präfix als IP-Netzwerk zu interpretieren. Bei einer ungültigen Eingabe wird eine ‘BadParameter‘-Exception ausgelöst.

Das Präfix wird analysiert, um festzustellen, ob es sich um ein IPv4- oder IPv6-Präfix handelt. Dies wird im ‘encode‘-Attribut des Routenobjekts vermerkt. Im Fall von übergebenen Communities werden diese analysiert und validiert. Jede Community wird auf ihre Struktur überprüft, und die einzelnen Teile werden auf ihre Gültigkeit im Hinblick auf ASN und Wertigkeit geprüft. Fehlerhafte Communities führen zu entsprechenden ‘BadParameter‘-Exceptions.

Abschließend werden die validierten Informationen, einschließlich Präfix und Communities, im Routenobjekt gespeichert. Die Funktion gibt dieses Objekt zurück, das nun die validierten Daten enthält.

Zur Vereinfachung der Interaktionen mit der Kommandozeile wird die Bibliothek ‘click‘ verwendet. Durch diese können Exceptions leicht an den Benutzer übermittelt werden, und Tests können einfach gestaltet werden.

Der Operator welcher letztendlich den Emergency Mode bedienen wird, hat zwei Eingabemöglichkeiten:

- add-route <prefix> <communities>
- delete-route <prefix>

Wobei ‘<>‘ für Platzhalter des entsprechenden Parameters stehen. Eine Möglichkeit, schon existierende Routen zu updaten bietet der Emergency Mode nicht. Routen welche über den Emergency Mode hinzugefügt wurde, haben immer Vorrang gegenüber Routen, welche über Consul geladen wurden. Eine weitere Anforderung an den Emergency Mode war, dass Routen auch nach Reboot des Injectors erhalten bleiben. Dies forderte, dass Routen auf einer Weise im Dateisystem erhalten werden. Um dies zu Realisieren bestünde die Möglichkeit eine Datenbank wie ‘sqlite‘ zu nutzen. Eine einfachere Lösung dieses Problems war es jedoch, die Routen als JSON in eine Datei zu schreiben. Die schon bei den Konfigurationsdateien für Bird, wurden IPv4 und IPv6 aus demselben Grund getrennt.

Um die Konsistenz und Integrität dieser Dateien, auch ‘Emergency Files‘ genannt zu gewährleisten, wurde ein Filelock gesetzt. Zur Vermeidung des Dirty read Problems, welches in der Vorlesung Datenbanken erläutert wurde, wurde das Filelock sowohl für Schreib- als auch für Lesevorgänge gesetzt. So

kann ein zweiter Prozess das Emergency File erst lesen, wenn der erste Prozess den Schreibvorgang abgeschlossen hat. Dies dient nicht nur zur Mehrbenutzersynchronisation von mehreren Menschen, sondern hauptsächlich, dass der Hauptprozess nicht versucht das Emergency File zu lesen, während ein Operator mittels des Emergency Mode Änderungen vornimmt.

Da Python, beziehungsweise die benutze JSON Bibliothek Probleme damit hatte verschachtelte JSONs zu de- und enkodieren, wurde eine weitere Dataclass angelegt. Diese Dataclass dient nur als Container, um eine Liste an Routenobjekten anzulegen.

```
@dataclass_json
@dataclass
class RouteContainer:
    routes: list[Route]

    def __init__(self, routes):
        self.routes = routes

    def __str__(self):
        return f"{self.routes}"
```

Code Snippet 5.12: Route Container Dataclass

Zu Beginn des Programmstarts werden die Pfade der Lockfiles und Emergency Files innerhalb des Docker Containers festgelegt:

```
emergency_file_v4 = "/var/lib/route_injector/emergency_route_v4.json"
emergency_file_v6 = "/var/lib/route_injector/emergency_route_v6.json"
lock_file_v4 = "/var/lib/route_injector/emergency_route_v4.lock"
lock_file_v6 = "/var/lib/route_injector/emergency_route_v6.lock"
```

Code Snippet 5.13: Deklaration der Dateipfade

Die Methode welche beim Aufruf von `add-route` über die Kommandozeile aufgerufen wird lässt sich wie folgt darstellen:

```
@click.argument("communities")
@click.argument("prefix")
@cli.command()
def add_route(prefix, communities):
    route = validate_route(prefix, communities)
    emergency_file = emergency_file_v4
    lockfile = FileLock(lock_file_v4)
    if route.encode == "IPv6":
        emergency_file = emergency_file_v6
        lockfile = FileLock(lock_file_v6)
    with lockfile.acquire():
        current_routes = read_emergency_file(emergency_file)
        new_routes = find_and_remove_in_list(current_routes,
    route)
        new_routes.append(route)
        route_container = RouteContainer(new_routes)
        write_emergency_file(route_container, emergency_file)
```

Code Snippet 5.14: add_route Methode

Zuerst wird über den erwähnten Validierungsprozess sichergestellt, dass die vom Nutzer eingegeben Route eine valide Route ist. Über eine if Abfrage wird geprüft, ob das Präfix der eingegebenen Route ein IPv6 Präfix ist. Ist das der Fall, dann wird das entsprechende Emergency File und Lockfile einer Variablen zugewiesen. Anschließend wird das Filelock auf das entsprechende Emergency File gesetzt, um sicherzustellen, dass keine weiteren Prozesse auf das File zugreifen können. Im Folgenden werden die schon im Emergency File enthaltenen Routen mit den neu hinzugefügten verglichen. Sollte eine Route hinzugefügt werden, wessen Präfix schon im aktuellen Emergency File enthalten ist, wird diese über die `find_and_remove_in_list` entfernt.

```
def find_and_remove_in_list(route_list: list, list_element:
    Route):
    for element in route_list:
        if element.prefix == list_element.prefix:
            route_list.remove(element)
    return route_list
```

Code Snippet 5.15: find_and_remove_in_list Methode

Die neue Route wird danach der Liste von Routen hinzugefügt, und über die RouteContainer Dataclass wieder zu einer verschachtelten JSON konvertiert.

Zum Löschen von Routen aus den Emergency Files, gibt es die `delete_route` Methode, welche sich maßgeblich dadurch unterscheidet, dass sie keine BGP-Communities als Parameter benötigt, sondern lediglich das Routenpräfix. Infolgedessen, fehlt in dieser Methode auch der Teil, welcher die neue Route der Routenliste hinzufügt, da hier nur die Route entfernt werden muss.

Da das Lesen und Schreiben der Files mehrmals im Programmcode geschieht, wurde hierfür jeweils eine Methode geschrieben um Codeduplizierung möglichst zu vermeiden und das Don't repeat yourself (DRY) Prinzip einzuhalten.

```
1 def read_emergency_file(emergency_route_file: str) -> list:
2     if not os.path.exists(emergency_route_file):
3         return []
4     with open(emergency_route_file, "r") as emergency_route:
5         json_routes = emergency_route.read()
6         routes_from_file = RouteContainer.from_json(json_routes
      ).routes
7     return routes_from_file
```

Code Snippet 5.16: read_emergency_file Methode

```
1 def write_emergency_file(routes: RouteContainer,
      emergency_route_file: str):
2     with NamedTemporaryFile(delete=False, mode="w") as
      tmp_emergency_route_file:
3         tmp_emergency_route_file.write(routes.to_json())
4     try:
5         shutil.move(tmp_emergency_route_file.name,
      emergency_route_file)
6     except Exception as e:
7         click.echo(e)
8         os.remove(tmp_emergency_route_file.name)
```

Code Snippet 5.17: write_emergency_file Methode

## 5.3 Testen

# Kapitel 6

# Staging Umgebung

## 6.1 Planung

## 6.2 Umsetzung

# Kapitel 7

# Fazit

# Literatur


Beijnum, Iljitsch van [2002]. *Building Reliable Networks with the Border Gateway Protocol*. O'Reilly [siehe S. 6, 7].

King, Thomas u. a. [Okt. 2016]. *BLACKHOLE Community*. RFC 7999. doi: 10.17487/RFC7999. url: https://www.rfc-editor.org/info/rfc7999 [siehe S. 8].

Rekhter, Yakov, Susan Hares und Tony Li [Jan. 2006]. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. doi: 10.17487/RFC4271. url: https://www.rfc-editor.org/info/rfc4271 [siehe S. 6, 7].

Schoch, Leon [Okt. 2022]. *API für Route Injection* [siehe S. 3, 4].