Balanceo automatizado del tráfico de la red *core* de un proveedor de servicios de internet a través de la plataforma ONAP con la obtención de datos desde la herramienta Kentik

Ricardo Pellecer Orellana

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Balanceo automatizado del tráfico de la red *core* de un proveedor de servicios de internet a través de la plataforma ONAP con la obtención de datos desde la herramienta Kentik

Trabajo de graduación presentado por Ricardo Pellecer Orellana para optar al grado académico de Licenciado en Ingeniería Electrónica

Guatemala,

2024

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

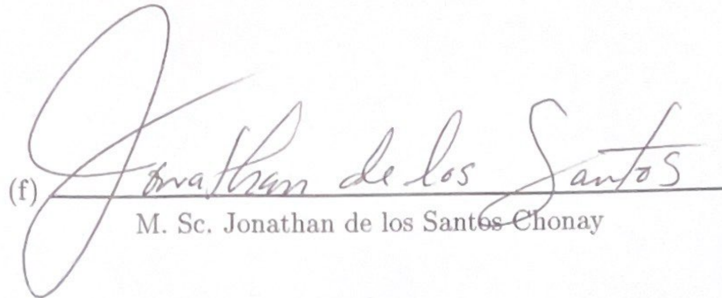
Balanceo automatizado del tráfico de la red *core* de un proveedor de servicios de internet a través de la plataforma ONAP con la obtención de datos desde la herramienta Kentik

Trabajo de graduación presentado por Ricardo Pellecer Orellana para optar al grado académico de Licenciado en Ingeniería Electrónica

Guatemala,

2024

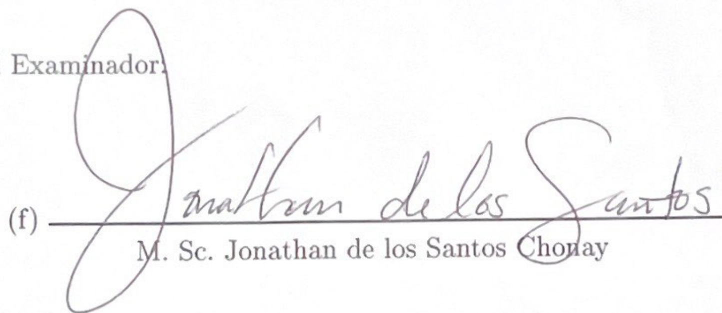
Vo.Bo.:



(f) ________________________________

M. Sc. Jonathan de los Santos Chonay

Tribunal Examinador:

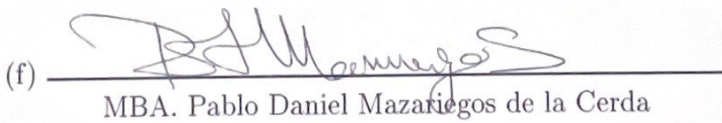


(f) ________________________________

M. Sc. Jonathan de los Santos Chonay

(f) ________________________________

MBA. María Alejandra Carrillo Albeño



(f) ________________________________

MBA. Pablo Daniel Mazariegos de la Cerda

Fecha de aprobación: Guatemala, 6 de enero de 2024.

Prefacio

A lo largo de mis estudios he desarrollado una gran pasión por la automatización, puesto que me permite combinar muchos temas que desde que entré a la universidad me han gustado para aplicarlos a ámbitos de la vida real, como lo es este proyecto. Para mí, estar en este punto de mi carrera universitaria es un gran logro, el cual sé perfectamente bien que no habría logrado solo. En primer lugar, le agradezco a Dios, quien me ha guiado en este camino lleno de obstáculos y me ha ayudado a no rendirme nunca. En segundo lugar, le agradezco a mis papás, quienes desde que tengo memoria han creído en mí y me han apoyado en mis sueños, que también vivieron muchos de mis desvelos y que se alegraron por mis logros y me animaron en mis preocupaciones.

También agradezco a mis hermanos, quienes me han apoyado en todo este trayecto. Quiero agradecer especialmente a mi hermano mayor, José David, ya que siempre ha sido un gran ejemplo a seguir, no solo académicamente, sino también como persona. Además de esto, también quiero agradecerle a mi novia, Daniela Morales, quien ha sido un gran apoyo a lo largo de la carrera y, aún más importante, la vida. Le agradezco también a mi abuelita, mi *mamita*, quien, aunque ya no esté conmigo, sé que me cuida desde el cielo y se alegra por mis logros.

Agradezco también a todos los profesores que demostraron que enseñar es su vocación día con día. Especialmente, le agradezco a Luis Rivera, quien siempre ha sido un maestro responsable y dedicado, a Magda Moscoso, que desde que la conocí mostró mucha fe en mí y me ha impulsado a crecer, y a José Manuel Morales, a quien yo considero un mentor y que me ha enseñado mucho más que solo conocimientos académicos.

De igual forma, quiero agradecer a mis compañeros de trabajo, Julio Shin, Israel Arévalo, José David Ponce y Rafael Paz, quienes me han apoyado en este proyecto y me han ayudado a crecer profesionalmente.

Finalmente, agradezco a todos mis compañeros de la carrera, con quienes compartimos muchos momentos buenos y nos apoyamos mutuamente en momentos difíciles. Les deseo lo mejor tanto en su vida profesional como personal.

Índice

Prefacio v

Lista de figuras xii

Lista de cuadros xiii

Lista de códigos xvii

Resumen xix

Abstract xxi

1. Introducción 1

2. Antecedentes 3

3. Justificación 5

4. Objetivos 7

4.1. Objetivo general 7

4.2. Objetivos específicos 7

5. Alcance 9

6. Marco teórico 11

6.1. Redes de telecomunicaciones 11

6.1.1. Internet 12

6.1.2. Red *Edge* 13

6.1.3. Red *Core* 13

6.1.4. Protocolo BGP 14

6.1.5. Arquiectura de red 16

6.2. APIs 18

6.2.1. REST 19

6.2.2. gRPC 19

6.3. Microservicios . . . 20
6.3.1. Virtualización . . . 21
6.3.2. Contenedores . . . 22
6.3.3. Orquestación de contenedores . . . 23
6.4. ONAP . . . 24
6.4.1. Arquitectura . . . 25
6.4.2. A&AI . . . 26
6.4.3. CDS . . . 27
6.4.4. SO - Camunda . . . 28

7. Prerequisitos para el desarrollo del proyecto en la plataforma ONAP 31
7.1. Prerequisitos de Camunda . . . 31
7.2. Creación del proyecto de Camunda . . . 33
7.3. Despliegue del flujo en Camunda . . . 34
7.4. Prerequisitos del CDS . . . 36
7.5. Creación del proyecto en el CDS . . . 37
7.6. Despliegue del CBA del CDS . . . 39

8. Obtención de la información del tráfico a través de las herramientas Kentik y SolarWinds 43
8.1. Flujo de Camunda para el descubrimiento de capacidades disponibles . . . 43
8.1.1. Diseño del *payload* para la REST API de Camunda . . . 43
8.1.2. Diseño del flujo en Camunda Modeler . . . 45
8.1.3. Adición de la funcionalidad del flujo . . . 47
8.2. Conexión a Kentik . . . 52
8.2.1. Creación de vistas en Kentik . . . 52
8.2.2. Obtención de la llamada a la API de Kentik correspondiente a la vista creada . . . 54
8.3. CBA para el descubrimiento de capacidades disponibles . . . 55
8.3.1. Configuración de archivos necesarios para el funcionamiento del CBA . 55
8.3.2. Diseño del *payload* para la REST API del CDS . . . 57
8.3.3. Algoritmo para determinar las capacidades disponibles para balancear el tráfico . . . 58
8.4. Conexión a SolarWinds . . . 63
8.4.1. Creación de la alarma . . . 63

9. Configuración de los equipos de red para realizar el balanceo de carga 69
9.1. Flujo de Camunda para la configuración . . . 69
9.1.1. Diseño del *payload* para la REST API de Camunda . . . 69
9.1.2. Diseño de los flujos en Camunda Modeler . . . 70
9.1.3. Adición de la funcionalidad del flujo . . . 72
9.2. CBA para la configuración . . . 76
9.2.1. Configuración de archivos necesarios para el funcionamiento del CBA . 76
9.2.2. Diseño del *payload* para la REST API del CDS . . . 77
9.2.3. Algoritmo para la generación de las plantillas de configuración . . . 78

10. Integración de herramientas de monitoreo 83
10.1. Creación de tableros en Grafana . . . 83

11. Funcionamiento integral del caso de uso **91**
11.1. Desarrollo de REST API para la ejecución completa del caso de uso 92
11.2. Simulación de la alarma de SolarWinds 94
11.3. Análisis de resultados 96

12. Conclusiones **101**

13. Recomendaciones **103**

14. Bibliografía **105**

15. Anexos **109**
15.1. Información general útil del proyecto 109
15.2. Utilidades para el uso de Kentik 109
15.3. Utilidades para el uso de Camunda 113
15.3.1. Definición de constantes 113
15.3.2. Métodos para la construcción de las peticiones a la base de datos . . . 115
15.3.3. Métodos para el *parseo* de información 116
15.4. Utilidades para el uso de SolarWinds desde un CBA 117
15.5. Detalle del algoritmo para determinar los posibles movimientos para el balanceo de tráfico 118
15.6. Detalle del algoritmo para generar las plantillas de configuración de los equipos 120
15.7. Utilidades diversas para los *scripts* de Python que son parte del CBA 124

Lista de figuras

1. Los modelos OSI y TCP/IP [8]. 12
2. La red de acceso con el *core* y *edge* de la red [15]. 13
3. Comparación entre BGP con y sin reflectores de rutas [18]. 15
4. Comparación entre BGP con y sin reflectores de rutas [22]. 16
5. Topología de red de un ISP con conexión a un NAP para lograr salir a Internet. 17
6. Mapa con las capacidades submarinas para la conexión a Internet conectadas a Guatemala [24]. 18
7. Diseño de APIs gRPC [30]. 20
8. Ejemplo de una arquitectura de microservicios [31]. 21
9. Comparación entre máquinas virtuales y contenedores [34]. 23
10. Vista general de las funciones principales de ONAP [38]. 25
11. Arquitectura de alto nivel de ONAP [39]. 26
12. Arquitectura de alto nivel de la A&AI [40]. 27
13. Arquitectura de bajo nivel del CDS [41]. 28
14. Estructura de un CBA [43]. 28
15. Ejemplo de un flujo siguiendo el estándar BPMN2.0 con Camunda Modeler [44]. 29
16. Camunda Cockpit [45]. 30
17. Modelo BPMN del flujo de Camunda para el descubrimiento de capacidades. 45
18. Modelo DMN para la obtención de credenciales de los módulos de ONAP y plataformas externas. 46
19. Proceso de selección y almacenaje de credenciales de plataformas externas a Camunda 46
20. Suscripción de un proceso implementado con *External Task* a un *topic*. 52
21. Interfaz de configuración vistas de Kentik. 53
22. Consumo en Gbps por interfaces de IP Transit. 54
23. Consumo en Gbps por interfaces de IP Transit según el prefijo. 54
24. Opción en Kentik para obtener el *payload* para la llamada a su API REST. . 55
25. Tablero para el manejo de las alertas de SolarWinds 64
26. Creación de una nueva alerta 64
27. Definición de las propiedades de la alerta 65
28. Definición de la condición que provoca que se levante la alerta 66

29. Definición de la condición para que la alerta se resuelva 66
30. Configuración para las horas del día en las cuales la alerta tiene efecto 67
31. Definición de la acción a realizar cuando se levanta la alerta 67
32. Detalles de la acción a realizar cuando se levanta la alerta 68

33. Modelo BPMN del flujo de Camunda para la configuración para el balanceo de carga. 71
34. Modelo BPMN del flujo de Camunda la ejecución de la configuración por equipo. 72

35. Página principal de Grafana. 84
36. Opción de Grafana para ver y crear tableros. 85
37. Folders de Grafana con los tableros. 85
38. Vista principal de un tablero de Grafana. 86
39. Creación de un nuevo panel para un tablero de Grafana. 86
40. Edición de un panel para un tablero de Grafana. 87
41. Edición de un *query* para la obtención de información para un panel. 88
42. Tablero de Grafana para el monitoreo de la salud del ambiente. 89
43. Tablero de Grafana para ver los *logs* de Camunda. 89
44. Tablero de Grafana para ver los *logs* del CDS. 90

45. Vista de la alarma de SolarWinds para la simulación de la ejecución del flujo integral a través del aplicativo en FastAPI. 95
46. Vista de la alarma de SolarWinds para la simulación de la ejecución del flujo integral a través del aplicativo en FastAPI. 96
47. Ejecución de la simulación del flujo integral sobre una interfaz específica. . . . 96

Lista de cuadros

1. Árbol de directorios y archivos del proyecto bpmn-commons 32
2. Árbol de directorios y archivos del proyecto bpmn-external-task-handler . . . 33
3. Árbol de directorios y archivos del proyecto load-balancing-workflows 34
4. Árbol de directorios y archivos del proyecto bgp-load-balancer 37

5. Árbol de directorios y archivos del proyecto bpmn-commons 92
6. Tiempos de ejecución de las pruebas de funcionamiento integral del caso de uso 99

7. APIs utilizadas en el proyecto 109

Lista de códigos

7.1. Comprobación de la existencia de los *pods* de Camunda . . . 31
7.2. *Script* de automatización para cargar los proyectos de Java a los *pods* de Camunda . . . 35
7.3. *Logs* esperados luego de subir los proyectos de Java a Camunda . . . 36
7.4. Comprobación de la existencia de los *pods* del CDS . . . 36
7.5. Archivo TOSCA.meta del CBA del proyecto . . . 38
7.6. Definición de flujos a utilizar en el CBA . . . 38
7.7. Configuración del macro para el *Python Executor* . . . 39
7.8. *Enrichment* de los CBAs para poder desplegarlos al CDS . . . 39
7.9. *Save/Deploy* de los CBAs . . . 40
7.10. Automatización del proceso de *Enrichment* y *Save/Deploy* de los CBAs . . . 40
8.1. *Payload* para la ejecución del flujo de descubrimiento de capacidades . . . 44
8.2. Almacenaje de variables acorde a las selecciones del DMN . . . 47
8.3. Construcción y ejecución de la petición a la API REST de Kentik . . . 48
8.4. Construcción y ejecución de la petición a la API REST de Kentik . . . 49
8.5. Respuesta de Kentik con la información del tráfico por prefijo . . . 50
8.6. Construcción del *payload* para la ejecución del CBA de descubrimiento de capacidades . . . 51
8.7. Definición del *Workflow* de descubrimiento de capacidades disponibles . . . 55
8.8. Definición del nodo para la ejecución del código en el *python executor* para el descubrimiento de capacidades disponibles . . . 56
8.9. Definición del macro para la aceptación de parámetros . . . 56
8.10. *Payload* para la constula al servicio cds-blueprints-processor-http para la ejecución del CBA de descubrimiento de capacidades . . . 57
8.11. Clase para la ejecución del CBA de descubrimiento de capacidades disponibles 58
8.12. Clase para la obtención y uso de la información de SolarWinds . . . 60
8.13. Método para la obtención de información de las interfaces IP Transit . . . 61
8.14. Método para la obtención de los posibles movimientos que se pueden realizar para balancear la carga . . . 62
9.1. *Payload* para la ejecución del flujo de configuración . . . 70
9.2. Fase de construcción de la petición en el patrón *Construct - Put - Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos . . . 73

9.3. Fase de envío de la petición en el patrón *Construct - Put - Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos 73
9.4. Fase de transformación, filtrado y utilización de la información de la petición realizada en el patrón *Construct - Put - Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos 74
9.5. Fase de construcción de la petición en el patrón *Construct - Put - Parse* para la obtención de las credenciales del equipo a configurar 74
9.6. Fase de transformación, filtrado y utilización de la información de la petición realizada en el patrón *Construct - Put - Parse* para la obtención de las credenciales del equipo a configurar 75
9.7. Construcción del *payload* para la ejecución del CBA 75
9.8. Definición del *Workflow* de configuración 77
9.9. Definición del nodo para la ejecución del código en el *python executor* para la configuración 77
9.10. *Payload* para la constula al servicio cds-blueprints-processor-http para la ejecución del CBA de configuración 78
9.11. Clase para la ejecución del CBA de configuración 79
9.12. Plantilla de configuración de los equipos locales Juniper para el balanceo de tráfico de internet 81
11.1. Archivo main.py para el funcionamiento del *gateway* 92
11.2. Definición de los esquemas para los *payloads* de entrada de los *endpoints* . . . 94
11.3. *Payload* para la ejecución del flujo integral a través del aplicativo en FastAPI 95
15.1. *Payload* para la API REST de Kentik que obtiene la información del tráfico a través de capacidades de IP Transit 110
15.2. Clase para la definición de constantes útiles para el funcionamiento de los flujos de Camunda 114
15.3. Clase para la definición de constantes útiles para el funcionamiento de los flujos de Camunda 114
15.4. Fase de construcción de la petición en el patrón *Construct - Put - Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos 115
15.5. Fase de construcción de la petición en el patrón *Construct - Put - Parse* para la obtención de las credenciales del equipo que se va a configurar 115
15.6. Fase de *parseo* de la petición en el patrón *Construct - Put - Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos116
15.7. Fase de *parseo* de la petición en el patrón *Construct - Put - Parse* para la obtención de las credenciales del equipo que se va a configurar 117
15.8. Clase que contiene las peticiones a la base de datos de SolarWinds 117
15.9. Método para la creación de las posibles combinaciones entre interfaces y prefijos para realizar el balanceo 118
15.10Método para la obtención de los posibles movimientos que se pueden realizar para balancear la carga 119
15.11Método para la obtención del país al que pertenece un PNF en específico . . . 120
15.12Clase para el descubrimiento de información de los equipos de la red 120
15.13Método para la obtención, interpretación y uso de la información de los equipos de red para generar las plantillas de configuración 121
15.14Clase con métodos útiles para la ejecución de DSL *queries* y *parseo* de la información 123

15.15 Decorador para el manejo de errores 124
15.16 Decorador para medir el tiempo de ejecución de las funciones 124

Resumen

En este trabajo se presentó una aplicación diseñada sobre la plataforma ONAP, la cual se encarga de automatizar el proceso de balanceo de tráfico de la red core de un proveedor de servicios de internet (ISP). La red utilizada está conformada por equipos Juniper y el balanceo de tráfico se realiza a través de la conmutación del tráfico sobre los enlaces a nivel BGP con comunidades. Estas comunidades son utilizadas por las políticas de importación y exportación de los equipos para poder anunciar de forma diferente un prefijo de red específico a través del agregado de *prepends* en diferente cantidad para beneficiar un camino u otro, creando así el balanceo de carga en el tráfico de la red.

Esta aplicación fue diseñada y desarrollada sobre una arquitectura de microservicios, la cual consta de cuatro servicios principales: un servidor HTTP, el cual cuenta con una API que contiene múltiples *endpoints* para mandar a ejecutar instrucciones de forma remota; un servidor RPC, el cual es el ejecutor de comandos y el que se conecta con los equipos de la red para configurar de forma remota a través del protocolo Netconf; una base de datos no relacional, en la cual se guarda con una topología de grafos toda la información correspondiente a los equipos para poder configurarlos y saber sus estados cuando se necesite; y otro servidor HTTP, el cual también cuenta con una API con múltiples *endpoints* para ejecutar archivos que siguen el estándar BPMN, los cuales se encargan de realizar toda la preparación de la información de la base de datos para poder utilizar los dos primeros servicios de forma automática.

Para lograr que el funcionamiento de la aplicación sea automático, se utilizó la herramienta SolarWinds, la cual está sensando la red de forma periódica. Sobre esta herramienta se configuró una alarma, la cual se activa cuando el tráfico de internet sobrepasa un umbral predefinido y se encarga de ejecutar a través del servidor HTTP antes mencionado los flujos definidos en los archivos BPMN. Estos flujos se encargan de consultar a la plataforma de monitoreo Kentik, la cual permite hacer consultas sobre el tráfico de cada enlace desglosado según el prefijo. Finalmente, se creó un tablero en la herramienta Grafana para mostrar información valiosa sobre todo el proceso de balanceo de tráfico y sus resultados más importantes.

Abstract

In this work, an application was designed and implemented with the use of the network automation platform ONAP, which automates the traffic balancing process of the core network of an Internet Service Provider (ISP). The network used is composed of Juniper equipment and the traffic balancing is done through the switching of traffic over the BGP links with communities. These communities are used by the import and export policies of the equipment to be able to announce a specific network prefix differently through the addition of prepends in different amounts to benefit one path or another, thus creating the load balancing in the network traffic.

This application was designed and developed on a microservices architecture, which consists of four main services: an HTTP server, which has an API that contains multiple endpoints to execute instructions remotely; an RPC server, which is the command executor and which connects to the network equipment to configure remotely through the Netconf protocol; a non-relational database, in which all the information corresponding to the equipments is saved with a graph topology to be able to configure them and know their current configurations when needed; and another HTTP server, which also has an API with multiple endpoints to execute files that follow the BPMN standard, which are responsible for performing all the preparation and processing of the information in the database to be able to use the first two services automatically.

To achieve the automatic operation of the application, the SolarWinds tool was used, which senses the network periodically. On this tool, an alarm that is responsible for executing the flows defined in the BPMN files through the aforementioned HTTP server was configured, which is activated when the Internet traffic exceeds a predefined threshold. These workflows are in charge of consulting the Kentik monitoring platform, which allows queries to be made about the traffic of each link broken down by prefix. Finally, a dashboard was created in the Grafana tool to show valuable information about the entire traffic balancing process and its most important results.

CAPÍTULO 1

Introducción

El balanceo de carga del tráfico de internet es un proceso que realizan todas las empresas de telecomunicaciones, ya que este permite que las empresas aprovechen de la mejor forma posible las capacidades que tienen de tránsito IP, *peering*, terrestres y submarinas. Nótese que el balanceo de carga, si bien es cierto que se da principalmente para que las capacidades no se saturen y, por ende, no se pierdan paquetes, este proceso no necesariamente busca una distribución equitativa del tráfico, puesto que diferentes proveedores de enlaces submarinos o de enlaces de tránsito IP cobran diferente con modelos de cobro complejos. Además de esto, una vez las empresas de telecomunicaciones empiezan a crecer y cuentan con múltiples operaciones (ya sea por estados, departamentos, países o cualquier división lógica para el negocio), existen más prefijos que pueden pasar por alguna de las capacidades. En otras palabras, cuando la empresa escala, también lo hace la complejidad de su operación. Sin embargo, es muy importante mencionar que, a pesar de que la complejidad se vuelve más fuerte, el trabajo sigue siendo mecánico y se pueden encontrar patrones en los procesos a realizar para poder alcanzar un balanceo de tráfico exitoso. Esta complejidad agregada de un trabajo repetitivo y con un algoritmo lógico estandardizado permite que la automatización de procesos sea una solución viable para este tipo de problemas.

En este trabajo de graduación se propone una solución por *software* a través de la herramienta de automatización de redes ONAP, con la cual se logra automatizar el proceso mecáncio, manual y difícil de escalar del balanceo de tráfico de internet, como fue mencionado anteriormente. Además de esto, para que más automatizaciones puedan realizarse y construirse encima de este proyecto, este se desarrolló de tal forma que siga el estándar Cloud Native, lo cual quiere decir que tiene una arquitectura de microservicios y que está enfocado a que los módulos sean accesibles entre sí y desacoplados. Esto se logró a través del desarrollo dejando todas las funcionalidades en APIs, ya que de esta forma, cualquier desarrollo futuro puede utilizar el actual sin ningún problema y de forma desacoplada a través del consumo del *endpoint* que desee.

En el caso de esta tesis, el desarrollo se enfocó específicamente en el balanceo de tráfico de internet en las capacidades de tránsito IP. Este proceso, como se mencionó antes, puede

llegarse a complicar y a tener problemas de escalamiento cuando la empresa que entrega los enlaces de internet va creciendo, ya que existen más opciones de *Tier 1* con el que se tienen contratos de tránsito IP, existen redundancias de los equipos en donde se realiza el tránsito (que en este caso son equipos en el IXP de Miami), existen múltiples operaciones (por ejemplo, una empresa internacional puede tener operaciones en Guatemala, El Salvador, entre otros países), cada operación cuenta con redundancias de los equipos locales (siguiendo el ejemplo de una empresa internacional, se pueden tener equipos en diferentes centros de datos físicos), entre otros casos a considerar. Para lidiar con estas dificultades, en este proyecto se trabajó con los gestores de monitoreo SolarWinds y Kentik, los cuales son utilizados para obtener información confiable en tiempo real del estado de la red de una forma granular; con estas herramientas se logró detectar cuando una capacidad de tránsito IP se comienza a saturar y qué opciones existen de movimientos de prefijos para lograr un balanceo correcto. Finalmente, se utilizaron *scripts* de Python, con los cuales se generó la configuración necesaria para los equipos de las operaciones pertinentes para realizar los cambios encontrados con la ayuda de los gestores.

CAPÍTULO 2

Antecedentes

Para un proveedor de servicios de internet (ISP) es crucial el balanceo del tráfico para asegurar una conectividad constante, garantizar la calidad del servicio entregado y generar redundancias que evitan puntos únicos de falla o *Single Points of Failure* (SPOFs). Este proceso de balanceo de cargas tiene muchas formas de lograrse, sin embargo, dos de las formas principales en la que los ISPs realizan esto es a través del uso de los protocolos MPLS y BGP. La forma en que se utilizan estos protocolos, sin embargo, no contempla la necesidad del balanceo de tráfico debido a la saturación de los diferentes enlaces, por lo que muchas veces no se aprovecha de la mejor forma posible las redundancias que se tienen. Por esto, actualmente, estos cambios deben de ser realizados de forma manual, ocasionando problemas de agilidad en los procesos, errores humanos y mal uso de recursos en actividades mecánicas y repetitivas.

Un caso real de un ISP actualmente es que cuenta con su interconexión a Internet a través de una arquitectura de red que separa a la red *core* en tres capas lógicas: un anillo terrestre, varias capacidades submarinas que van desde el anillo terrestre hacia un punto de interconexión a Internet (IXP) y el peering con otros carriers dentro del IXP. El anillo terrestre cumple la función de brindar conectividad a nivel local con redundancias por si se cae algún enlace; las capacidades submarinas también brindan redundancias para hacer la conexión entre la red local y la red global que se conoce como Internet; y los peerings permiten otro nivel de redundancia a través de la elección de por dónde vendrá el tráfico que venga de la red de Internet. El tráfico dentro de las primeras dos capas lógicas se maneja a través del protocolo MPLS con LSPs ya definidos que corresponden a los diferentes caminos que el tráfico puede tomar, mientras que la tercera capa lógica maneja el tráfico a través del protocolo BGP con el uso de comunidades y prepends, esta última siendo el enfoque en este proyecto. Actualmente, si se desea realizar algún tipo de balanceo de carga, debe de encontrarse el prefijo que desea moverse, ver en cuál de estas capas lógicas necesita hacerse el movimiento, encontrar las políticas y comunidad BGP con algún *prepend* y cambiarla para que ahora el prefijo tenga asignado otra comunidad y el tráfico pueda fluir por otro camino con menor carga. Como puede verse, este método manual actual tiene varias complicaciones, dentro de las cuales se podría decir que las principales son que no se sabe realmente el

consumo de ancho de banda del prefijo, por lo que no se sabe si al realizar el movimiento se podría estar liberando una capacidad solo para sobrecargar otra en el proceso, haciendo del proceso algo iterativo y a prueba y error; y que hay demasiados comandos y configuraciones de los equipos de red, por lo que hay mucho riesgo de que haya algún error humano.

Por situaciones similares a esta, en los últimos años, muchos procesos de telecomunicaciones antes manuales se han automatizado gracias a herramientas como ONAP, Intraway Symphonica, Open Source MANO (OSM), OpenStack Tacker, Apstra, Ansible, entre otros. Un ejemplo claro de esto es que, como puede verse en [1], ONAP ha sido utilizado para realizar la automatización del aprovisionamiento de servicios capa 2 a través de la orquestación de las configuraciones de dispositivos virtualizados. Por otro lado, en [2] se mostró una prueba de concepto en la que se realiza un balanceo de cargas en el tráfico de dos sitios técnicos con alta disponibilidad (HA) que utilizan OpenStack para montar su infraestructura. Estos trabajos logran demostrar que es factible utilizar sistemas de orquestación para la automatización de procesos de telecomunicaciones.

Este tipo de proyectos también pueden verse en el caso de uso de Aarna Networks presentado en [3], en donde la empresa de telecomunicaciones Tigo Guatemala colaboró con Aarna Networks para lograr la automatización del aprovisionamiento y diagnóstico de servicios de enlaces de datos capa 3 sobre MPLS dados a través de equipos físicos de múltiples proveedores. Este caso de uso logra sentar las bases para la automatización de cualquier proceso de telecomunicaciones que implique obtener información de equipos físicos como routers y switches, tomar decisiones en base a dicha información para luego configurarlos. Parte de este trabajo de graduación es tomar esa base ya existente para poder obtener información de equipos de red Juniper y tomar decisiones en base a eso para poder manejar el tráfico en la red.

A pesar de estas herramientas mencionadas, lograr realizar este balanceo de tráfico puede llegar a ser un reto muy complejo solo utilizando la información que brindan los equipos, por lo que existen herramientas que permiten obtener esta información de forma más fácil, rápida y detallada. Una de las herramientas más utilizadas a nivel mundial es Kentik. Según [4], la empresa de telecomunicaciones vietnamita FPT logró reducir el tiempo de identificación y diagnóstico de problemas en la red y obtener conocimientos sobre el flujo del tráfico de la red para evitar congestiones utilizando el módulo de *Network Observability* de Kentik, el cual da una visibilidad del estado del tráfico en la red a un nivel tan detallado como la utilización del ancho de banda por prefijo. Esta herramienta junto con este caso de uso logran ser un apoyo para este trabajo de graduación, haciendo una combinación de la obtención de data desde Kentik con el uso de ONAP para la automatización de flujos para tomar decisiones según esa información para que la configuración remota de los equipos sea posible.

CAPÍTULO 3

Justificación

Internet es la red global de redes que permite la existencia de la comunicación como se conoce, en donde dos o más personas pueden hablar entre sí a través de mensajes de forma casi instantánea, se puede consumir contenido como música, fotos y videos, se puede tener toda la información importante guardada en un solo lugar seguro como la nube, entre otros casos de uso. Para que esto pueda suceder y se mantenga funcionado todo el tiempo, las compañías de telecomunicaciones que proveen servicios de Internet (ISP) necesitan lograr mantener conectados a todos sus clientes con todos los servicios que la red ofrece, para lo cual, los ISP dividen a la red que manejan en dos grandes grupos: el *core* y el *edge*. La red *core* es la encargada de interconectar las redes de los distintos ISP del mundo para que el Internet sea lo que se conoce en la actualidad; por otro lado, el *edge* es la red que da hacia los clientes. Por esto mismo, sobre la red *core* existe mucho tráfico en comparación al *edge*, ocasionando la necesidad de tener enlaces con mayor capacidad, redundancias y balanceo de carga. Este último punto mencionado es realizado comúnmente a través de las comunidades del protocolo BGP y del uso de caminos LSP del protocolo MPLS, los cuales permiten anunciar cómo llegar hacia una parte de la red, indicando por cuál enlace pasar.

Esta tarea de balanceo de carga es típicamente encargada a los ingenieros del NOC (*Network Operations Center*), quienes se encargan de hacer las configuraciones en los equipos de la red de forma manual. Esto implica que los ingenieros tienen que estar constantemente monitoreando la red y estar pendientes para realizar el cambio en el equipo correcto. Dicho proceso tiene la desventaja que los ingenieros pueden llegarse a equivocar en el equipo que se configura o en la configuración que se ingresa en caliente, pudiendo causar grandes pérdidas en la comunicación e incluso desconexiones temporales entre Internet y el ISP. Asimismo, esta tarea puede ser tediosa para los ingenieros, quienes podrían estar invirtiendo sus esfuerzos en tareas más innovadoras para la empresa. Por esto, es necesario hacer la transición para que esta configuración sea realizada de forma automática por una plataforma que hace las cosas de forma más eficiente, no se equivoca con *typos*, no cruza equipos y puede hacerlo en cualquier momento en caliente sin tener ningún riesgo de tener un impacto negativo sobre el rendimiento de la red. Por ende, este proyecto tendría un alto impacto en la calidad de servicio entregado a los clientes del ISP y mejoraría la productividad y el impacto del trabajo

de los ingenieros. Debido a esto, durante este proyecto se realizará la automatización del balanceo de cargas antes mencionado teniendo como alcance la parte de BGP.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Utilizar la plataforma ONAP para realizar flujos automatizado para realizar el proceso completo de balanceo de cargas BGP en una red *core* de un ISP para la entrega de internet, haciendo uso de información del tráfico proveniente del sistema de monitoreo de redes Kentik.

4.2. Objetivos específicos

- Comprender a detalle el flujo completo de balanceo de cargas manual que realiza un ISP real a nivel de BGP.
- Utilizar una arquitectura de infraestructura con máquinas virtuales con Linux Ubuntu para la implementación de un caso de uso sobre la plataforma ONAP a través de un cluster de Kubernetes.
- Realizar el desarrollo de los *scripts* de automatización en los lenguajes Python y Java siguiendo el estándar de desarrollo del ecosistema de ONAP.
- Seguir la arquitectura de microservicios para implementar la plataforma ONAP junto con Kentik y otras herramientas de monitoreo interno como el stack tecnológico de Grafana y SolarWinds.
- Realizar desarrollos haciendo uso de la API de la herramienta Kentik para poder analizar el tráfico de la red *core* de internet de un ISP y configurar para que esta información llegue a ONAP.
- Implementar una primera versión de la solución como primer caso de uso en la red *core* de un ISP real para demostrar que la red balancea de forma automática, haciendo uso de dos flujos principales: la identificación de las interfaces saturadas, con sus prefijos y la cantidad de ancho de banda que están consumiendo, y las interfaces a donde se

puede hacer el movimiento; y la ejecución de las configuraciones en los equipos para poder hacer el balanceo del tráfico.

CAPÍTULO 5

Alcance

Este trabajo de graduación está enfocado en el desarrollo del caso de uso de balanceo automatizado del tráfico de internet sobre la plataforma de automatización ONAP, haciendo uso de la herramienta de monitoreo Kentik. El tráfico de internet tiene muchos niveles a través de los cuales se puede balancear; este trabajo de graduación está enfocado en el balanceo al nivel de los *Tier 1* y está desarrollado para que funcione únicamente con tráfico que utiliza IPv4, es decir, este trabajo no contempla el balanceo a nivel de capacidades submarinas o de anillos terrestres, ni tampoco el balanceo de tráfico que utiliza IPv6. De igual forma, este trabajo de graduación busca seguir la arquitectura de microservicios que ofrece ONAP para conectar sus componentes a las herramientas externas Kentik, SolarWinds y Grafana; es decir, se busca conectar a todos los aplicativos entre sí a través de APIs, de tal forma que los componentes sean independientes entre sí. Nótese que no es parte del alcance de este proyecto de graduación levantar estas herramientas, sino utilizarlas y conectarlas entre sí para lograr desarrollar el caso de uso propuesto.

CAPÍTULO 6

Marco teórico

6.1. Redes de telecomunicaciones

Las redes de telecomunicaciones son formas de interconectar dispositivos, como PCs, teléfonos, servidores, entre otros, para que intercambien información entre sí [5]. Hay muchas formas de diseñar redes de telecomunicaciones, sin embargo, los modelos más utilizados son el modelo OSI y el modelo TCP/IP. Estos modelos nacen de la necesidad de realizar capas de abstacción por la complejidad de todo lo que implica poder interconectar dos dispositivos para que intercambien información de forma escalable, ya que se necesita pensar en cómo van a llegar los mensajes desde un punto de vista físico hasta un punto de vista funcional para que las aplicaciones puedan ser útiles y estándares. Para cada una de estas capas de abstracción se crean protocolos o conjuntos de reglas que se tienen que seguir para que se puedan entender entre dispositivos.

Por esto, el modelo OSI es un modelo que define 7 capas de funcionamiento: las capas física, de enlace de datos, de red, de transporte, de sesión, de presentación y de aplicación [6]. La capa física es la encargada de ver todo lo relacionado al aspecto físico del transporte de datos, incluyendo al medio que se utiliza (radiofrecuencia, pares trenzados de cobre, fibra óptica, cables coaxiales, entre otros) y la forma en que se degrada la señal y cómo lidiar con estas situaciones. La capa de enlace de datos es la capa encargada de lograr que la información llegue de un dispositivo de origen al dispositivo considerado como el siguiente salto o *next hop*, el cual es el siguiente paso para poder llegar hasta el destino final deseado en la comunicación. La capa de red se divide en el plano de control y el plano de datos, los cuales son los encargados de determinar cuál debe de ser el siguiente salto y de realizar la conmutación del paquete hacia dicho siguiente salto respectivamente. La capa de transporte es la que permite la comunicación entre dos o más procesos entre dispositivos finales. La capa de sesión es la responsable de iniciar y cerrar las sesiones de comunicación entre dos dispositivos finales de la red. La capa de presentación es la responsable de preparar los datos para que puedan ser utilizados por las aplicaciones o procesos de dos dispositivos finales, tomando en cuenta la encriptación y compresión de los datos. Finalmente la capa de

aplicación es la encargada de interactuar con los datos del usuario utilizados en los procesos de los dispositivos finales [7]. Este modelo tiene una variación en donde solo se dejan 5 capas, donde las capas de sesión, presentación y aplicación se juntan en una sola. Por otro lado, el modelo TCP/IP es otra variación del modelo OSI de 5 capas, en donde se reduce una capa más juntando las capas física y de enlace de datos en una sola, como se puede ver en la Figura 1.

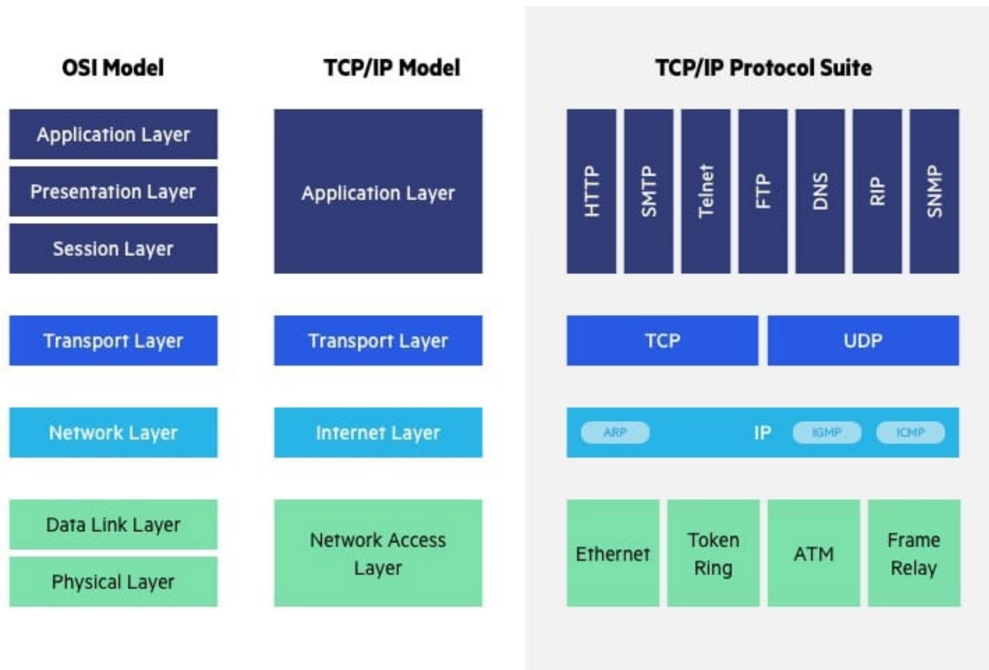


Figura 1: Los modelos OSI y TCP/IP [8].

Gracias a la escalabilidad de estos modelos, con este tipo de redes de telecomunicaciones se puede obtener conectividad a todos los niveles requeridos, desde una red local privada de una empresa pequeña hasta una red global como lo es Internet.

6.1.1. Internet

El Internet es una red de computadoras alrededor del mundo interconectadas entre sí que permite a las personas que se comuniquen y a que compartan y accedan a información de cualquier lugar del mundo [9]. A nivel lógico, la forma en que el Internet logra realizar este traslado de información de un usuario final (conocido como *host*) a otro es a través del uso de estándares y protocolos definidos por asociaciones como la IEEE, la IETF, la IANA, entre otros [10]. Entrando a un nivel más profundo, a nivel de dispositivos, la forma en que el Internet logra realizar esta interconexión es a través de equipos de red que se encargan de recibir la información y redirigirla al equipo correcto para que la información llegue de un usuario final a otro, a los cuales se les conoce como *routers* y *switches*, teniendo cada uno una función diferente dentro del funcionamiento del Internet según la capa del modelo OSI en la que operen. Finalmente, entrando al nivel más bajo, la forma en que el Internet se da es a través de la transmisión de la información dada por los *routers* y *switches* a través

de medios físicos como cobre, fibra óptica, radio frecuencia, entre otros, siguiendo de igual forma los protocolos y estándares antes mencionados. A grandes rasgos, el Internet se divide en dos: el *edge* y el *core* [5], resultando en una topología de red como la que se puede ver en la Figura 2.

6.1.2. Red *Edge*

La red *edge* es la parte de la red en donde se encuentran los *hosts*, los cuales son los equipos que corren las aplicaciones de internet, tanto los servidores como los equipos que consumen la información dada por los servidores [11]. La red del *edge* se conecta hacia el *core* de un lado y hacia los usuarios finales del otro a través de la red de acceso, la cual es la red que físicamente conecta al *host* al primer equipo de red, ya sea *router* o *switch*, que va a llevar la información hacia otro *host*. La tecnología utilizada para estas redes de acceso varía según la aplicación y pueden ser redes DSL, cable, FTTH, *Dial-Up* o satelitales. Estas redes están definidas por los protocolos y estándares sobre los que están diseñados y el medio físico que utilizan, ya sea pares cruzados de cobre, cable coaxial, fibra óptica, radio frecuencia, etc [12].

6.1.3. Red *Core*

Por otro lado, *core* es la parte de la red donde se encuentran todos los *routers* y *switches* interconectados entre sí para hacer posible que los *hosts* del *edge* tengan conectividad entre sí [13]. El Internet está compuesto de varios *cores*, donde cada proveedor de Internet cuenta con el uno o varios, según sea la escala del servicio que entrega; es decir, el Internet no tiene un solo *core*, sino que el Internet es la red que conecta a los *hosts* de todo el mundo, interconectando múltiples *cores* para poder lograrlo. La forma en que el Internet logra realizar esta conectividad entre varios proveedores de internet es a través del protocolo BGP, el cual permite compartir la información que tiene un proveedor de cómo llegar a ciertos dispositivos finales a otros proveedores. Los lugares en donde se realizan estas conexiones se les conoce como NAPs o *Network Access Points* [14].

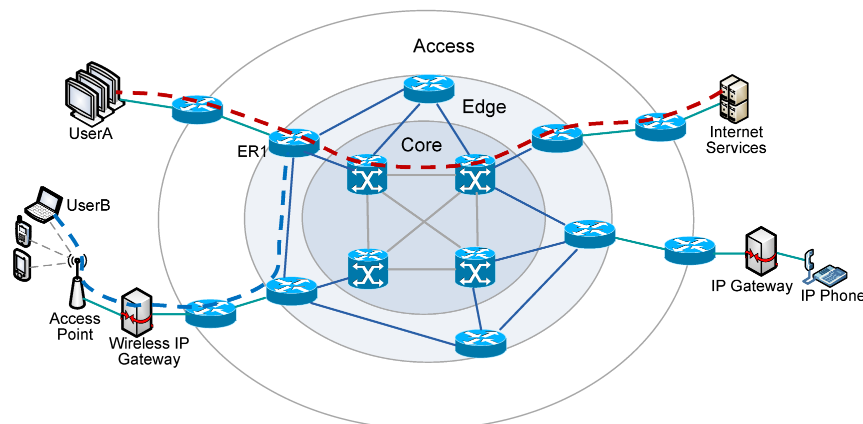


Figura 2: La red de acceso con el *core* y *edge* de la red [15].

6.1.4. Protocolo BGP

El Internet, como se mencionó antes, está dividido en varios proveedores que tienen sus propios *cores* de la red que se interconectan entre sí. Estas diferentes redes de proveedores que conforman al Internet se organizan por sistemas autónomos o AS por sus siglas en inglés, los cuales de forma interna comparten las mismas políticas de enrutamiento y están administradas por una sola entidad [16]; en el contexto de Internet, un proveedor puede ser uno o varios sistemas autónomos según el tamaño de la red que tenga ese proveedor. BGP utiliza esta idea y es utilizado como el protocolo que logra hacer el enrutamiento a nivel inter sistemas autónomos, siendo así el protocolo de enrutamiento de Internet por excelencia.

BGP es un protocolo de la capa de aplicación que es utilizado como protocolo de enrutamiento dinámico de tipo vector de distancias, lo cual quiere decir que este protocolo se encarga de determinar de forma dinámica cuál es el mejor próximo salto para alimentar a la tabla de rutas que es utilizada por los *routers* (que operan en capa 3) para conmutar los paquetes hacia ese mejor próximo salto a través del algoritmo de selección del mejor camino. BGP es un protocolo de reenvío, lo que significa que los *routers* que están corriendo BGP levantan sesiones entre sí para volverse vecinos y así poder mandarse la información del enrutamiento dinámico que realizan [17].

En BGP, para poder levantar una sesión y mantenerla se pasa por muchos pasos, los cuales están mejor representados en el diagrama de la Máquina de Estados Finitos del algoritmo que realiza este proceso, el cual puede verse en la Figura 3. Como puede verse, la sesión inicia como *Idle*, es decir, el equipo detecta un mensaje de inicio. Luego, el segundo paso es *Connect*, en el cual se inicia la sesión TCP con el equipo que originó el mensaje de inicio a través del *3-way handshake*. Si la sesión TCP no es levantada exitosamente, el equipo se queda intentando volver a iniciar la sesión TCP, quedándose en estado *Active*. Por otro lado, si la sesión TCP sí se logra concretar se manda un mensaje de apertura para la comunicación, por lo que el equipo pasa a estado *OpenSent*, durante el cual el equipo está a la espera de que el equipo vecino también mande su mensaje de apertura de la comunicación para poder revisar que todo esté en orde. Una vez el equipo reciba el mensaje de apertura y revise que todo está bien, este pasa a *OpenConfirm*, en donde el equipo espera una notificación o un *keep alive* de su vecino para poder confirmar la adyacencia creada. Finalmente, si se confirma la relación entre los equipos, la sesión pasa a estar en estado *Established*, en donde se está esperando mensajes *keep alive* cada cierto tiempo para poder mantener la sesión o regresarla a estado *Idle* de ser necesario [18].

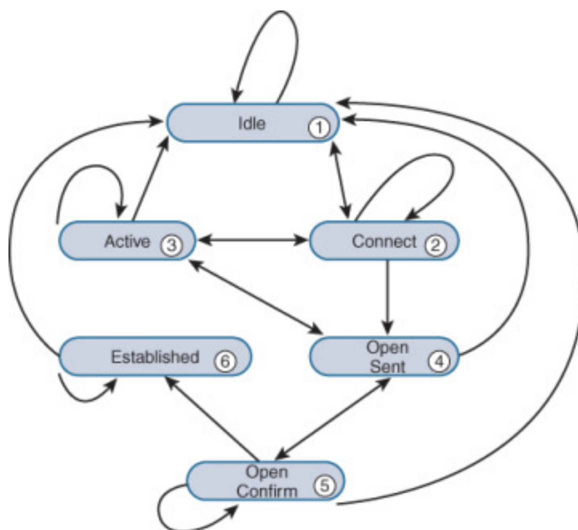


Figura 3: Comparación entre BGP con y sin reflectores de rutas [18].

Este protocolo está dividido en BGP externo (eBGP) y BGP interno (iBGP), los cuales se encargan de hacer enrutamiento inter sistema autónomo e intra sistema autónomo respectivamente. A pesar de que iBGP no es necesariamente el protocolo que utilizan los proveedores de Internet para realizar su enrutamiento interno, eBGP sí es el protocolo que todos los proveedores de Internet a nivel mundial utilizan como estándar para comunicarse entre sí [19]. La forma en que BGP funciona es a través del envío de anuncios de prefijos de red para decirle al resto de dispositivos dónde está esa subred y por dónde llegar, luego los equipos a los que le llega esa información determinan cuál de los anuncios proporcionó el mejor próximo salto para poder llegar a ese prefijo y lo coloca en la tabla de rutas, finalmente, el protocolo reenvía esa información que tiene en su tabla de rutas a los otros *routers* con los que tiene sesión BGP [20].

BGP, como se explicó antes, es un protocolo que toma la información que tiene en su tabla de rutas y la reenvía a sus vecinos BGP, sin embargo, este comportamiento tiene una excepción, lo cual es una de las diferencias entre eBGP e iBGP, ya que si la tabla de rutas tiene una ruta aprendida por iBGP, esta no se reenvía. Esto es realizado de esta forma porque iBGP se utiliza de forma interna en los AS, por lo que, si no se colocara esta restricción, se crearían bucles en el tráfico ocasionado por los anuncios y se saturaría la red. Esto ocasiona que, si de forma interna se quiere utilizar iBGP, se tienen que crear sesiones BGP de todos los equipos con todos los equipos, también conocido como *full mesh*, lo cual es algo no es escalable para sistemas autónomos como los de proveedores de internet grandes. Esta situación tiene una solución alterna mucho más escalable conocida como reflectores de rutas, los cuales son *routers* que se colocan en medio del *core* de la red a los cuales se les levanta sesiones iBGP especiales que sí realizan el reenvío de rutas contra todos los demás *routers*. De esta forma, un *router* del *core* aprendería una ruta proveniente de otro AS por eBGP, se la mandaría por iBGP al reflector de rutas, este reenviaría por iBGP (que es la parte especial de esta solución) hacia todos los *routers* con los que tiene sesiones levantadas y estos ya no las reenviarían de regreso porque están corriendo iBGP normal. Esta solución hace que al ingresar un nuevo *router* este solo necesite levantar una sesión con el reflector de rutas, mientras que en la solución del *full mesh* se tendrían que levantar sesiones contra

todos los *routers* ya existentes, como se ve en la Figura 4 [21].

Para poder ver la comparación entre los dos métodos de una forma más numérica, la cantidad de enlaces requeridos para que cierta cantidad de equipos en una red compartan sus tablas de rutas entre sí a través de BGP sin reflectores está dada por $n = \frac{m(m-1)}{2}$, donde n es la cantidad de enlaces y m es la cantidad de equipos. Por otro lado, en el caso en que haya un reflector la cantidad de enlaces está dada por $n = m$, donde n es la cantidad de enlaces y m es la cantidad de equipos diferentes del reflector. Esto demuestra que, para redes con más de tres equipos ya se comienza a ver una reducción en la cantidad de enlaces, aunque este método se utiliza comúnmente para redes mucho más grandes.

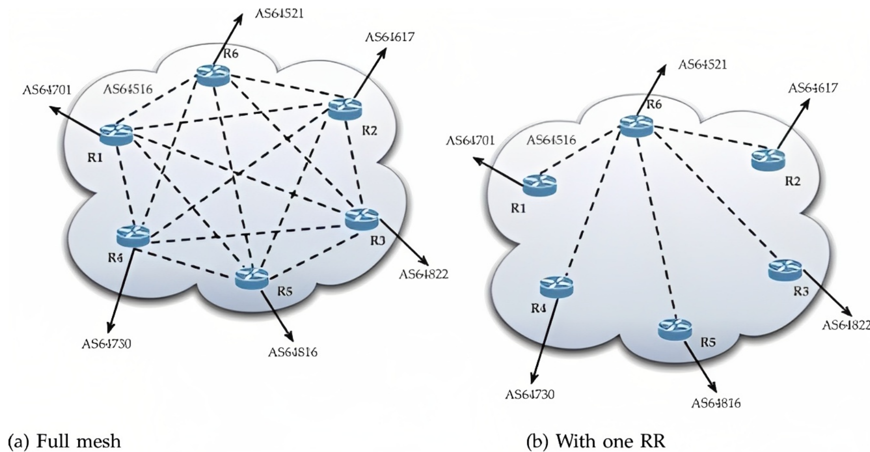


Figura 4: Comparación entre BGP con y sin reflectores de rutas [22].

El protocolo BGP, dentro de su estándar, cuenta con ciertas reglas para la selección de la mejor ruta. Acorde al RFC 4271, el protocolo BGP realiza un proceso de decisión que consta de tres fases. En la primera fase, se calcula el grado de preferencia para cada ruta recibida de un vecino BGP que se encuentra en la base de datos Adj-RIBs-In (*Adjacent Routing Information Base In*, la información no editada enviada por los vecinos BGP). En la segunda fase se escoge la mejor ruta de las disponibles obtenidas en la fase anterior y de instalar esas rutas en la base de datos local Loc-RIB (*Local Routing Information Base*). Finalmente, la tercera fase se encarga de esparcir la información del Loc-RIB hacia los vecinos BGP, realizando agregados de prefijos o reducción de información de ser posible o necesario. Siguiendo estas fases, lo normal en una configuración estándar es que los primeros criterios de decisión para determinar la ruta preferida es el *Local Preference* primero y luego el *AS Path*; la forma en que cada una de estas fases se desarrolla puede variar ligeramente según el proveedor que lo implemente, por ejemplo Cisco agrega un criterio sobre todos los demás que se llama *weight*, además existen muchos otros parámetros para realizar la decisión de la mejor ruta [23].

6.1.5. Arquiectura de red

Existen muchas topologías que se utilizan para diseñar la red *core* de un ISP, pero como se puede ver en la Figura 5, una topología de una red de un ISP puede consistir de un anillo

terrestre, el cual consiste de varios *routers* conectados entre sí formando un anillo con el fin de tener redundancia en sus enlances por si en algún momento alguno de ellos se cae. Luego, de uno o varios de los *routers* del anillo salen capacidades submarinas (en el caso de que sea necesario hacer una conexión que pase por el mar, el cual es el caso de Guatemala) hacia el NAP, en donde se puede hacer la interconexión con otros ISP como Tiers 1. Siguiendo en el contexto de BGP, en este caso se tiene que en el anillo terrestre se compartes las rutas a través de un reflector de rutas, por lo que los *routers* levantan sesiones iBGP hacia este relfector; además de esto, los Tier 1 son otros sistemas autónomos, por lo que se levantan sesiones de eBGP hacia ellos en el NAP.

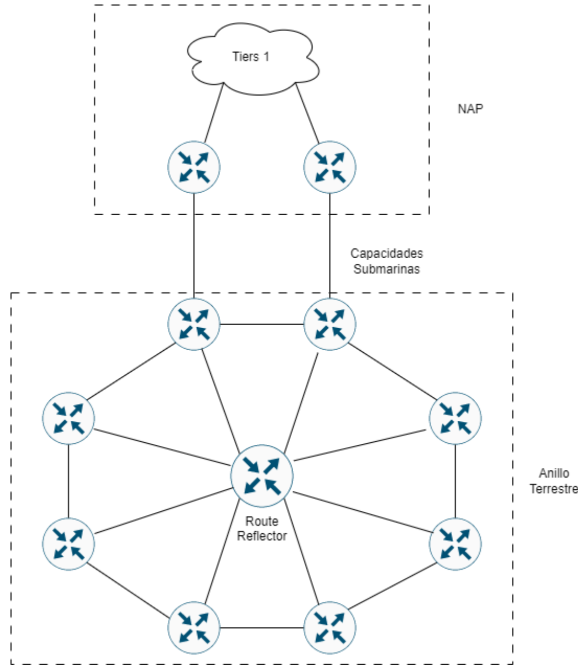


Figura 5: Topología de red de un ISP con conexión a un NAP para lograr salir a Internet.

Como se mencionó antes, en el caso de Guatemala, sí es necesario realizar la conexión al NAP a través de una capacidad submarina. Las capacidades submarinas que Guatemala usa pueden observarse en la Figura 6.

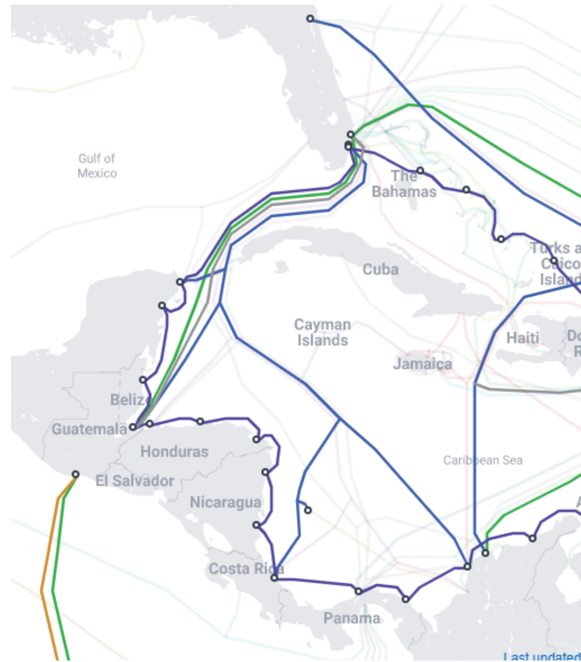


Figura 6: Mapa con las capacidades submarinas para la conexión a Internet conectadas a Guatemala [24].

6.2. APIs

Anteriormente se habló sobre los modelos OSI y TCP/IP, donde se ve que existe en ambos modelos una capa de aplicación, la cual es la que interactúa con los aplicaciones que están corriendo en los *hosts* que se encuentran en el *edge* de la red. Las aplicaciones que se corren en los *hosts* pueden seguir dos arquitecturas o paradigmas principales: cliente-servidor o *peer-to-peer*. En este trabajo se concentrará en el paradigma de cliente-servidor, el cual tiene un diseño donde existe un *host* que es el servidor de la aplicación y otro *host* que consume dicha aplicación. Un ejemplo claro de esta arquitectura son los servidores web, los cuales están corriendo los programas necesarios para que la página web esté levantada para que un cliente pueda entrar desde un buscador y entrar a la página y consumir la información que el servidor ofrece.

Sobre esta lógica nacen las APIs o *Application Programming Interface*, la cual es una forma en que un cliente puede acceder a información de un servidor. Sin embargo, las APIs van más allá de únicamente permitir obtener información del servidor, las APIs permiten correr una función de forma remota en el servidor, dando así la capacidad de hacer uso de las aplicaciones corriendo en los servidores y sus diferentes funciones sin necesidad de tener que programarlas en el cliente [25]. Existen varios tipos o arquitecturas de diseño de APIs que aprovechan diferentes protocolos para realizar su comunicación, como REST, RPC, SOAP, WebSocket y GraphQL, sin embargo, en este trabajo se estará concentrando en REST y RPC, específicamente gRPC, que es una API RPC desarrollada inicialmente por Google.

6.2.1. REST

REST no es un protocolo o un estándar, sino un conjunto de restricciones para el diseño de arquitectura de una API. Esta arquitectura tiene 6 restricciones: la arquitectura de aplicación debe de ser de cliente-servidor y debe de utilizar el protocolo HTTP, la comunicación cliente-servidor debe de ser *stateless*, la respuesta obtenida puede ser *cacheable* o temporalmente almacenable, una interfaz uniforme entre componentes, una arquitectura del sistema en capas y la capacidad de tener código bajo de manda (aunque esta última restricción es opcional) [26].

Ampliando más en estas restricciones, para la arquitectura de cliente-servidor, las aplicaciones que se están comunicando deben de ser totalmente independientes entre sí, donde la única información que conoce una aplicación de la otra es el URI o *Uniform Resource Identifier*. Por otro lado, cuando se dice que la comunicación debe de ser *stateless* o sin estado se refiere a que la información del cliente no es guardada entre peticiones y cada petición al servidor es un evento aislado. Luego, si la información obtenida en las respuestas es *cacheable* significa que esta puede ser reusada durante cierto tiempo por si alguna petición similar se ejecuta; además, si la respuesta es *cacheable* o no debe ser indicado por la misma respuesta. Además de esto, que se necesite una interfaz uniforme entre componentes quiere decir que para las solicitudes del mismo recurso a través de una API REST deben de tener el mismo formato independientemente de cualquier situación. Después, tener una arquitectura del sistema en capas quiere decir que las APIs REST deben de diseñarse de tal forma que ni el cliente ni el servidor saben si se comunican directamente entre sí o si existe algún intermediario. Finalmente, tener la capacidad del código bajo demanda quiere decir que existe la posibilidad de que la información que se envía en la petición puede no solamente ser estática, sino que también código ejecutable; en estos casos, dicho código debe de ejecutarse únicamente bajo demanda. Si el diseño de arquitectura de una API cumple con estos 6 requerimientos, significa que la API es RESTful [27].

Puesto que para las APIs REST es necesario que la comunicación sea por HTTP, cuando se utiliza alguna se debe de levantar primero un servidor web, por lo que en este trabajo se mencionará que se utilizan diferentes servidores web sobre los cuales se montan las APIs REST que se consumirán.

6.2.2. gRPC

El diseño de arquitectura de APIs RPC o *Remote Procedure Call* fue uno de los primeros diseños que existió y tiene un concepto bastante sencillo: ejecutar funciones del servidor a través de una petición remota del cliente. Este diseño es bastante flexible en cuanto a formato y a protocolo de comunicación, por lo que algunos diseños de RPC han sido XML-RPC y JSON-RPC, en los cuales la petición se realizaba por medio de HTTP y la información de respuesta llegaba en formato XML o JSON respectivamente [28]. Este diseño fue luego reemplazado por las APIs REST para casi toda aplicación debido a que las APIs REST son más escalables, fáciles de implementar en diferentes *hardwares*, con mejor rendimiento, entre otras ventajas sobre RPC [29].

Esto, sin embargo, cambió con la llegada de gRPC, el cual es un diseño de arquitectura

de APIs RPC desarrollado inicialmente por Google que viene a solucionar las desventajas del RPC original. gRPC, al igual que casi todos los modelos de APIs, define ciertos métodos en el servidor que el cliente puede acceder y ejecutar de forma remota. Este tipo de API, al igual que REST, utiliza HTTP como protocolo sobre el cual monta la información para transportarla, sin embargo, ni el cliente ni el servidor gRPC ven realmente la parte de HTTP, sino que se levantan servidores y clientes gRPC que están montados sobre HTTP/2, creando así un nivel de abstracción para no tener que preocuparse por los métodos de HTTP ni de levantar el servidor; gRPC internamente hace esa traducción. Estos servidores gRPC pueden levantarse desde diferentes lenguajes de programación, como Python, Ruby, C++, Java, entre otros, haciendo posible que un cliente de Python a través de gRPC ejecute una función de Java en un servidor, lo cual puede verse en la Figura 7. Por otro lado, así como las APIs REST tienden a utilizar JSON como formato para los mensajes, gRPC utiliza un concepto llamado *Protocol Buffers*, los cuales son métodos de serializar la data hasta llevarla a binario para que la transferencia de información sea más rápida, pequeña y sin necesidad de tanto procesamiento del lado del cliente o el servidor [30].

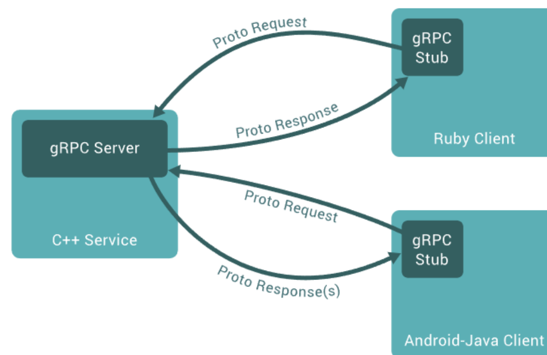


Figura 7: Diseño de APIs gRPC [30].

6.3. Microservicios

Anteriormente se ha hablado de la arquitectura de la red y de los servidores, sin embargo, también es importante hablar de la arquitectura de las aplicaciones que corren en los servidores en sí mismas. Existen muchas arquitecturas, sin embargo, las más importantes en la actualidad son la monolítica, de microservicios y la *serverless*. En este trabajo de graduación se estará enfocando únicamente en la arquitectura de microservicios.

Los microservicios son una arquitectura de aplicaciones que tiene como objetivo lograr modularizar e independizar cada funcionalidad de la aplicación, siendo estos módulos los servicios que por la modularización se vuelven muy pequeños y fáciles de modificar y reparar. Estos pilares en los que se basa esta arquitectura permite que las aplicaciones sean fáciles de escalar, donde en lugar de escalar toda la aplicación (como lo sería con una arquitectura monolítica), se puede escalar únicamente el componente o microservicio que lo requiere. Asimismo, esta arquitectura permite poder exponer ciertos componentes que se deseen para uso externo, como un servidor web que levanta una API REST para que los usuarios finales puedan acceder a la información que tiene el servidor de una forma rápida y sencilla, dando

también así capacidad para automatización de terceros, como puede verse en la Figura 8 [31].

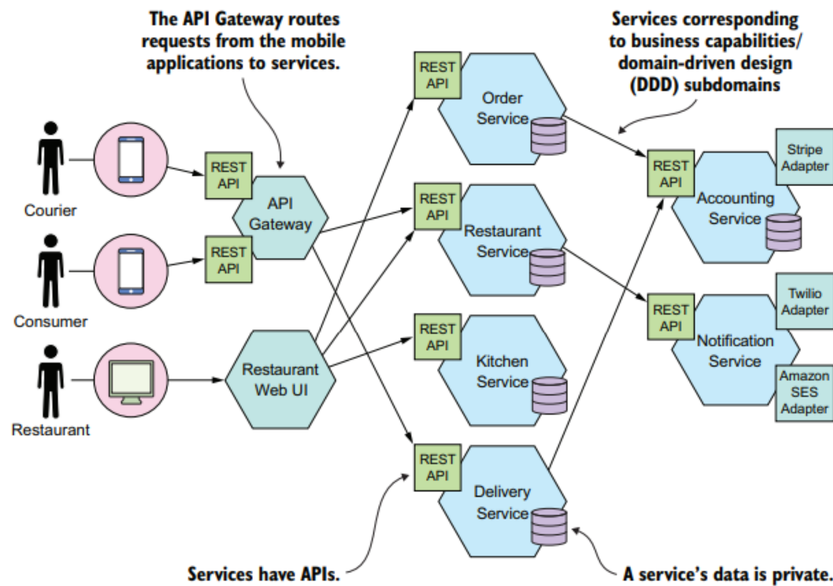


Figura 8: Ejemplo de una arquitectura de microservicios [31].

6.3.1. Virtualización

Luego de hablar de la arquitectura de las aplicaciones, surge un tema importante que es físicamente dónde están corriendo estas aplicaciones, es decir, la infraestructura. Originalmente, las aplicaciones corrían en un servidor dedicado a eso y cuando se necesitaba escalar la aplicación se tenía que comprar otro servidor físico (conocido como escalamiento horizontal) o agregar más recursos al servidor existente, como la memoria RAM (conocido como escalamiento vertical). Este tipo de soluciones eran bastante caras, difíciles de implementar y no se lograba hacer un buen uso de los recursos existentes, puesto que normalmente es solamente algunas partes de la aplicación que requieren más recursos; en el caso de una aplicación monolítica si se tenía que escalar una parte de la aplicación no se podía realizar de forma aislada, sino que tenía que escalarse todo; por otro lado, con microservicios sí se puede escalar solamente una parte o microservicio, sin embargo, no es rentable comprar un servidor completo para un solo microservicio.

Algo que soluciona este tipo de problemas es una idea que viene desde hace mucho tiempo llamado virtualización. La virtualización es un proceso de utilizar *hardware* de una forma más eficiente a través de la utilización de *software* para crear un nivel de abstacción de dicho *hardware* para que pueda ser dividido en múltiples *hardwares* virtuales, a lo que se le conoce como máquinas virtuales o VMs por sus siglas en inglés. Esta idea nació por la necesidad de dar múltiples accesos de forma simultánea a los recursos de una computadora a varias personas, por lo que se necesitaba poder tener espacios aislados e independientes para que cada quien pudiera utilizar y en la actualidad coloca las bases para solucionar el problema de rendimiendo, escalabilidad y costo antes mencionado, puesto que permite la creación de

máquinas virtuales que contengan cada una de las partes de la aplicación. Esta solución permite que si un módulo de una aplicación comienza a necesitar más recursos, puede escalarse solamente ese elemento otorgándole más recursos de la máquina física, mientras que los otros componentes pueden quedarse igual [32].

6.3.2. Contenedores

Así como las máquinas virtuales antes mencionadas solucionan el problema de la escalabilidad, rendimiendo y costos de infraestructura, los contenedores resuelven estos mismos problemas pero al nivel de aplicación (en la Figura 9 puede observarse una comparación entre las dos soluciones). Los contenedores también son una forma de virtualización, similar a las máquinas virtuales, con la diferencia que los contenedores comparten *kernel* con la máquina sobre la cual están corriendo. Esto quiere decir que un contenedor que usa un sistema operativo basado en Linux tiene que levantarse en una máquina (ya sea física o virtual) que esté corriendo un sistema operativo basado en Linux. Puesto que el *kernel* de un sistema es lo más pesado, esta idea permite a los contenedores ser bastante ligeros en comparación a las máquinas virtuales, las cuales contienen su propio sistema operativo con *kernel*. Por esto mismo, los contenedores requieren de menos recursos para levantarlos, correrlos y adminstrarlos. A su vez, los contenedores ofrecen portabilidad a las aplicaciones, puesto que un problema que siempre han enfrentado los programas es que funcionan bien en un ambiente, es decir, un sistema operativo específico, con una versión específica del lenguaje en que fue programado y con librerías específicas, por lo que muchas veces sucede que un programa se dice que ya funciona y al pasarlo a otro sistema no hace lo esperado [33]. Puesto que los contenedores son sistemas aislados y fáciles de levantar, los aplicativos pueden crearse en ambientes específicos que luego pueden ser levantados en un contenedor para que corra sobre cualquier otro ambiente.

Esta idea de los contenedores es utilizada por los microservicios, ya que cada microservicio puede ser un contenedor diferente, el cual es fácil de levantar, y por ende escalar, independiente del resto de componentes en el sistema y ligero. En la actualidad, muchas aplicaciones están corriendo en contenedores que a su vez están corriendo en máquinas virtuales que a su vez están corriendo en servidores físicos. De esta forma, las aplicaciones son fáciles de levantar, eficientes y portables, mientras que la infraestructura es utilizada de forma eficiente y fácil de administrar. Además de esto, los contenedores pueden exponer servicios para ser consumidos, como APIs, por lo que permiten hacer diseños de aplicaciones desacopladas, modulares y escalables, que es justamente en lo que se basan los microservicios.

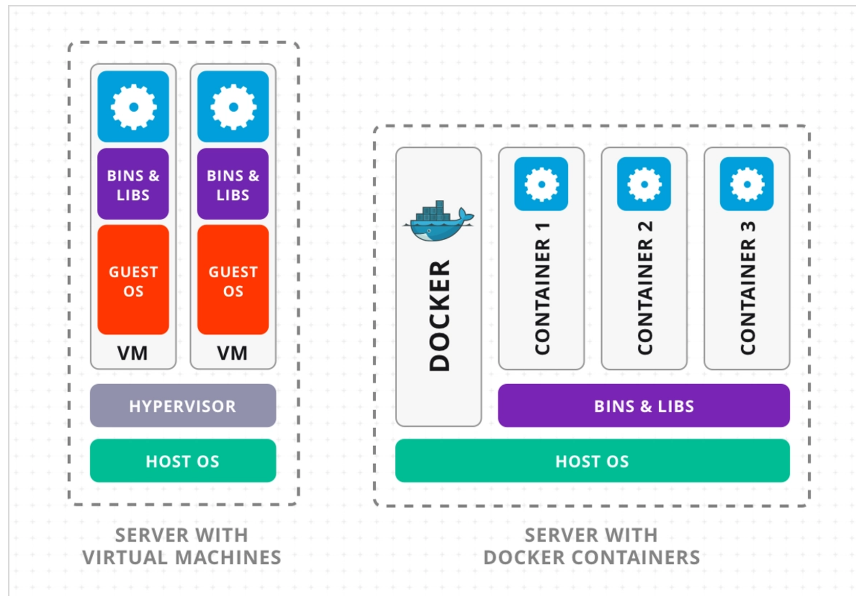


Figura 9: Comparación entre máquinas virtuales y contenedores [34].

6.3.3. Orquestación de contenedores

A pesar de que, como se mencionó antes, levantar y utilizar contenedores es relativamente sencillo, al tener aplicaciones que requieren de muchos contenedores, se vuelve un trabajo complejo el diseño de todos los aspectos de la comunicación interna, identificación de errores, tener persistencia, entre otros aspectos. Aquí es donde entran los orquestadores de contenedores, los cuales son programas que logran automatizar todo el esfuerzo operativo de utilizar contenedores, incluyendo el aprovisionamiento, escalamiento, manejo de las redes de telecomunicaciones internas, balanceo de cargas, entre otros aspectos [35].

A pesar de que existen varias soluciones que ofrecen la orquestación de contenedores, quizás una de las más famosas y utilizadas a nivel mundial y en la que se enfocará este trabajo es Kubernetes, también conocido como k8s. Kubernetes es un orquestador de código abierto que comenzó a ser desarrollado por Google y ahora es un proyecto base para el Cloud Native Computing Foundation. Este orquestador envuelve a los contenedores sobre una capa de abstracción que se conoce como *Pod*, en el cual se encuentra corriendo el módulo o microservicio de la aplicación. A su vez, Kubernetes hace uso de otras abstracciones como Servicios, los cuales permiten acceder a los *Pods*, *Deployments*, los cuales son plantillas de configuración para *Pods*, etc. Kubernetes como orquestador de contenedores es el encargado de [36]:

- Realizar el *rollout* y *rollback* de las aplicaciones en los *Pods*
- Dar un servicio de DNS y DHCP interno para que cada *Pod* y Servicio tenga su propia IP para que todos los componentes puedan verse entre sí
- Administrar el uso del espacio de almacenamiento
- Reparar los componentes de forma automática

- Adminstrar la configuración y el uso de variables de entorno
- Empaquetar binarios automáticamente
- Ejecutar procesos por lotes y administrar cargas de trabajo de *Continuos Integration* (CI)
- Escalar horizontalmente los componentes
- Alocar IPs (tanto IPv4 como IPv6) a *Pods* y Servicios

6.4. ONAP

ONAP u *Open Network Automation Platform* es un proyecto de código abierto del Linux Foundation que enfrenta la necesidad de tener una plataforma común de automatización para aplicaciones de telecomunicaciones, cable y proveedores de servicios de Nube. ONAP es una plataforma diseñada con una arquitectura de microservicios que se comunican internamente y con el exterior con una serie de APIs tanto REST como gRPC para automatizar los diferentes procesos involucrados en el ciclo de vida de servicios, desde su aprovisionamiento hasta la autoreparación de los mismos. En otras palabras, ONAP permite a las organizaciones proveedores de servicios de telecomunicaciones o nube colaborar para la instanciación de elementos de red y servicios de una forma dinámica, con procesos en ciclos cerrados controlados y con respuestas en tiempo real. Además de esto, ONAP da las ventajas que es *vendor-agnostic*, es decir, que funciona de la misma forma sin importar la marca de los dispositivos de red sobre los cuales se está aplicando la automatización [37].

ONAP consta de dos marcos de trabajo o *frameworks*: el *Design Time Framework*, sobre el cual se hace el diseño de los servicios, y el *Run Time Framework*, sobre el cual se realiza el aprovisionamiento del servicio y cualquier operación que el mismo necesite, como puede verse en la Figura 10. En el diseño del servicio se permite la especificación de los servicios en todos los aspectos, modelando todos los recursos y relaciones que los conforman, sus políticas, eventos de ciclo cerrado, entre otros. Por otro lado, el lanzamiento o aprovisionamiento del servicio está construido sobre un marco de trabajo que está basado en políticas para poder automatizar la instanciación de los servicios cuando se necesite, administrando las demandas de los servicios de forma elástica. Finalmente, las operaciones de los servicios se construyen sobre un marco de trabajo que está constantemente monitoreando y analizando el comportamiento de los servicios, basado en el diseño y políticas para poder ejecutar una acción de ser necesario [38].

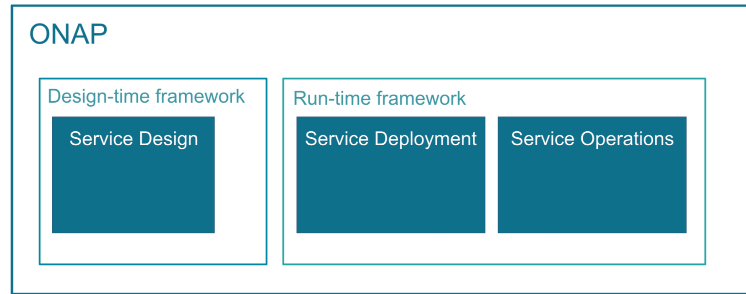


Figura 10: Vista general de las funciones principales de ONAP [38].

6.4.1. Arquitectura

Como se mencionó anteriormente, ONAP es una plataforma diseñada con una arquitectura de microservicios con dos módulos principales, el *Design-Time Framework* y el *Run-Time Framework*. Estos, sin embargo, no son los microservicios; como puede verse en la Figura 11, cada uno de los módulos principales está separado en muchos módulos más pequeños, los cual aún no son los microservicios, puesto que cada uno de los módulos que se ven en la imagen están compuestos por partes todavía más pequeñas que sí son los microservicios. Como puede verse, ONAP es una plataforma muy compleja y, por lo mismo, se vuelve clara la necesidad de utilizar Microservicios para diseñarla, ya que esto da la capacidad de solamente utilizar algunos módulos de la plataforma sin necesidad de gastar recursos en algo que no va a ser utilizado. Además de esto, como se ha mencionado varias veces al hablar de microservicios, esta arquitectura permite hacer un escalamiento sencillo e independiente de cada microservicio bajo demanda, por lo que permite utilizar los recursos de los servidores de forma mucho más eficiente que si no se utilizara este diseño.

Muchos de los módulos que ofrece ONAP están enfocados a la automatización de servicios que utilizan funciones de red virtualizadas o VNFs por sus siglas en inglés, es decir, *routers*, *switches*, *firewalls*, entre otros. Sin embargo, este proyecto está enfocado en la automatización de dispositivos de red físicos, o PNFs por sus siglas en inglés, por lo que en este trabajo de graduación se estará enfocando únicamente en los módulos *Active & Available Inventory* (A&AI), *Controller Design Studio* (CDS) y *Service Orchestration* (SO).

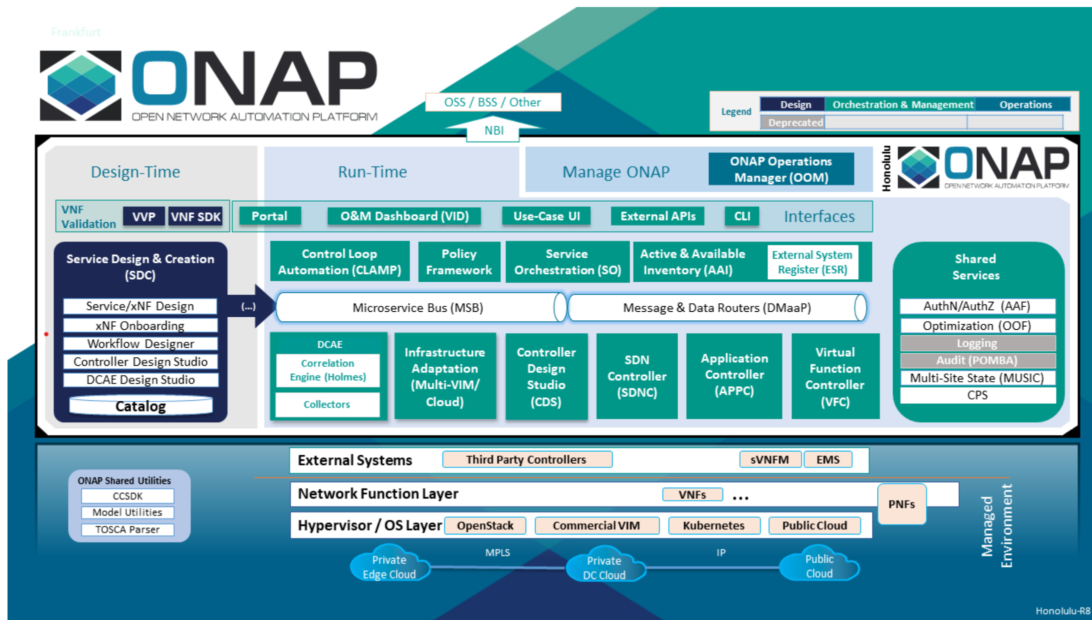


Figura 11: Arquitectura de alto nivel de ONAP [39].

6.4.2. A&AI

La A&AI es el módulo en donde se inventaría toda la información de la red en vivo y con todas las relaciones que los componentes tienen entre sí [40]. La A&AI tiene en su arquitectura, como se ve en la Figura 12, una base de datos Cassandra, la cual es una base de datos NoSQL, específicamente de grafos. Esta base de datos está diseñada de esta forma porque los elementos de red inventariados en la A&AI pueden tener numerosas relaciones entre sí, lo cual, si se deseara hacer con una base de datos relacional, tendrían que utilizarse muchas tablas intermedias para realizar relaciones *many-to-many* y no sería escalable. Además de la base de datos, este módulo cuenta de varios microservicios y librerías, sin embargo, las escenciales y utilizadas en este proyecto son:

- aai/resources: es un microservicio que levanta un servidor web para tener una API REST que da capacidades para operaciones CRUD (*Create*, *Read*, *Update* y *Delete*) de los recursos inventariados.
- aai/graphadmin: es un microservicio que permite la administración de los grafos almacenados en la base de datos.
- aai/schema-service: es una aplicación que guarda y provee versiones del esquema de la base de datos específicos.
- aai/traversal: es un microservicio que permite utilizar la API REST para realizar una búsqueda entre los elementos de red inventariados, navegando entre sí a través de las relaciones que tienen.

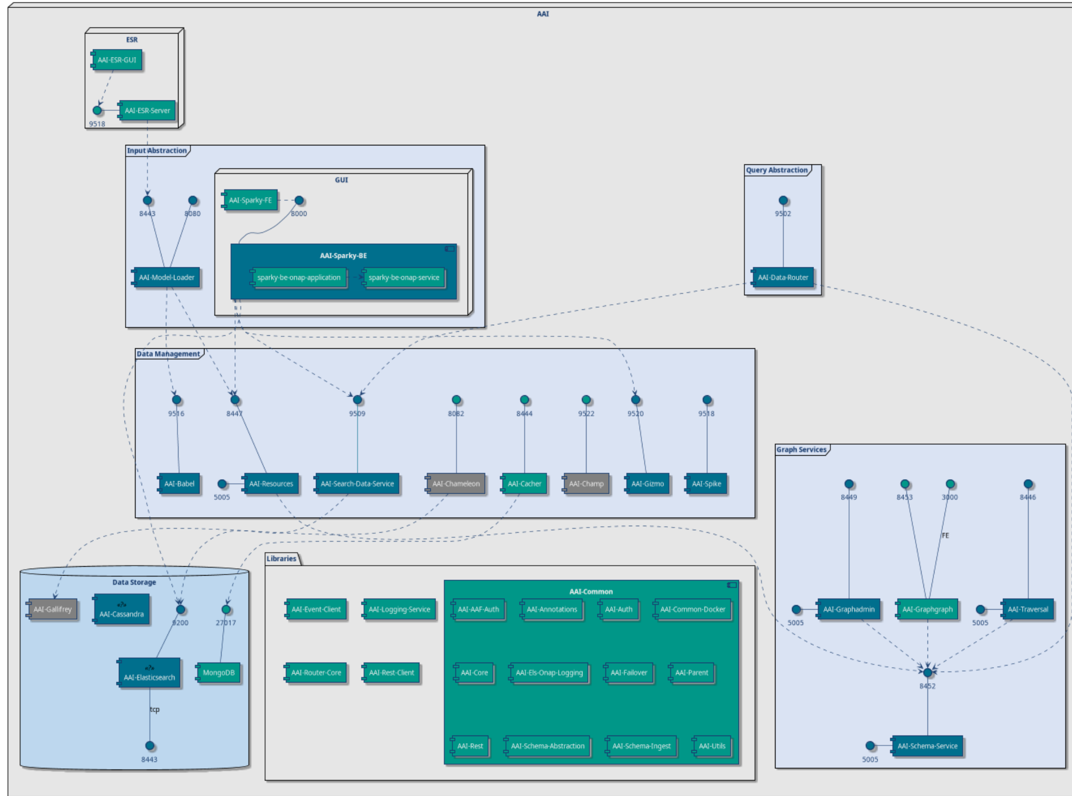


Figura 12: Arquitectura de alto nivel de la A&AI [40].

6.4.3. CDS

El CDS es un módulo diseñado con el propósito de ser *self service*, es decir, que tanto los programadores como los usuarios finales sean capaces de reconfigurar los programas del sistema según los requerimientos del cliente. Este tipo de ideología le permite a los proveedores de servicios una respuesta mejor a los clientes para cumplir con sus demandas. El CDS a grandes razgos está diseñado con un arquitectura de dos grandes módulos: la GUI (*Graphical User Interface*) y el *Run Time*; durante este proyecto se estará utilizando únicamente la parte del *Run Time*, el cual tiene como objetivo crear y llenar planos (*blueprints*) para que los controladores los utilicen para crear de forma automática archivos de configuración para aplicarlos a equipos de red, ya sea físicos o virtuales. La arquitectura del CDS puede observarse en la Figura 13, donde se ve que en el *Designer* es donde se crean los planos y el *Runtime* es donde se procesan los flujos automatizados siguiendo al diseño de los planos [41]. Además de esto, es importante mencionar que la forma en que los archivos de configuración creados son luego aplicados a los equipos de red es a través de los microservicios ejecutores que se ven, es decir, el C++ *Executor*, Python *Executor*, Kotlin *Executor* y GO *Executor*; estos microservicios, como se ve en el diagrama, se comunican de forma interna con los demás microservicios del CDS a través de APIs gRPC. Cabe mencionar también, que en este proyecto se estará utilizando el Python *Executor* únicamente.

La forma en que el CDS logra cumplir con su propósito de ser *self service* se basa en los CBAs (*Controller Blueprint Archive*), el cual es un archivo comprimido que contiene la

información necesaria para definir a los flujos de automatización, los programas a ejecutarse, las plantillas que se utilizan, entre otros aspectos que conforman a un plano. El CBA sigue el estándar TOSCA (*Topology and Orchestration Specification for Cloud Applications*) diseñado por la compañía OASIS, en el cual se especifica un lenguaje para describir y administrar aplicaciones complejas nativas de la nube con el fin de que sea portable y *vendor-agnostic*, lo cual es justamente lo que se realiza con los planos [42]. El formato que tienen estos CBAs puede observarse en la Figura 14

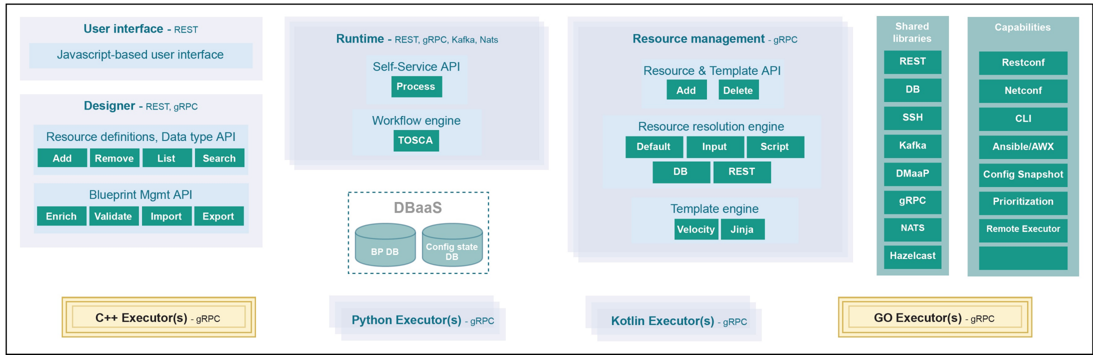


Figura 13: Arquitectura de bajo nivel del CDS [41].

Figura 14: Estructura de un CBA [43].

6.4.4. SO - Camunda

Finalmente, el SO es el módulo encargado de ejecutar flujos de alto nivel para poder realizar una automatización mucho más compleja. El SO es un microservicio del aplicativo Camunda, el cual es un orquestador de procesos Open Source que utiliza el estándar BPMN 2.0 y es utilizado para lograr automatizar la ejecución de múltiples procesos, creando un workflow o flujo de trabajo. Camunda usa Java para realizar sus workflows, por lo que la herramienta de Maven, un project manager de Java, es necesaria para hacer el build de

los proyectos. Maven es el encargado de tomar el código de Java utilizado por Camunda y los XML generados y lo construye para generar un archivo .war. Los archivos .war o Web Application Archive, son archivos comprimidos que contienen archivos de Java y de Markup Language para generar aplicativos web; en este caso, el archivo war generado por Maven usando el código de Camunda es copiado en el pod dedicado a Camunda para poder levantar un servidor web con los endpoints y métodos de la API que acciona a los wokflows. Este servicio web es el que permite el acceso a los flujos desde aplicaciones externas.

Esta plataforma se conforma de varias partes, entre las cuales las más utilizadas en este proyecto son Camunda Modeler y Camunda Cockpit. El Camunda Modeler es la interfaz en la cual se pueden realizar los diagramas, como se puede ver en la Figura 15, y que general el código XML que describe con código lo que se ve de forma gráfica. Esta interfaz puede ser utilizada en línea y de forma colaborativa con la licencia de Camunda para empresas o de forma local gratuitamente. Por otro lado, el Cockpit es una interfaz que es utilizada para monitorear los deployments, procesos y errores, como se ve en la Figura 16. Esta herramienta se puede observar en la página web que se levanta al realizar el proyecto de Maven para Camunda, en donde también se encuentran el Tasklist y el Admin.

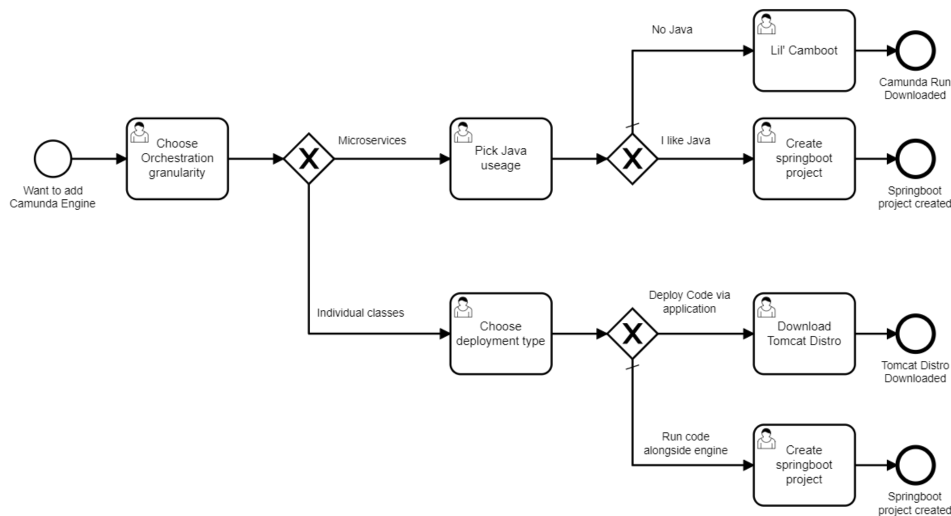


Figura 15: Ejemplo de un flujo siguiendo el estándar BPMN2.0 con Camunda Modeler [44].

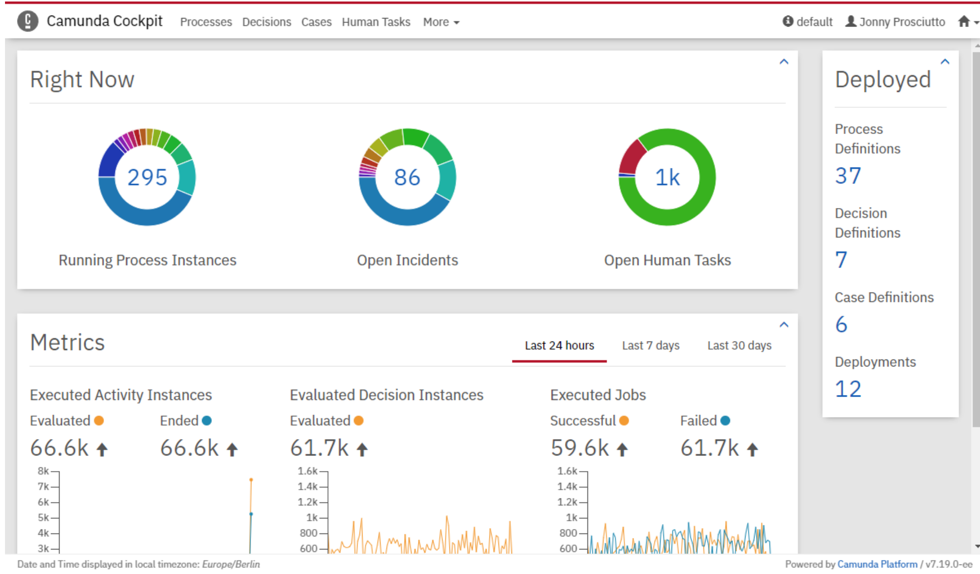


Figura 16: Camunda Cockpit [45].

CAPÍTULO 7

Prerequisitos para el desarrollo del proyecto en la plataforma ONAP

En este trabajo de graduación se propone desarrollar un caso de uso para la automatización del balanceo de tráfico de internet a través de la plataforma ONAP. Para esto, primero se crearon los proyectos en la plataforma ONAP para poder desplegar los flujos de Camunda y los CBAs del CDS. En este capítulo se detallan los pasos que se siguieron para la creación y el despliegue de estos proyectos. De igual forma, se muestra los prerrequisitos que tienen dichos proyectos para poder funcionar de forma adecuada.

7.1. Prerequisitos de Camunda

Puesto que ONAP es una plataforma Cloud Native, Camunda se despliega en un *pod* de Kubernetes como uno de los microservicios que conforman la plataforma, por lo que, como primer requisito para poder crear flujos en Camunda, debe de verificarse que los *pods* de Kubernetes estén funcionando de forma adecuada. Para esto, se debe de ejecutar el siguiente comando con una respuesta similar a la que se muestra a continuación:

```
1 onap@ONAPSCRBSDDEAP02:~$ kubectl -n onap get pod | grep camunda
  dev-camunda-695f4df4b6-ffdfl          1/1     Running         0          73d
  dev-camunda-695f4df4b6-gr87m          1/1     Running         0          73d
  dev-camunda-695f4df4b6-mbhb4          1/1     Running         0          73d
  dev-camunda-gui-84478994d4-b66b9      1/1     Running         0          130d
```

Código 7.1: Comprobación de la existencia de los *pods* de Camunda

En este caso, se ven tres *pods* (también conocidos como réplicas) de Camunda porque se busca tener alta disponibilidad en el sistema. Por otro lado, se ve que hay un *pod* que corresponde a la GUI o interfaz gráfica, que en la Subsección 6.4.4 se mencionó como Camunda Cockpit.

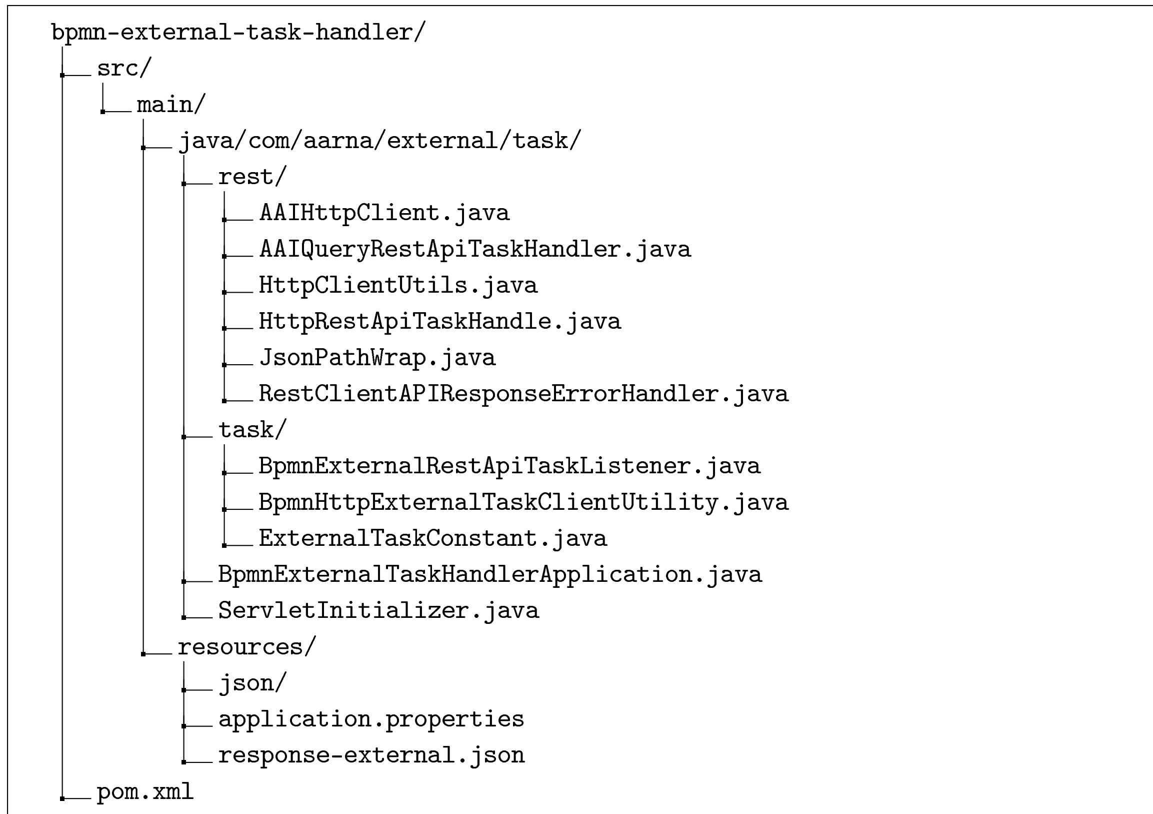
Luego de verificar que los *pods* de Camunda estén funcionando de forma adecuada, se debe de verificar la existencia de los proyectos que se utilizan de forma comunal para cualquier caso de uso que quiera implementarse: bpmn-external-task-handler y bpmn-commons.

El proyecto de bpmn-commons es utilizado como una librería, la cual contiene funciones útiles que cualquier flujo de Camunda puede llegar a necesitar. Un ejemplo claro de esto es el archivo `JsonPathWrap.java`, el cual contiene la funcionalidad para poder tomar una plantilla de un *payload* para una API REST y sustituir los valores parametrizados con aquellos encontrados en la ejecución del flujo de Camunda. La estructura de directorios de este proyecto puede observarse en el Cuadro 1.

```
bpmn-commons/
├── src/
│   └── main/
│       ├── java/com/aarna/bpmn/common/
│       │   ├── constant/
│       │   │   └── ProcessCommonConstants.java
│       │   └── util/
│       │       ├── rest/
│       │       │   ├── AAIHttpClient.java
│       │       │   ├── KentikHttpClient.java
│       │       │   ├── PortalHttpClient.java
│       │       │   └── RestClientAPIResponseErrorHandler.java
│       │       ├── JsonPathWrap.java
│       │       ├── PayloadConstructUtils.java
│       │       ├── ResponsePayloadParseUtils.java
│       │       └── SrchFields.java
│       └── resources/
│           └── application.properties
└── pom.xml
```

Cuadro 1: Árbol de directorios y archivos del proyecto bpmn-commons

Por otro lado, Camunda cuenta con múltiples formas de ejecutar código: con *Scripts*, ya sea *Inline*, que van dentro del mismo archivo BPMN, o *External Resource*, los cuales son referenciados en otra carpeta del proyecto; con *Delegate Expressions*, las cuales permiten agregar código a los procesos dentro del BPMN a través de la asignación de una clase de Java; y con *External Tasks*, las cuales permiten la ejecución de un proceso dentro del BPMN a través de un aplicativo externo. Esta última forma de ejecutar código es la que permite el proyecto de bpmn-external-task-handler, el cual es un aplicativo en Java que se encarga de ejecutar los *External Tasks* definidos en los BPMNs. La estructura de directorios de este proyecto se puede observar en el Cuadro 2.

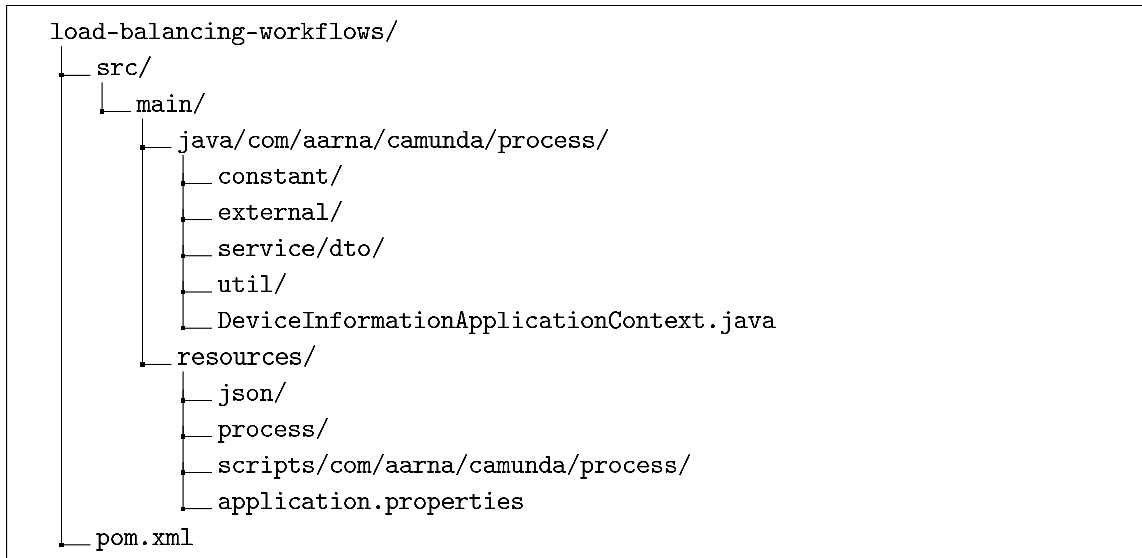


```
bpmn-external-task-handler/
├── src/
│   └── main/
│       ├── java/com/aarna/external/task/
│       │   ├── rest/
│       │   │   ├── AAIHttpClient.java
│       │   │   ├── AAIQueryRestApiTaskHandler.java
│       │   │   ├── HttpClientUtils.java
│       │   │   ├── HttpRestApiTaskHandle.java
│       │   │   ├── JsonPathWrap.java
│       │   │   └── RestClientAPIResponseErrorHandler.java
│       │   ├── task/
│       │   │   ├── BpmnExternalRestApiTaskListener.java
│       │   │   ├── BpmnHttpExternalTaskClientUtility.java
│       │   │   └── ExternalTaskConstant.java
│       │   ├── BpmnExternalTaskHandlerApplication.java
│       │   └── ServletInitializer.java
│       └── resources/
│           ├── json/
│           ├── application.properties
│           └── response-external.json
└── pom.xml
```

Cuadro 2: Árbol de directorios y archivos del proyecto bpmn-external-task-handler

7.2. Creación del proyecto de Camunda

Una vez se verificó que los proyectos mencionados en los prerrequisitos ya están funcionando, se creó el árbol de directorios y archivos para el proyecto de Camunda en donde está el funcionamiento del caso de uso propuesto en este trabajo de graduación, el cual se muestra en el Cuadro 3.



Cuadro 3: Árbol de directorios y archivos del proyecto load-balancing-workflows

La funcionalidad principal de los procesos se incluye en la carpeta main/.../process/. Dentro de esta carpeta, se encuentra la carpeta constant/, en la cual se agregan los *scripts* de Java que se encargan de tener todas las constantes utilizadas de forma concurrente en el resto de archivos que dan la funcionalidad de los flujos de Camunda. Por otro lado, en la carpeta external/ se agregan los *scripts* de Java que se encargan de preparar la información antes de utilizar un *External Task* y de procesar la información que estos devuelven luego de ser ejecutados. Luego, la carpeta service/dto/ se utiliza para agregar achivos en donde se definen los DTOs o *Data Transfer Objects*, los cuales son utilizados para serializar datos, convirtiéndolos del formato JSON con el que llegan a Strings. Además, la carpeta util/ se utiliza para agregar archivos de utilidades múltiples que son utilizados por los *scripts* de Java de la carpeta external/ con el fin de agregar modularización y abstracción en el código. Finalmente, el archivo `DeviceInformationApplicationContext.java` es utilizado para declarar todas las clases de Java que pueden ser luego instanciadas como *Delegate Expressions*.

Por otro lado, en la carpeta main/resources/ se encuentran archivos que también son importantes para darle la funcionalidad completa a los flujos desarrollados. En la carpeta json/ se encuentran todas las plantillas de los *payloads* para llamar a las APIs, tanto del CDS como de Kentik. En la carpeta process/ se encuentran todos los archivos bpmn creados en el Camunda Modeler, en donde se definen los flujos. Finalmente, en la carpeta scripts/.../process/ se encuentran los *scripts* utilizados para los *External Resource Scripts*.

7.3. Despliegue del flujo en Camunda

Una vez ya se tienen los tres proyectos de Java, se necesita desplegarlos en los *pods* de Camunda mostrados en los prerrequisitos, para que estos puedan levantar los proyectos y se puedan utilizar. Este es un proceso manual y tedioso, por lo que se creó el *script* de Bash visto en el Código 7.2 que automatiza el proceso, el cual supone que el usuario que lo ejecuta

se encuentra en la carpeta padre de en donde se encuentra el repositorio camunda-workflows, dentro del cual se encuentran los tres proyectos antes mencionados.

```
#!/bin/bash
# bpmn-adder.sh
Main()
{
  cd camunda-workflows
  cd bpmn-commons
  mvn clean install
  cd ../load-balancing-workflows
  mvn clean install
  for i in {1..3}
  do
    POD=$(kubectl get pods -n onap | grep 'camunda' | grep Running | awk 'FNR=='$i'
        {print $1}')
    kubectl -n onap -c camunda cp \
      ./target/load-balancing-workflows.war $POD:/camunda/bpmnapps/.
  done
}
External()
{
  cd camunda-workflows
  cd bpmn-commons
  mvn clean install
  cd ../bpmn-external-task-handler
  mvn clean install
  for i in {1..3}
  do
    POD=$(kubectl get pods -n onap | grep 'camunda' | grep Running | awk 'FNR=='$i'
        {print $1}')
    kubectl -n onap -c camunda cp \
      ./target/bpmn-external-task-handler.war $POD:/camunda/bpmnapps/.
  done
}
while getopts ":me" option; do
  case $option in
    m)
      Main
      exit;;
    e)
      External
      exit;;
    \?) # incorrect option
      echo "Error: Invalid option. Use -h for help"
      exit;;
  esac
done
```

Código 7.2: *Script* de automatización para cargar los proyectos de Java a los *pods* de Camunda

Este *script* se encarga de utilizar la herramienta de CLI de Maven para compilar los proyectos de Java y luego copiar los archivos .war generados a los *pods* de Camunda. Como puede verse, el *script* tiene dos funciones: **Main** y **External**. La función **Main** se encarga de compilar los proyectos bpmn-commons y load-balancing-workflows, mientras que la función **External** se encarga de compilar el proyecto de bpmn-commons y el proyecto bpmn-external-task-handler. En ambos casos, luego de realizar la compilación, se copia el archivo .war generado a los *pods* de Camunda; en el caso de la función **Main**, solamente se copia el archivo .war generado del proyecto load-balancing-workflows, ya que dentro del archivo pom.xml se define que el proyecto bpmn-commons se utiliza como dependencia, lo cual es interpretado por Maven, quien se encarga de incluir el archivo .jar generado del proyecto bpmn-commons dentro del archivo .war generado del proyecto load-balancing-workflows, por

lo que al subir este proyecto a los *pods* ya se incluyen ambos proyectos. Como se mencionó anteriormente, para que el proyecto de Camunda creado funcione, es necesario contar con los otro dos proyectos, por lo que ambas opciones del *script* deben de ser corridas aunque sea una vez y cada una debe de correrse nuevamente cada vez que cada proyecto tenga cambios. El resultado de la ejecución del *script* debe de verse en los *logs* de los *pods* de Camunda de forma similar a lo que se muestra en el Código 7.3, en donde se ve que los procesos de Camunda se suben de forma exitosa.

```
onap@ONAPSCRBSDDEAP02:~$ bash bpmn-adder.sh -m
...
onap@ONAPSCRBSDDEAP02:~$ kubectl -n onap exec -it \
  $(kubectl get pods -n onap | grep 'camunda' | awk 'FNR==3 {print $1}') \
  -- tail -fn 2 ../../../camunda/logs/catalina.2023-08-27.log
...
27-Aug-2023 06:38:22.629 INFO [Catalina-utility-2]
    org.camunda.commons.logging.BaseLogger.logInfo ENGINE-08023 Deployment summary
    for process archive 'traffic-load-balancing':

    process/CONFIGURATION-EXECUTION.bpmn
    process/TIGO-COMPONENTS-DMN-LOAD-BALANCING.dmn
    process/CONFIGURE-LOAD-BALANCER.bpmn
    process/TRAFFIC-LOAD-BALANCER.bpmn
...
```

Código 7.3: *Logs* esperados luego de subir los proyectos de Java a Camunda

7.4. Prerequisitos del CDS

Similar a Camunda, el CDS también se despliega en un *pod* de Kubernetes como uno de los microservicios que conforman la plataforma. Por esto, debe de verificarse que los *pods* de Kubernetes estén funcionando, para lo cual se ejecuta el siguiente comando con una respuesta similar a la que se muestra en el Código 7.4. Como se mencionó en la Subsección 6.4.3, el CDS puede utilizar ejecutores de múltiples lenguajes, como Python, Kotlin, GO, entre otros; en este caso se utilizan ejecutores de Python, correspondientes a los *pods* que tienen el nombre "dev-cds-py-executor". Por otro lado, también existen los *pods* que tienen el nombre "dev-cds-blueprints-processor", los cuales se encargan de procesar los CBAs cuando estos se cargan y de exponer una API REST para que estos CBAs se ejecuten de forma remota.

```
onap@ONAPSCRBSDDEAP02:~$ kubectl -n onap get pod | grep cds
dev-cds-blueprints-processor-5b78b5c84d-4prcr   1/1     Running     0      10d
dev-cds-blueprints-processor-5b78b5c84d-5g7pj   1/1     Running     0      10d
dev-cds-blueprints-processor-5b78b5c84d-5m7dq   1/1     Running     0      10d
dev-cds-blueprints-processor-5b78b5c84d-lj2lq   1/1     Running     0      10d
dev-cds-blueprints-processor-5b78b5c84d-vjzzl   1/1     Running     0      10d
dev-cds-py-executor-dc49fdd74-6qdb6             1/1     Running     0      73d
dev-cds-py-executor-dc49fdd74-9dh2k             1/1     Running     0      73d
dev-cds-py-executor-dc49fdd74-dfssc             1/1     Running     0      73d
dev-cds-py-executor-dc49fdd74-grvs9             1/1     Running     0      130d
dev-cds-py-executor-dc49fdd74-mgkh5             1/1     Running     0      130d
```

Código 7.4: Comprobación de la existencia de los *pods* del CDS

7.5. Creación del proyecto en el CDS

Al igual que con Camunda, una vez se verficó que los *pods* del CDS estén corriendo bien, ya puede crearse el proyecto para desarrollar el caso de uso propuesto. En el CDS, sin embargo, no se despliega el proyecto en sí, sino más bien se cargan versiones de plantillas, dentro de las cuales está descrito el funcionamiento de los procesos que se desean. Como se mencionó en la Subsección 6.4.3, a estas plantillas se les conoce también como CBAs. La estructura de directorios y archivos del CBA del proyecto puede observarse en el Cuadro 4.

```
bgp-load-balancer/
├── Definitions/
│   └── bgp-load-balancer.json
├── Scripts/
│   ├── python/
│   │   ├── __init__.py
│   │   ├── aai_service_utility.py
│   │   ├── bgp_load_balancer_executor.py
│   │   ├── capacity_discovery_executor.py
│   │   ├── cba_common_keywords.py
│   │   ├── console_listener.py
│   │   ├── discover_juniper_router_device.py
│   │   ├── discover_router_device.py
│   │   ├── dsl_constants.py
│   │   ├── generic_input_parameter_validator.py
│   │   ├── juniper_router_discovery_service.py
│   │   ├── netconf_constants.py
│   │   ├── query_handler.py
│   │   ├── response_utility.py
│   │   ├── solarwinds_handler.py
│   │   ├── success_response_keys.py
│   │   ├── swisclient.py
│   │   ├── swql_constants.py
│   │   └── url_constants.py
│   └── dd.json
├── Templates/
│   ├── bgp-load-balancer-request-payload.json.template
│   ├── capacity-discovery-request-payload.json.template
│   └── config-existing-term.json.template
├── Tests/
└── TOSCA-Metadata/
    └── TOSCA.meta
```

Cuadro 4: Árbol de directorios y archivos del proyecto bgp-load-balancer

Como se mencionó en la Subsección 6.4.3, los CBAs siguen el estándar TOSCA, por lo que el archivo TOSCA.meta es utilizado por el CBA para poder procesar los archivos dentro del CBA. Este archivo, como puede verse en el Código 7.5, muestra tanto metadata útil del

proyecto, como el creador, etiquetas y la versión, como información indispensable para el procesamiento del proyecto, como la ubicación del archivo que contiene la definición de los flujos del CBA y la versión de CSAR (el cual es utilizado para procesar el CBA).

```
TOSCA-Meta-File-Version: 0.0.1
CSAR-Version: 1.0
Created-By: Ricardo Pellecer
Entry-Definitions: Definitions/bgp-load-balancer.json
Template-Name: bgp-load-balancer
Template-Version: 0.0.1
Template-Tags: bgp-load-balancer
Template-Description: Controller Blueprint to execute the bgp load balancing
```

Código 7.5: Archivo TOSCA.meta del CBA del proyecto

Por otro lado, como se observó en el archvo TOSCA.meta, el archivo que contiene la definición de los flujos del CBA es bgp-load-balancer.json, el cual es un archivo JSON con la estructura que se observa en el Código 7.6. En el campo "metadata" se ingresa, similar al TOSCA.meta, información sobre el autor del CBA, el correo electrónico, el nombre del CBA, la versión y las etiquetas. Luego, en el campo "dsl_definitions" se pueden ingresar macros, los cuales, acorde al estándar TOSCA, son definiciones que pueden ser reutilizadas en otras partes del código haciendo uso de la llave que se coloca de nombre al macro. ONAP aprovecha esta característica del estándar TOSCA para poder realizar conexiones a otros sistemas de forma dinámica y *plug-able*. En este caso, se definen dos macros: "py-executor", el cual se usa para conectarse al *Python Executor*, y "parameters", el cual es utilizado para definir los parámetros que se utilizan en el CBA. Luego, en el campo "workflows" se encuentran los flujos que pueden realizarse en el CBA con todos sus pasos. Finalmente, en el campo "node_templates" se encuentran los nodos que pueden ser utilizados en los flujos definidos en el campo "workflows"; estos nodos son los encargados de ejecutar el flujo.

```
{
    "metadata": {},
    "dsl_definitions": {
        "py-executor": {},
        "parameters": {}
    },
    "topology_template": {
        "workflows": {},
        "node_templates": {}
    }
}
```

Código 7.6: Definición de flujos a utilizar en el CBA

En el caso del macro definido para el *Python Executor*, la configuración que se realiza es para un sistema gRPC. Como puede verse en el Código 7.7, el sistema gRPC necesita de un *host*, puerto y *token* de autenticación para poder acceder a la API y poder ejecutar código de forma remota. Es importante notar que el *host* es obtenido de forma dinámica a través de la utilización de la llave "get_input", la cual es utilizada para obtener la IP del *pod* de Kubernetes en donde se encuentra el *Python Executor*. En la práctica, nunca se ingresa este valor de la IP de forma directa, sino que se utiliza el nombre del servicio asignado a los *pods* del *Python Executor*, ya que Kubernetes se encarga de forma interna con su DNS de relacionar ese nombre con las IPs de todos los *pods* que correspondan a esa aplicación y de realizar el balanceo. Sin embargo, aunque en teoría Kubernetes ya debería de hacer el balanceo, puesto que el *Python Executor* utiliza una API gRPC (la cual utiliza HTTP2),

el balanceo nativo de Kubernetes no logra balancear de forma adecuada los requerimientos hacia los *pods*. Esto se debe a que Kubernetes realiza el balanceo a nivel de conexión TCP y el protocolo HTTP2 utiliza conexiones únicas y de larga duración, lo que significa que solo se realiza una conexión una vez inicia la comunicación, por lo que el balanceo no es posible a menos que se realice de forma lógica al nivel de aplicación o a través de un aplicativo externo como Linkerd o Istio.

```
{
    "py-executor": {
        "type": "token-auth",
        "host": {
            "get_input": "cds-py-exec-pod-ip"
        },
        "port": "",
        "token": ""
    }
}
```

Código 7.7: Configuración del macro para el *Python Executor*

7.6. Despliegue del CBA del CDS

Para realizar el despliegue del CBA del CDS a los *pods* del *Python Executor* se requiere de dos pasos: *Enrichment* y *Save/Deploy*. El *Enrichment* es el proceso a través del cual ONAP vuelve a los CBAs paquetes autosuficientes, embebiendo todas las definiciones de los tipos de objectos que se crearon en el CBA dentro del mismo CBA. Para lograr esto, se utiliza el *script* de automatización que puede verse en el Código 7.8. Este *script* utiliza un *endpoint* de la API REST del CDS, el cual tiene como entrada al CBA en formato .zip y como salida el mismo archivo .zip, pero agregando archivos llamados "node_types.json", "data_types.json", "artifact_types.json" y "resources_definition_types.json". Por otro lado, el *Save/Deploy* es el proceso a través del cual se suben los CBAs al CDS para que ya puedan utilizarse; como puede verse en los códigos 7.8 y 7.9, se realizó un *script* para poder realizar el *Enrichment* y *Save/Deploy* del CBA ya enriquecido respectivamente.

```
#!/bin/bash
zip_file=$1

CDS_BP_POD_NAME=$(kubectl get pods -n onap | grep 'cds-blueprints-processor' | head
    -n 1 | awk '{print $1}')
CDS_BP_SVC_IP=$(kubectl get svc -n onap | grep cds | grep Node | awk '{print $3}')

if [ -z "${CDS_BP_SVC_IP}" ]
    then
        exit 1;
fi

if [ ! -f "$zip_file" ]
    then
    exit 0
fi

rm -rf /tmp/CBA
mkdir -p /tmp/CBA

ENRICHED_CBA_FILE="/tmp/CBA/ENRICHED-CBA.zip"

for i in {1..6}
```

```
do
        POD_IP=$(kubectl -n onap get pod -o wide | grep processor | grep Running |
            awk 'FNR=='$i' {print $6}')
        curl -v --location --request POST
            http://${POD_IP}:8080/api/v1/blueprint-model/enrich \
                --header 'Authorization: Basic Y2NzZGthcHBzOmNjc2RrYXBwcw==' \
                --form "file=@${zip_file}" \
                -o ${ENRICHED_CBA_FILE}
done
echo "You can take a look at the enriched CBA archive file ${ENRICHED_CBA_FILE}"
exit 0
```

Código 7.8: *Enrichment* de los CBAs para poder desplegarlos al CDS

```
#!/bin/bash
zip_file=$1

CDS_BP_POD_NAME=$(kubectl get pods -n onap | grep 'cds-blueprints-processor' | head
    -n 1 | awk '{print $1}')
CDS_BP_SVC_IP=$(kubectl get svc -n onap | grep cds | grep Node | awk '{print $3}')

if [ -z "${CDS_BP_SVC_IP}" ]
    then
        exit 1;
fi

if [ ! -f "$zip_file" ]
    then
    exit 0
fi

for i in {1..6}
do
        POD_IP=$(kubectl -n onap get pod -o wide | grep processor | grep Running |
            awk 'FNR=='$i' {print $6}')
        curl -v --location --request POST
            http://${POD_IP}:8080/api/v1/blueprint-model \
                --header 'Authorization: Basic Y2NzZGthcHBzOmNjc2RrYXBwcw==' \
                --form "file=@${zip_file}" | python3 -m json.tool
done
exit 0
```

Código 7.9: *Save/Deploy* de los CBAs

A pesar de que estos *scripts* ya brindan cierto nivel de automatización para el despliegue de los CBAs, este proceso sigue siendo bastante manual y tedioso, puesto que cada vez que se realiza un cambio en el código, deben de volverse a subir. Por esto, se desarrolló otro *script* de automatización que puede verse en el Código 7.10 y que hace todavía más sencillo el proceso de desplegar CBAs.

```
#!/bin/bash
if [ $# -ne 2 ]
    then
        echo "Required arguments are missing..."
        echo "Usage: $0: <CBA version> <Mount dir path>"
        exit 1
fi

version=$1
mount_dir=$2

echo "################## Changing the version for all CBAs ##################"
./change-versions-all-cbas.sh $version
if [ $? -eq 0 ]
```

```
        then
16          echo "Successfully updated the version for all the CBAs"
    else
            echo "Failed to updated the version for all the CBAs"
            exit 1
    fi
21
    echo "######################## Enriching all the CBAs  ########################"
    ./enrich-all-cbas.sh
    if [ $? -eq 0 ]
        then
26          echo "Successfully Enriched all the CBAs"
    else
            echo "Failed to Enrich all the CBAs"
            exit 1
    fi
31  exit 0
```

Código 7.10: Automatización del proceso de *Enrichment* y *Save/Deploy* de los CBAs

CAPÍTULO 8

Obtención de la información del tráfico a través de las herramientas Kentik y SolarWinds

Luego de cumplir con todos los prerrequisitos mencionados en los capítulos anteriores y con el conocimiento de cómo desplegar los proyectos de Camunda y los CBAs del CDS, ya puede procederse a desarrollar el caso de uso. El primer paso para lograr la automatización del balanceo de tráfico de Internet es obtener la información del tráfico a través de las herramientas Kentik y SolarWinds y utilizar dicha información para poder indicar qué movimientos pueden realizarse para lograr el balanceo del tráfico. Para lograr esto siguiendo el estándar de la plataforma ONAP, deben de crearse los flujos de Camunda y los CBAs del CDS correspondientes. A continuación, se muestra tanto el proceso de creación de estos flujos y CBAs, como la conexión a Kentik y SolarWinds para la obtención de la información pertinente.

8.1. Flujo de Camunda para el descubrimiento de capacidades disponibles

Para poder utilizar el flujo de Camunda, debe de diseñarse el *payload* para la REST API de Camunda, el flujo en Camunda Modeler y la funcionalidad del flujo. A continuación, se muestra el diseño de cada uno de estos elementos.

8.1.1. Diseño del *payload* para la REST API de Camunda

Al realizar el diseño del *payload* que se ve en el Código 8.1 se identificó que se necesitan los siguientes parámetros:

- Nombre de la interfaz: Para poder identificar qué interfaz es la que está saturada.
- Descripción de la interfaz: Esta etiqueta sigue el estándar “IP Transit | CapacidadEnGB | Tier1”, por lo que es necesario obtenerla para poder identificar el Tier 1 al que está asociada la interfaz.
- IP del equipo al que pertenece la interfaz: Para poder identificar de forma única la interfaz, puesto que el NAP cuenta con varios equipos, los cuales pueden tener interfaces que se llamen igual.

Es importante notar que, puesto que el objetivo de este trabajo de graudación es que SolarWinds sea quien ejecute este flujo, todas las variables antes mencionadas pueden ser obtenidas por SolarWinds, es decir, deben de estar en su base de datos. Estos valores que SolarWinds obtiene de forma automática son los que están en el formato `${N=SwisEntity;M=}`. Por otro lado, el resto de variables que se incluyen en el *payload* se utilizan en el CBA que se mostrará y explicará más a detalle más adelante, por lo que el trabajo de Camunda es solamente pasarlos para que cuando llegue a la etapa de ejecutar el CBA, estas variables se llenen en el *payload* y se puedan mandar adecuadamente.

```
{
    "variables": {
        "payload": {
            "type": "String",
            "value": "{\"InterfaceName\":\"${N=SwisEntity;M=InterfaceName}\", \"InterfaceDescription\":\"${N=SwisEntity;M=Alias}\", \"pnf_ipv4_address\":\"${N=SwisEntity;M=Node.IP_Address}\"}"
        },
        "cbaVersion": {
            "type": "String",
            "value": "13.11.2023-6"
        },
        "semaphore": {
            "type": "String",
            "value": "5"
        },
        "connections": {
            "type": "String",
            "value": "2"
        },
        "retry": {
            "type": "String",
            "value": "3"
        },
        "logLevel": {
            "type": "String",
            "value": "info"
        },
        "threshold": {
            "type": "String",
            "value": "0.15"
        },
        "maxMoves": {
            "type": "String",
            "value": "5"
        }
    }
}
```

Código 8.1: *Payload* para la ejecución del flujo de descubrimiento de capacidades

8.1.2. Diseño del flujo en Camunda Modeler

Como puede observarse en el Código 17, el flujo de descubrimiento de capacidades puede separarse en tres etapas: la obtención de información necesaria como prerrequisito, la conexión a Kentik y la ejecución del CBA.

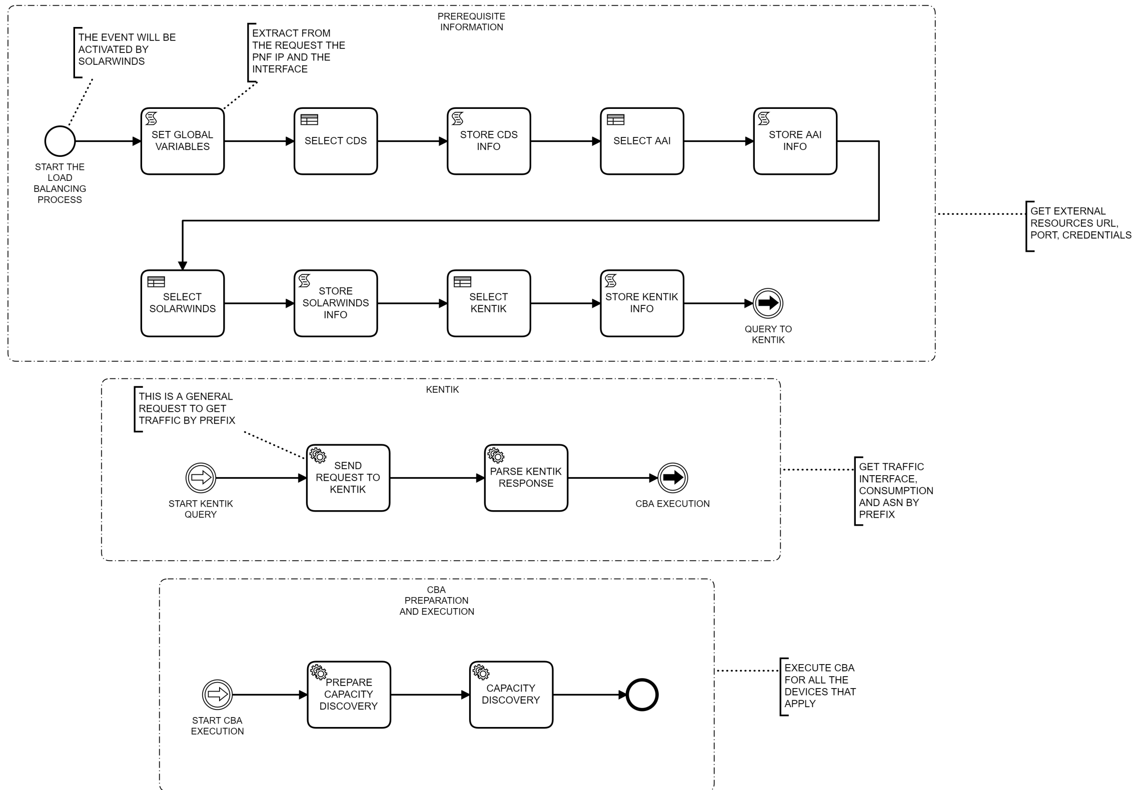


Figura 17: Modelo BPMN del flujo de Camunda para el descubrimiento de capacidades.

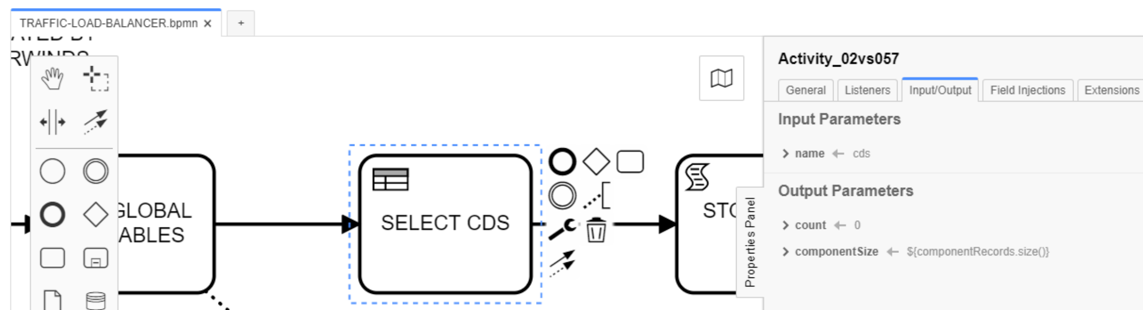
En la primera etapa, se ve que está la figura geométrica de un círculo con contorno simple y delgado, el cual es un evento que indica el inicio del flujo. Luego de esto, se ve un rectángulo también simple y con contorno delgado (que significa un proceso) llamado "SET GLOBAL VARIABLES" con un símbolo de una hoja en la esquina superior izquierda, indicando que es un proceso que tiene una implementación de *Script*, que en este caso es de tipo *Inline*; este proceso se encarga de obtener el JSON de la variable "*payload*" que se observa en el Código 8.1 para poder ser procesada luego, y de mostrar en los archivos de *logs* el valor de las variables pasadas en el POST hacia la API. Luego de esto, se ve un patrón de 4 parejas de procesos; en el primer proceso de todas las parejas se realiza una selección a través un proceso que, pasándole una variable en la sección de "Input/Output" como se ve en la Figura 19a, manda a llamar a un bloque de tipo DMN (*Decision Model and Notation*), el cual funciona como un *switch-case*, como se puede ver en la Figura 18; por otro lado, la segunda parte de todas las parejas cumple la función de ejecutar un *External Resource Script* como se ve en la Figura 19b, el cual se encarga de tomar los valores obtenidos del DMN y guardalos en variables de Camunda para que puedan ser utilizadas más adelante de ser necesario.

TIGO COMPONENTS DMN L... **Hit Policy:** First

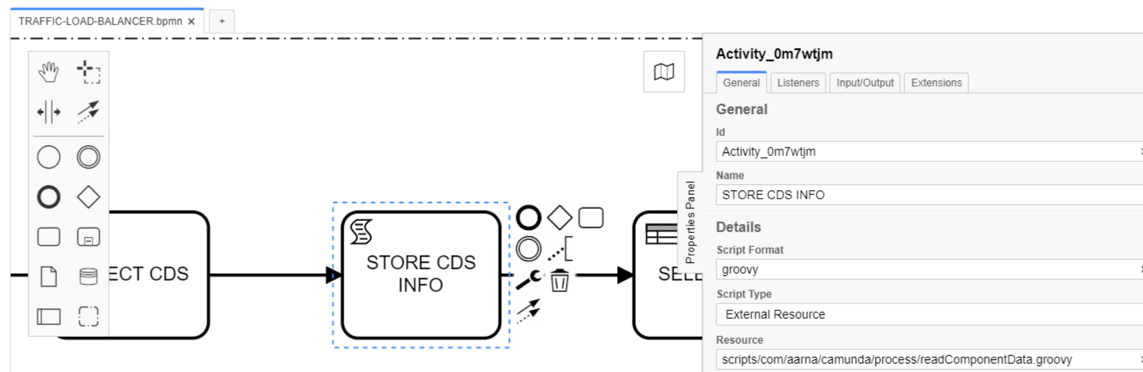
	When Component Name "aai","cds"	Then Host Name "host1","host2"	And Host Port string	And User Name string	And Password string	And Authorization string
1	"aai"	"aai"	[illegible]	[illegible]	[illegible]	
2	"cds"	"cds"	[illegible]	[illegible]	[illegible]	[illegible]
3	"kentik"	"kentik"		[illegible]	[illegible]	
4	"solarwinds"	[illegible]	[illegible]	[illegible]	[illegible]	
+	-					

Nótese que algunos valores están borrosos, lo cual se debe a que contienen credenciales confidenciales de un proveedor de servicios de Internet

Figura 18: Modelo DMN para la obtención de credenciales de los módulos de ONAP y plataformas externas.



(a) Selección de la plataforma externa de la cual se necesita las credenciales



(b) Ejecución de *External Resource Script* para almacenar las credenciales en variables de Camunda

Figura 19: Proceso de selección y almacenaje de credenciales de plataformas externas a Camunda

Luego de la obtención de credenciales de todas las plataformas externas que se verán involucradas (CDS, A&AI, Kentik y SolarWinds), se puede ver que hay un círculo con contorno doble y con una flecha rellena adentro, el cual es un evento *Throw* y que funciona en conjunto con un evento *Catch* con el fin de poder hacer que el flujo sea más fácil de leer y seguir visualmente; en este caso, los eventos *Throw* y *Catch* fueron utilizados para ordenar el flujo en sus tres diferentes etapas. Por esto, para iniciar la segunda etapa hay un evento *Catch*, el cual se ve idéntico al *Throw*, pero con la flecha sin rellenar. En esta segunda etapa se realiza la conexión a Kentik para obtener la información del tráfico a nivel de prefijo,

para lo cual se realizaron dos procesos: el envío de la petición a la API REST de Kentik y el *parsing* de la información retornada por la API. Ambos procesos, como se puede ver en la Figura 17, tienen una notación de un par de engranajes en la esquina superior izquierda, lo que significa que tienen su funcionalidad implementada a través de un *Service Task*, el cual puede ser un *Delegate Expression* o un *External Task* (en este caso, ambos procesos se implementaron a través de un *Delegate Expression*).

Finalmente, tras haber obtenido la información de Kentik y haberla procesado, nuevamente se agregan eventos de *Throw* y *Catch* para pasar a la útlima etapa: la ejecución del CBA. En esta etapa se ven nuevamente dos procesos de tipo *Service Task*. El primera proceso es el encargado de formar el *payload* para mandarlo al CDS; este proceso también fue implementado a través de un *Delegate Expression*. El segundo proceso, por otro lado, es el encargado de ejecutar el CBA, el cual fue implementado a través de un *External Task* ya predefinido que se encarga de ejecutar llamadas a APIs REST. El motivo por el cual esta consulta a una API REST se hizo a través de las clases de Java ya creadas en los *External Tasks* del proyecto bpmn-external-task-handler y la consulta a la API REST de Kentik no, es que los *External Tasks*, siguiendo el estándar que se utilizó para la ejecución del CBA antes mencionada, necesita de dos procesos: la creación del *payload* y la ejecución de la petición a la API, lo cual es más complejo y largo de realizar en código, lo cual únicamente vale la pena en algunos escenarios. En el caso de la ejecución del CBA sí aplica, puesto que dos de las ventajas más grandes de los *External Tasks* son que la petición es asíncrona, permitiendo que Camunda no se quede esperando a que el CBA termine de ejecutarse, y que los *External Tasks* son ejecutados por agentes externos (en este caso, el agente externo es el aplicativo de Java creado por el proyecto bpmn-external-task-handler que luego manda a llamar a los *pods* del CDS), lo cual permite un escalamiento específico para las tareas que lo requieran; en otras palabras, el uso de *External Tasks* para la ejecución de CBAs permiten independencia entre procesos del SO y el CDS, lo cual se ajusta perfectamente al objetivo de la plataforma ONAP de ser Cloud Native.

8.1.3. Adición de la funcionalidad del flujo

Como se mencionó en las secciones anteriores, el funcionamiento de este primer flujo automatizado consiste de tres etapas. Para la primera etapa se mencionó que se utilizó un *External Resource Script* para agregar el funcionamiento. Este *script* está escrito en Groovy y tiene una estructura similar a la que se puede ver en el Código 8.2, en donde puede observarse se obtiene la información que devuelve del DMN en el primer bloque de cada una de las 4 parejas de bloques y la guarda en variables de Camunda a través del comando `execution.setVariable('variable_name', value)` según la selección que se haya hecho en el DMN.

```
import groovy.json.JsonSlurper

Integer compSize = execution.getVariable("componentSize");
temprowData = execution.getVariable("componentRecords");
Map<String,String> mapData = new HashMap<String,String>();

if(compSize == 1)
    mapData =  temprowData.get(0);

mapData.each{entry -> println "$entry.key: $entry.value"}
String tempHost, tempport, tempUser, tempPass, tempAuth,tempOtherhost,
        tempType,tempBasicUrl,tempGrantType;;
```

```
for (entry in mapData) {
    println "Key: $entry.key = Value: $entry.value"
    if(entry.key == "host")
        tempHost = entry.value;
    if(entry.key == "port")
        tempport = entry.value;
    if(entry.key == "user")
        tempUser = entry.value;
    if(entry.key == "password")
        tempPass = entry.value;
    if(entry.key == "authorization")
        tempAuth = entry.value;
    if(entry.key == "otherhost")
        tempOtherhost = entry.value;
    if(entry.key == "type")
        tempType = entry.value;
    if(entry.key == "basicUrl")
        tempBasicUrl = entry.value;
    if(entry.key == "grantType")
        tempGrantType = entry.value;
}
if(tempType == "kentik") {
    execution.setVariable("kentikUrl", tempBasicUrl);
    execution.setVariable("kentikUsername", tempUser);
    execution.setVariable("kentikPassword", tempPass);
} else if(tempType == "solarwinds") {
    execution.setVariable("solarwindsHost", tempHost);
    execution.setVariable("solarwindsUsername", tempUser);
    execution.setVariable("solarwindsPassword", tempPass);
}
...
execution.setVariable("cbaStatusOut", false);
execution.setVariable("aaiStatusOut", false);
```

Código 8.2: Almacenaje de variables acorde a las selecciones del DMN

Luego de esto, en la segunda etapa se menciona que hay dos procesos, uno donde se envía la petición a la API REST de Kentik y otro donde se procesa la información obtenida. En el proceso de envíar la petición también se construye el *payload* que va hacia Kentik (este *payload* puede observarse en el Código 15.1, pero se explicará más a detalle en secciones futuras). La forma en que se realiza esto es a través de un *Delegate Expression* que ejecuta el Código 8.3. En este código, se puede observar que el sistema toma las variables que se guardaron en la primera etapa y las utiliza para colocar las credenciales y el URL de la API REST de Kentik que se va a utilizar. Después, se ve que se utiliza un archivo JSON como el *payload* de la petición a la API. Finalmente, se envía la petición a través de un POST a la API REST de Kentik y se guarda la respuesta en una variable de Camunda para poder ser utilizada más adelante.

```
...
private void getTraffic(DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "getTraffic: ");
    try {
        // General objects
        KentikHttpClient kentikClient = new KentikHttpClient();
        // Get global variables
        String resVariable = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_OUTPUT_KEY)
            .toString();
        // Get the variables to post the request
        String kentikUrl = delegateExecution
            .getVariable(DeviceConstants.KENTIK_URL)
            .toString();
```

```
        String kentikPayloadKey = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_PAYLOAD_KEY)
            .toString();
        String kentikUsername = delegateExecution
            .getVariable(DeviceConstants.KENTIK_USERNAME)
            .toString();
        String kentikPassword = delegateExecution
            .getVariable(DeviceConstants.KENTIK_PASSWORD)
            .toString();
        // Craft the payload
        String payloadFileName = "json/" + kentikPayloadKey + ".json";
        String kentikPayload = readResourceFile(payloadFileName);
        String kentikResponse = kentikClient.sendHttpPost(
                kentikUrl,
                kentikPayload,
                kentikUsername,
                kentikPassword);
        // Show response
        delegateExecution.setVariable(resVariable, kentikResponse);
    } catch (Exception e) {
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR, "getTraffic Exception
            Occurred!!");
    }
}
...
```

Código 8.3: Construcción y ejecución de la petición a la API REST de Kentik

Luego, la parte del procesamiento de la respuesta de la API REST de Kentik también se implementa con un *Delegate Expression*, el cual ejecuta el Código 8.4. Como puede verse en el código, esta parte itera sobre la respuesta obtenida de Kentik y extrae y transfoma únicamente la información útil para el CBA; la estructura de esta respuesta proveniente de Kenik puede verse en el Código 8.5.

```
...
private void parseTraffic(DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "parseTraffic: ");
    try {
        // Get global variables
        String outVariable = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_OUTPUT_KEY)
            .toString();
        String resVariable = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_RESPONSE_KEY)
            .toString();
        String kentikResponse = delegateExecution
        .getVariable(resVariable)
        .toString();
        // Original info
        ObjectMapper objectMapper = new ObjectMapper();
        JsonNode rootNode = objectMapper.readTree(kentikResponse);
        ArrayNode originalResultsArray = (ArrayNode) rootNode.get("results");
        // Transformation
        ObjectNode transformedKentikResponse = objectMapper.createObjectNode();
        ArrayNode transformedDataArray = objectMapper.createArrayNode();
        for (JsonNode resultNode : originalResultsArray) {
            ArrayNode originalDataArray = (ArrayNode) resultNode.get("data");
            for (JsonNode originalData : originalDataArray) {
                ObjectNode transformedDataObject = objectMapper.createObjectNode();
                ObjectNode srcRoutePrefixLenObject = objectMapper.createObjectNode();

                String avgBitsPerSec = originalData
                    .get("avg_bits_per_sec")
                    .asText();
                String iDstAsName = originalData
```

```java
                    .get("i_dst_as_name")
                    .asText();
                String inputPort = originalData
                    .get("input_port")
                    .asText();
                String iDDeviceName = originalData
                    .get("i_device_name")
                    .asText();
                String prefixLen = originalData
                    .get("dst_route_prefix_len")
                    .asText();

                srcRoutePrefixLenObject.put("bps", avgBitsPerSec);
                srcRoutePrefixLenObject.put("asn", iDstAsName);
                srcRoutePrefixLenObject.put("port", inputPort);
                srcRoutePrefixLenObject.put("pnf", iDDeviceName);
                srcRoutePrefixLenObject.put("prefix", prefixLen);

                transformedDataObject.set("prefix_data", srcRoutePrefixLenObject);
                transformedDataArray.add(transformedDataObject);
            }
        }
        transformedKentikResponse.set("results", objectMapper.createArrayNode()
                .add(objectMapper.createObjectNode()
                .set("data", transformedDataArray)));
        // Convert to string
        String finalResponse = objectMapper
            .writeValueAsString(transformedKentikResponse)
            .replace("\"", "\\\"");
        delegateExecution.setVariable(outVariable, finalResponse);

    } catch (JsonProcessingException jsonExp) {
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR, "parseTraffic
            Exception Occurred !!");
    } catch (Exception exception) {
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR, "parseTraffic
            Exception Occurred !!");
    }
}
...
```

Código 8.4: Construcción y ejecución de la petición a la API REST de Kentik

```json
{
    "results": [
        {
            "bucket": "Left +Y Axis",
            "data": [
                {
                    "key": "",
                    "avg_bits_per_sec": ,
                    "p95th_bits_per_sec": ,
                    "max_bits_per_sec": ,
                    "input_port": "",
                    "dst_route_prefix_len": "",
                    "i_dst_as_name": "",
                    "i_device_name": "",
                    "lookup": ""
                }
            ]
        }
    ]
}
```

Código 8.5: Respuesta de Kentik con la información del tráfico por prefijo

Finalmente, para la tercera y última etapa del flujo, se mencionó que se utilizan dos procesos: uno para construir el *payload* y otro para ejecutar el CBA. El proceso de construcción del *payload* se implementó a través de un *Delegate Expression* que ejecuta el Código 9.7. En este código, se puede observar que se obtiene la información de la alarma de SolarWinds para guardar en una variable la interfaz saturada, luego se obtiene el archivo .json en el que se encuentra el template del CBA y se utiliza la función constructJson del objeto payLoadBuilder, el cual es una instancia de la clase `JsonPathWrap.java`, para rellenar esa plantilla con la información encontrada a través de la ejecución del flujo. Finalmente, se guarda el *payload* en una variable de Camunda para que pueda ser utilizada por el proceso de ejecución del CBA.

```
private void prepareTraffic(DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "prepareTraffic: ");
    try {
        // Get the saturated interface
        String alarmInfo = delegateExecution
            .getVariable("alarm_json")
            .toString();
        ObjectMapper objectMapper = new ObjectMapper();
        JsonNode alarmInfoData = objectMapper.readTree(alarmInfo);
        String interfaceName = alarmInfoData.get("InterfaceName").asText();
        delegateExecution
            .setVariable("saturatedInterface", interfaceName);
        String pnfIP = alarmInfoData.get("pnf_ipv4_address").asText();
        delegateExecution
            .setVariable("pnfIPv4Address", pnfIP);
        // Give initial values and check the variables
        String payloadKey = "";
        if (delegateExecution.getVariable(ProcessDeviceConstants.VAR_PAYLOAD_KEY) !=
            null) {
            payloadKey = delegateExecution
                .getVariable(ProcessDeviceConstants.VAR_PAYLOAD_KEY)
                .toString();
        }
        String stateTempPayload = "{}";
        if (delegateExecution.getVariable(ProcessDeviceConstants.VAR_PAYLOAD) !=
            null) {
            stateTempPayload = delegateExecution
                .getVariable(ProcessDeviceConstants.VAR_PAYLOAD)
                .toString();
        }
        delegateExecution
            .setVariable("processInstanceId",
                delegateExecution.getVariable("instanceid"));
        // Form the JSON
        StringBuilder tempBuiler = new StringBuilder();
        tempBuiler.append("json/").append(payloadKey).append(".json");
        String payloadStateFileName = tempBuiler.toString();
        JsonPathWrap payLoadBuilder = new JsonPathWrap();
        String finalStatePayload = payLoadBuilder.constructJson(
                payloadStateFileName,
                stateTempPayload,
                delegateExecution,
                null);
        String resStateVariable = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_OUTPUT_KEY)
            .toString();
        // Check the payload
        delegateExecution
            .setVariable(resStateVariable, finalStatePayload);
        String stateDryRun = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_DRY_RUN)
            .toString();
    } catch (Exception e) {
```

```
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR, "prepareTraffic
            Exception Occurred!!");
    }
}
```

Código 8.6: Construcción del *payload* para la ejecución del CBA de descubrimiento de capacidades

Luego, el segundo proceso de la tercera etapa se implementó con un *Service Task* de tipo *External Task*, el cual, como se ha mencionado antes, se implementa a través de un componente externo a Camunda, que en este caso es el proyecto de Java bpmn-external-task-handler. La forma en que Camunda maneja este tipo de tareas es que el aplicativo externo está *polleando* un *endpoint* de la API REST de Camunda en donde se exponen las *External Tasks* con un *topic* específico; este aplicativo está suscrito a un *topic*, por lo que cuando hace el *poll* y encuentra una tarea pendiente con el *topic* al que está suscrito, la toma y la ejecuta a través del *handler* que tiene definido para esas tareas. En este caso, como se ve en la Figura 20, el proceso coloca el *External Task* bajo el *topic* "externalRestApiTask".

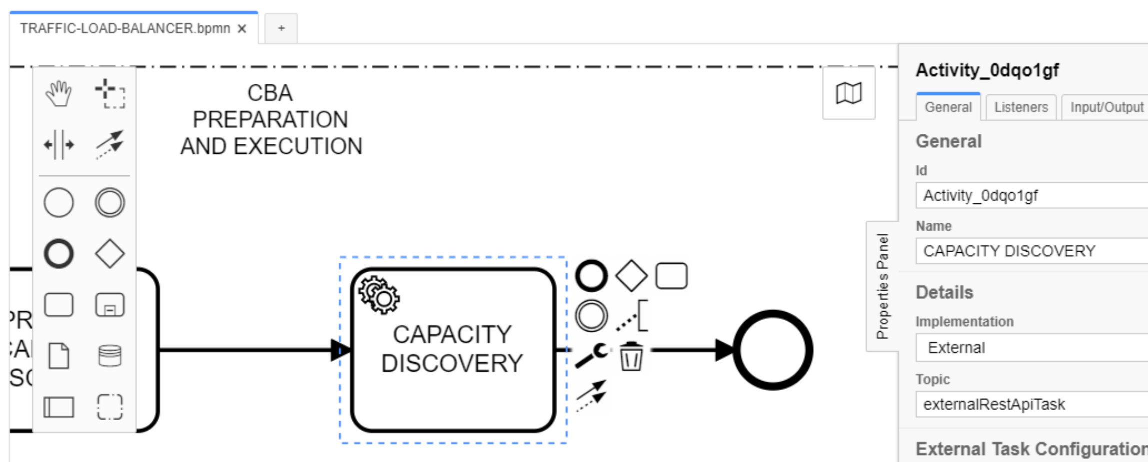


Figura 20: Suscripción de un proceso implementado con *External Task* a un *topic*.

8.2. Conexión a Kentik

8.2.1. Creación de vistas en Kentik

Anteriormente se mencionó que en la segunda etapa del proceso se obtiene la información de Kentik. Para lograr esto, primero debe de crearse una vista que contenga la información que se desea obtener a través de la API REST. Además de la vista que se hizo para obtener la información del tráfico en interfaces de IP Transit, se diseñaron otras vistas para poder observar el funcionamiento de la red. Para realizar el diseño de las vistas, Kentik ofrece una opción para configurar tableros, como se ve en la Figura 21. Esta opción que ofrece Kentik cuenta con las siguientes opciones para obtener la información y graficarla:

- Saved View: Se utiliza para dar información extra a la vista, como un nombre, la descripción y la categoría en la que se encuentra.

- Visualization: Habilita la selección del tipo de gráfica que se hará con los datos. Existen muchas opciones, pero para este trabajo de graduación se trabajó únicamente con las gráficas de tipo “Line Chart”.
- Data Sources: Permite seleccionar de qué sistemas se obtiene la información. Kentik obtiene la información de los equipos a través de protocolos de monitoreo como SNMP directamente de los equipos, por lo que esta opción permite filtrar específicamente de qué equipos se desea obtener la información.
- Dimensions: Esta opción permite seleccionar qué datos desean mostrarse en la tabla que se visualiza debajo de la gráfica.
- Metrics: Indica la dimensional del eje Y de la gráfica. En el caso de este trabajo de graduación se trabajó con bits/s.
- Time: Permite seleccionar el rango de tiempo que se desea visualizar. En el caso de este trabajo de graduación se trabajó con los últimos 5 minutos. Por otro lado, esta opción también permite seleccionar la zona horaria que se despliega en el eje X de la gráfica.
- Filtering: Se utiliza para colocar filtros en la información que se obtiene de los equipos que se seleccionen en la sección “Data Sources” antes mencionada.

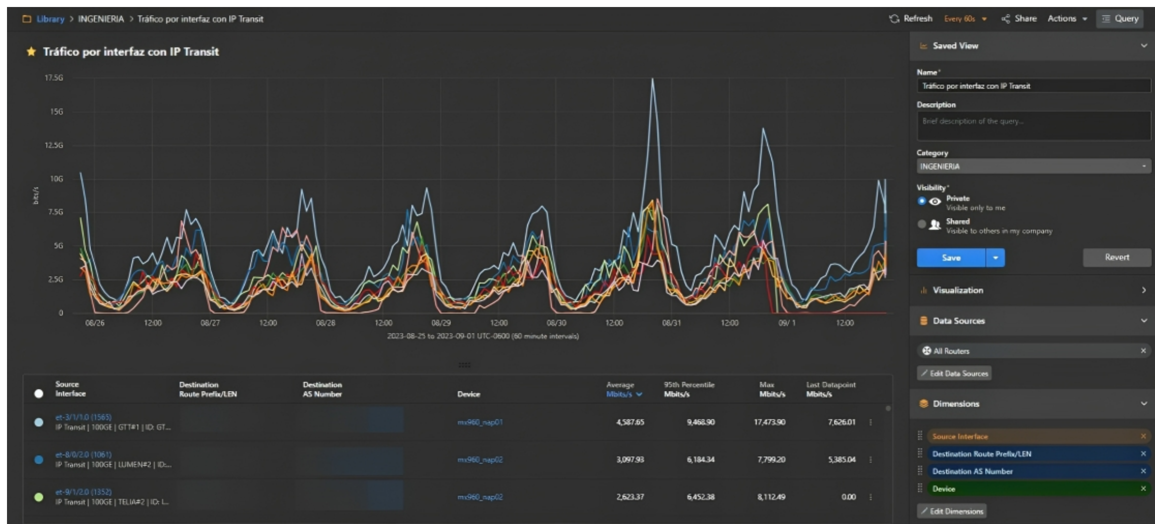


Figura 21: Interfaz de configuración vistas de Kentik.

La primera vista que se diseñó puede verse en la Figura 22, en donde se obtiene el tráfico de las interfaces de IP Transit, la cual fue utilizada para determinar un buen umbral a partir del cual ya se debe realizar un balanceo y para comparar con la información que luego se mostrará en SolarWinds y comprobar que la información que está siendo *polleada* por Kentik sea correcta. Por otro lado, en la Figura 23 ya se ve el tráfico que consume cada prefijo que pertenece a una interfaz de IP Transit, indicando la interfaz por la que está pasando, el prefijo que es, a qué ASN pertenece, el equipo al que pertenece la interfaz y el consumo que está teniendo; la información que proporciona esta vista es la que se utiliza en este

primer flujo de automatización para la obtención de las opciones para conmutar el tráfico y balancear la carga.

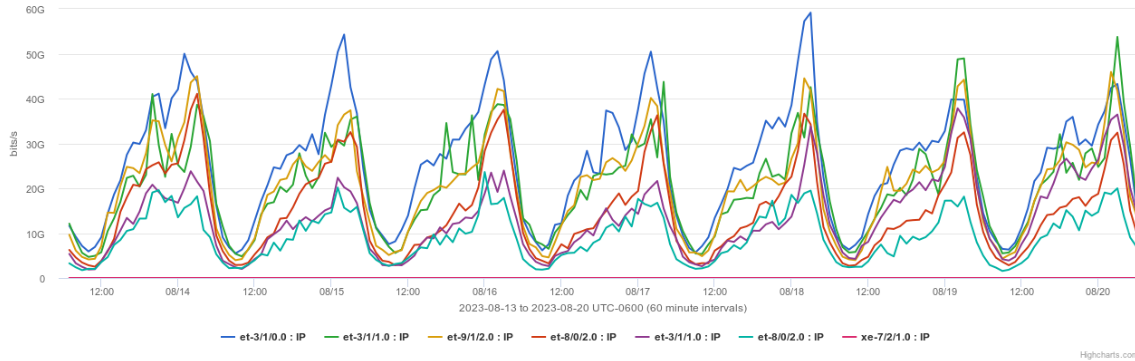


Figura 22: Consumo en Gbps por interfaces de IP Transit.

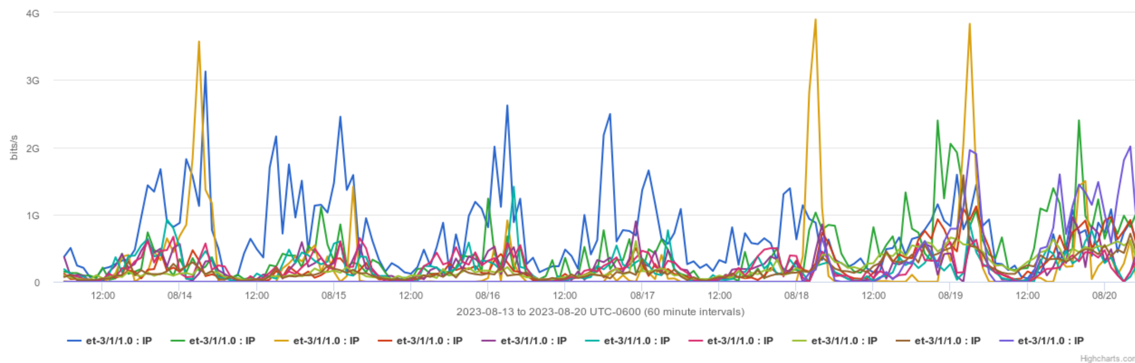


Figura 23: Consumo en Gbps por interfaces de IP Transit según el prefijo.

8.2.2. Obtención de la llamada a la API de Kentik correspondiente a la vista creada

Una vez creada la vista, Kentik ofrece una opción para obtener la llamada a la API REST correspondiente a la vista creada, como se ve en la Figura 24. Esta opción permite obtener el *payload* para obtener la información que se ve desplegada en la vista creada o el comando que utiliza el *software* `curl` para ejecutar el POST a la API REST. En este caso, se utilizó únicamente el *payload*, puesto que no se va a correr el comando desde la línea de comandos de alguno de los nodos del servidor de ONAP, sino más bien desde Camunda a través de los archivos descritos en la Sección 8.1. Puesto que en la Figura 24 no se puede ver la URL que se utiliza para realizar la consulta a Kentik, esta información puede ser consultada en la Sección 15 en el Cuadro 7.

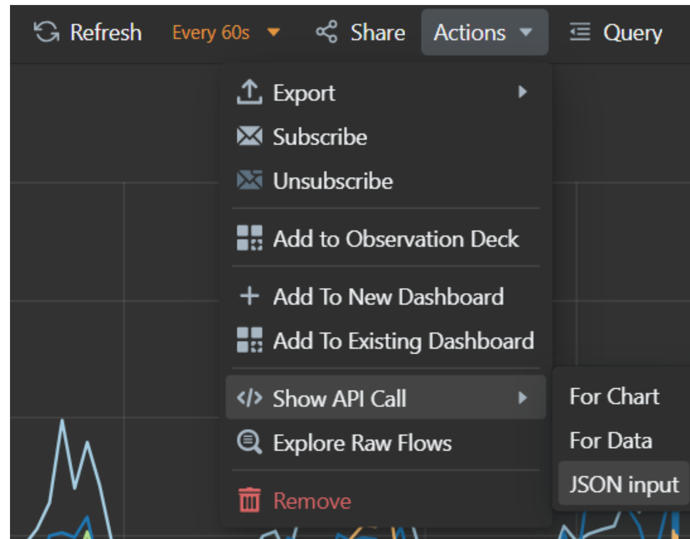


Figura 24: Opción en Kentik para obtener el *payload* para la llamada a su API REST.

8.3. CBA para el descubrimiento de capacidades disponibles

Como se mencionó en la Sección 8.1, el último paso del flujo de descubrimeinto de capacidades es la ejecución del CBA. Para esto, se tuvo que realizar el diseño del CBA, tanto su funcionamiento, como la forma en que se le envía el *payload* para su ejecución, lo cual se muestra en las siguientes subsecciones.

8.3.1. Configuración de archivos necesarios para el funcionamiento del CBA

Como fue explicado en la Sección 7.5, el CBA para poder funcionar necesita de un archivo en donde estén las definiciones y un archivo para la metadata. En el caso del archivo de metadata, es exactamente como el archivo mostrado en el Código 7.5 y no necesita cambiar en todo el desarrollo del proyecto. Por otro lado, el archivo de definiciones sí puede cambiar en el desarrollo de un caso de uso, puesto que cada flujo contemplado requiere de agregar información específica para que este funcione. En este trabajo de graduación, el CBA creado consta de dos flujos. En los códigos 8.7 y 8.8 se muestran los fragmentos de las definiciones correspondiente al flujo de descubrimiento de capacidades disponibles; estos fragmentos de código se agregan a las llaves "*workflows*" y "*node_templates*" del JSON mostrado en el Código 7.6 correspondientemente. Además de esto, como se ve en el Código 8.9, se agregó un macro llamado *parameters* que jerárquicamente se encuentra debajo de la llave "*dsl_-definitions*"; en este macro se definen todos los parámetros que se utilizarán en los flujos configurados dentro del CBA.

```
{
    "capacity-discovery": {
        "steps": {
            "execute-script": {
5               "description": "",
```

```
                "target": "execute-capacity-discovery"
            }
        },
        "inputs": {},
        "outputs": {}
    }
}
```

Código 8.7: Definición del *Workflow* de descubrimiento de capacidades disponibles

```
{
    "execute-capacity-discovery": {
        "type": "component-remote-script-executor",
        "interfaces": {
            "ComponentRemoteScriptExecutor": {
                "operations": {
                    "process": {
                        "inputs": {
                            "selector": "*py-executor",
                            "blueprint-name": "bgp-load-balancer",
                            "blueprint-version": "",
                            "blueprint-action": "CapacityDiscoveryExecutor",
                            "timeout": {
                                "get_input": "py-exec-grpc-timeout"
                            },
                            "request-data": "*parameters"
                        }
                    }
                }
            }
        }
    }
}
```

Código 8.8: Definición del nodo para la ejecución del código en el *python executor* para el descubrimiento de capacidades disponibles

```
{
    "parameters": {
        "input-params": {
            "get_input": "input-params"
        },
        "cds-py-exec-pod-ip": {
            "get_input": "cds-py-exec-pod-ip"
        },
        "py-exec-grpc-timeout": {
            "get_input": "py-exec-grpc-timeout"
        },
        "workflow-name": {
            "get_input": "workflow-name"
        },
        "skip-input-params-keys-validation": {
            "get_input": "skip-input-params-keys-validation"
        }
    }
}
```

Código 8.9: Definición del macro para la aceptación de parámetros

8.3.2. Diseño del *payload* para la REST API del CDS

Como se mencionó en la Sección 7.4, uno de los componentes clave del CDS es el "blueprint-processor-http", el cual se encarga de exponer APIs REST (siguiendo la ideología *Cloud Native*) para poder realizar el proceso de despliegue descrito en la Sección 7.6 y ejecutar los flujos definidos en la Subsección 8.3.1. Puesto que la información que se pasa por HTTP a través de la API REST expuesta se utiliza para activar un flujo en los *python executors* a través de una API gRPC, los campos que se ven en el Código 8.10 son también campos definidos en el *Protocol Buffer* para este aplicativo. La información debajo de la llave "commonHeader" es para metadata que sirve para identificar el proceso que se está ejecutando; la información debajo de la llave "actionIdentifiers" es para identificar el flujo del CBA que se desea ejecutar, que en este caso es el CBA "bgp-load-balancer" y el flujo "capacity-discovery" que se creó en la Subsección 8.3.1; y la información debajo de la llave "payload" es la información que se pasa al flujo que se desea ejecutar. Nótese que la información que se pasa en el *payload* son justamente los parámetros definidos en el macro "parameters" en el Código 8.9.

En el caso de este *payload*, puede verse que se envía la información de credeciales de SolarWinds y la A&AI, la cual fue obtenida en el flujo de Camunda descrito en la Sección 8.1. Por otro lado, también se incluye la información de la interfaz saturada y el *pnf* de origen que proviene de la alarma de SolarWinds y la información del tráfico que se obtuvo de Kentik. Finalmente, puede verse que se manda en el *payload* parámetros específicos para la ejecución del código dentro del *pod* de Python, para el uso de *logs*, para la ejecución asíncrona de ciertas partes del código, para determinar el umbral que se está utilizando y para colocar una cantidad máxima de movimientos válidos. La información de los *logs* indica qué tantos *logs* serán mostrados en la terminal de los *python executors*, puesto que si se están haciendo pruebas puede convenir utilizar la opción de *DEBUG*, que se encarga de mostrar la mayor cantidad posible de información útil para determinar si la aplicación está haciendo lo esperado o no y que facilita la búsqueda de errores, sin embargo, si el aplicativo ya se encuentra en producción, lo mejor sería utilizar la opción de *INFO*, la cual se encarga de mostrar únicamente la información realmente necesaria. Por otro lado, para la ejecución asíncrona se utilizan las librerías "asyncio" y "aiohttp", las cuales, para tener un funcionamiento más controlado y óptimo, pueden utilizarse en conjunto con los conceptos de semáforos, cantidad de conexiones permitidas y cantidad de reintentos al ejecutar una sección de código antes de darlo por fallido, que es lo que se realiza en este trabajo de graduación.

```
1  {
       "commonHeader": {
           "originatorId": "",
           "requestId": "",
           "subRequestId": ""
6      },
       "actionIdentifiers": {
           "blueprintName": "bgp-load-balancer",
           "blueprintVersion": "",
           "actionName": "capacity-discovery",
11         "mode": "sync"
       },
       "payload": {
           "capacity-discovery-request": {
               "implementation": {
16                 "timeout":
```

```
            },
            "cds-py-exec-pod-ip": "",
            "py-exec-grpc-timeout": "",
            "workflow-name": "capacity-discovery",
      "skip-input-params-keys-validation": [],
      "input-params": {
        "solarwinds-access-info": {
          "solarwinds-host-name": "",
          "solarwinds-username": "",
          "solarwinds-password": ""
        },
        "aai-resources-access-info": {
          "aai-service-host-name": "",
          "aai-base-url": "",
          "aai-service-port": "",
          "aai-username": "",
          "aai-password": "",
          "aai-api-timeout":
        },
        "async-access-info": {
          "semaphore": "",
          "connections": "",
          "retry": ""
        },
        "traffic-info": "",
        "saturated-interface": "",
        "from-pnf": "",
        "log-level": "",
        "threshold": "",
        "max-moves": ""

      }
        }
    }
}
```

Código 8.10: *Payload* para la constula al servicio cds-blueprints-processor-http para la ejecución del CBA de descubrimiento de capacidades

8.3.3. Algoritmo para determinar las capacidades disponibles para balancear el tráfico

Una vez se realiza una petición a la API del "blueprint-processor-http", como se mencionó en la Subsección 8.3.2, este microservicio se encarga de redireccionar la petición al microservicio gRPC (que en este caso de uso es el *python executor*). Puesto que dentro del *payload* se incluye el nombre del CBA y el flujo que desea ejecutarse, el servidor gRPC es capaz de buscar los CBAs que tiene disponibles (gracias a que se hizo el proceso de despliegue mencionado en la Sección 7.6) y luego ejecutar el *script* correspondiente (el cual fue definido como se ve en la Subsección 8.3.1). Como se puede ver en el Código 8.8, existe un objeto debajo de la llave "operations" con nombre "process", el cual indica qué función debe de ejecutarse, mientras que el valor de la llave "blueprint-action" indica que la clase que deberá de utilizarse es "CapacityDiscoveryExecutor", la cual está definida dentro del archivo "capacity_discovery_executor.py". En el Código 8.11 puede verse un fragmento del código que implementa la función "process" de la clase "CapacityDiscoveryExecutor" antes mencionada.

```
# Imports
...
```

```
class CapacityDiscoveryExecutor(AbstractScriptFunction):

    def __init__(self):
        # Init method
        ...

    def process(self, execution_request):
        try:
            common_header =
                json.loads(json_format.MessageToJson(execution_request.commonHeader))
            input_json =
                json.loads(json_format.MessageToJson(execution_request.payload))[
                CbaCommonKeywords.CAPACITY_CBA_PAYLOAD]
            skip_input_vars = input_json.get(CbaCommonKeywords.SKIP_INPUT_KEY)
            input_params = input_json.get(CbaCommonKeywords.INPUT_PARAMS)
            workflow_name = input_json.get(CbaCommonKeywords.CBA_WORKFLOW_NAME)
            final_resp_data = {
                "py-executor-originator-id": common_header.get("originatorId"),
                "py-executor-request-id": common_header.get("requestId"),
                "py-executor-subrequest-id": common_header.get("subRequestId"),
                "capacity-discovery-status" : {
                    "capacity-discovery-response" : [],
                    "response-code":"200",
                    "response-data":"",
                    "custom-response-attributes":[],
                    "status":"success"
                }
            }

            input_params_obj = GenericInputParameterValidator(
                workflow_name,
                input_params,
                skip_input_vars
            )
            validated_info = input_params_obj.process_validation()

            if validated_info.get(’status’):
                is_connection = False
                solarwinds_input_params = input_params.get("solarwinds-access-info")
                aai_input_params = input_params.get("aai-resources-access-info")
                async_input_params = input_params.get("async-access-info")
                saturated_interface = input_params.get("saturated-interface")
                from_pnf = input_params.get("from-pnf")
                traffic_input_params = input_params.get("traffic-info")
                if traffic_input_params:
                    solarwinds_host =
                        solarwinds_input_params.get("solarwinds-host-name")
                    solarwinds_username =
                        solarwinds_input_params.get("solarwinds-username")
                    solarwinds_password =
                        solarwinds_input_params.get("solarwinds-password")
                    self.solarwinds_handler = SolarWindsHandler(
                        solarwinds_host,
                        solarwinds_username,
                        solarwinds_password
                    )

                    all_ip_transit_interfaces = (self
                        .solarwinds_handler
                        .get_interfaces_available_bandwidth())
                    load_balancing_options =
                        self.solarwinds_handler.get_load_balancing_options(
                        saturated_interface,
                        from_pnf,
                        traffic_input_params,
                        all_ip_transit_interfaces
                    )
```

```python
                    execution_response = executor_utils.success_response(
                        execution_request,
                        final_resp_data,
                        CbaCommonKeywords.BP_SUCCESS_STATUS_CODE
                    )
                else:
                    ...

        except Exception as err:
            ...

        yield execution_response
```

Código 8.11: Clase para la ejecución del CBA de descubrimiento de capacidades disponibles

Como puede observarse, el código puede separarse en dos partes: la primera parte consta de la obtención y *parseo* de la información proveniente de la llamada gRPC (la cual es la misma que la descrita en el *payload* mostrado en el Código 8.10); mientras que la segunda parte consiste en el uso de la información obtenida para obtener los posibles movimientos. Esta segunda parte, como se observa, no es implementada directamente en este *script*, sino que se crea una instancia de la clase “SolarWindsHandler”, la cual está definida en el archivo “solarwinds_handler.py”. Esta clase se encarga de utilizar la API REST que ofrece SolarWinds para realizar peticiones a la base de datos del sistema para poder obtener información sobre las interfaces de IP Transit. En esta clase se crearon los métodos `get_interfaces_available_bandwidth` y `get_load_balancing_options`, los cuales sirven para obtener información de las interfaces IP Transit y los posibles movimientos que se pueden realizar para balancear la carga respectivamente. En los códigos 8.13 y 8.14 se muestran fragmentos de cada uno de los métodos, mientras que en el Código 8.12 se ve un fragmento de la clase “SolarWindsHandler”.

En el Código 8.12, se puede apreciar que se utilizó un decorador para medir el tiempo de ejecución de cada uno de los métodos de la clase, lo cual es útil para determinar si el tiempo de ejecución de cada uno de los métodos es aceptable o si se debe de optimizar el código; el código de este decorador puede verse a más detalle en el Código 15.16. De igual forma, se ve que se utiliza la función `urllib3.disable_warnings()`, la cual es necesaria utilizarla para que la librería para realizar peticiones HTTP “urllib3” no muestre un *warning* en la terminal de los *python executors* cuando se realiza una petición a la API REST de SolarWinds, puesto que esta API no cuenta con un certificado SSL válido, sino uno auto firmado.

```python
# Imports
...

urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

...

class SolarWindsHandler:

    def __init__(self, npm_server, username, password, logger=None, port=17778):
        ...

    def logger_for_decorator(self):
        ...

    @timer(logger_for_decorator)
    def get_interfaces_available_bandwidth(self):
        ...
```

```python
    @timer(logger_for_decorator)
    def get_load_balancing_options(self, saturated_interface, pnf, traffic_info,
        ip_transit_interfaces):
        ...

    @timer(logger_for_decorator)
    def get_possible_moves(self, prefixes_from_sat, from_bps, from_threshold,
        ip_transit_interfaces, possible_moves):
        ...

    @timer(logger_for_decorator)
    def check_move_combination(self, move_combinations, valid_moves, new_from_bps):
        ...
```

Código 8.12: Clase para la obtención y uso de la información de SolarWinds

Por otro lado, en el Código 8.13 se puede observar que se realiza una petición a la base de datos de SolarWinds a través de su API REST al utilizar la función `self.swis.query(query)`, donde la petición (*query*) está definido en el archivo "swql_constants.py", el cual puede verse más a detalle en el Código 15.8. Con la información que devuelve esta petición, se guarda en una lista todas las interfaces de IP Transit encontradas en la base de datos de SolarWinds con los siguientes atributos: nombre, *Tier 1* al que pertenece, bits por segundo que están entrando, porcentaje de utilización, velocidad máxima en bits por segundo y router al que pertenece.

```python
def get_interfaces_available_bandwidth(self):
    interfaces_info = []
    query = SWQLConstants.INTERFACE_INPUT_AVAILABLE_BANDWIDTH
    results = self.swis.query(query)
    for interface in results['results']:
        try:
            interface_name = interface['Caption'].split('|')[0].split()[0]
            tier1 = interface['Caption'].split('|')[2].strip()
            bps = float(interface['Inbps'])
            utilization = float(interface['InPercentUtil'])
            speed = float(interface['Speed'])
            pnf_ipv4 = interface['IPAddress']
            interface_info = {}
            interface_info['pnf_ipv4'] = pnf_ipv4
            interface_info['interface_name'] = interface_name
            interface_info['tier1'] = tier1
            interface_info['bps'] = bps
            interface_info['utilization'] = utilization
            interface_info['threshold'] = self.saturation_factor*speed
            interfaces_info.append(interface_info)
        except Exception as error:
            pass
    return interfaces_info
```

Código 8.13: Método para la obtención de información de las interfaces IP Transit

Luego, como se ve en el Código 8.14, se obtiene la información del tráfico que está pasando por la interfaz saturada, para poder tener la información de esta interfaz separada del resto, con el fin de tener una distinción entre la interfaz de origen y de destino para hacer el movimiento. Al igual que para el método `get_interfaces_available_bandwidth`, de el resultado de esta petición se obtuvieron los parámetros más importantes de la interfaz saturada. Luego de esto, se realizaron los siguientes filtros para poder descartar movimientos que no son posibles:

- Si la cantidad de bits por segundo que están saliendo de la interfaz saturada es más pequeño que el umbral configurado para esa interfaz, significa que hubo un falso positivo en la alarma de SolarWinds, por lo que no se debe de realizar ningún movimiento.
- Si la cantidad de bits por segundo calculada que saldrían de la interfaz saturada si se hiciera el movimiento sigue siendo más grande que el umbral configurado para esa interfaz, significa que la interfaz seguiría saturada aunque se hiciera el movimiento y que este no logra balancear la carga, por lo que no debe de realizarse.
- Si la cantidad de bits por segundo que estarían saliendo de la interfaz de destino luego de realizar el movimiento es más grande que el umbral configurado para esa interfaz, significa que la interfaz de destino se saturaría, por lo que ese movimiento no debe de realizarse.

Nótese que no todos estos filtros pueden verse en el Código 8.14, lo cual se debe a que están en los métodos `get_possible_moves` y `check_move_combination`, los cuales pueden verse más a detalle en los códigos 15.9 y 15.10 respectivamente. Una parte crucial del algoritmo para encontrar los movimientos posibles para lograr un balanceo de carga y que se explica también en los códigos mencionados anteriormente es que debe de tomarse en cuenta que el tráfico de cada prefijo puede moverse en grupos, por lo que el concepto de combinatorias y de los problemas de creación de grupos con la ecuación "n escoge k" debe de ser tomado en cuenta; además de esto, no solo pueden hacerse grupos de prefijos, sino que cada uno de estos grupos puede moverse a cualquier grupo de interfaces donde quepa, por lo que la aplicación de combinatoria se extiende también a la selección de las interfaces de destino del tráfico. Por esto, el algortímo lo que hace es que, de todos los prefijos que pertenecen a la interfaz saturada, obtiene todas las combinaciones de grupos posibles (empezando por grupos de 1 y creciendo el tamaño si se agotan las opciones para ese tamaño específico), y utiliza todas las capacidades de destino posibles para armar todos los grupos de movimientos posibles. Luego, para cada uno de estos grupos de movimientos, se verifica que se cumplan los filtros mencionados anteriormente y que se ven en los códigos 15.9 y 15.10. Puesto que puede que existan demasiados posibles movimientos, el algoritmo termina una vez se encuentre una cantidad máxima de movimientos válidos definidos en el *payload* de Camunda que luego es pasado al *payload* del CDS, como se muestra en la Subsección 8.3.2.

```
def get_load_balancing_options(self, saturated_interface, pnf, traffic_info,
    ip_transit_interfaces):
    prefixes_from_sat = []

    query = SWQLConstants.SPECIFIC_INTERFACE_INPUT_AVAILABLE_BANDWIDTH
5   from_interface_info = self.swis.query(query.format(saturated_interface,
        pnf))['results'][0]

    from_tier1 = from_interface_info['Caption'].split('|')[2].strip()
    from_bps = float(from_interface_info['Inbps'])
    from_threshold = self.saturation_factor*float(from_interface_info['Speed'])
10  possible_moves = {
        "from-pnf": pnf,
        "from-interface": saturated_interface,
        "from-tier1": from_tier1,
        "from-bps": from_bps,
15      "from-threshold": from_threshold,
        "move-groups":[]
    }

    if from_bps < from_threshold:
```

```
        return possible_moves

    ip_transit_interfaces = [
        interface for interface in ip_transit_interfaces
        if not (interface['pnf_ipv4'] == pnf and interface['interface_name'] ==
            saturated_interface)
    ]

    # First Step:
    traffic = json.loads(traffic_info)
    for prefix in traffic['results'][0]['data']:
        prefix_info = {}
        usage = float(prefix['prefix_data']['bps'])
        asn = prefix['prefix_data']['asn']
        interface_name = prefix['prefix_data']['port'].split(':')[0].strip()
        pnf_ipv4 = 'x.x.x.x' if prefix['prefix_data']['pnf'] == 'mx960_nap01' else
            'y.y.y.y'
        prefix_len = prefix['prefix_data']['prefix'].split()[0].strip()

        if interface_name != saturated_interface or pnf_ipv4 != pnf:
            continue
        prefix_info = {
            "prefix": prefix_len,
            "prefix-bps": usage,
            "asn": asn
        }
        prefixes_from_sat.append(prefix_info)

    prefixes_from_sat = sorted(prefixes_from_sat, key=lambda x: x['prefix-bps'],
        reverse=True)

    # Second Step:
    possible_moves = self.get_possible_moves(prefixes_from_sat, from_bps,
        from_threshold, ip_transit_interfaces, possible_moves)
    return possible_moves
```

Código 8.14: Método para la obtención de los posibles movimientos que se pueden realizar para balancear la carga

8.4. Conexión a SolarWinds

Una parte clave del funcionamiento de este caso de uso es que la activación de los flujos sea automática y en tiempo real según la necesidad de la red. Esto puede conseguirse a través de el gestor SolarWinds, el cual utiliza el protocolo SNMP para estar monitoreando de forma continua a la red, permitiendo detectar cuando la utilización de una capacidad sobrepasa el umbral configurado. Para que el sistema efectúe una acción al momento de detectar este evento, es necesario configurar la alarma con sus acciones de activación y reinicio. Estas configuraciones pueden verse a continuación.

8.4.1. Creación de la alarma

En primer lugar, una vez se esté en el portal principal, se debe de ir al apartado de alertas, mostrado en verde en la barra superior que se ve en la Figura 25. Luego, es necesario ubicarse en la pantalla de gestión de alertas. Para acceder a esta debe de irse a la opción de "*Manage Alerts*" que también puede verse seleccionada en verde en la Figura 25.

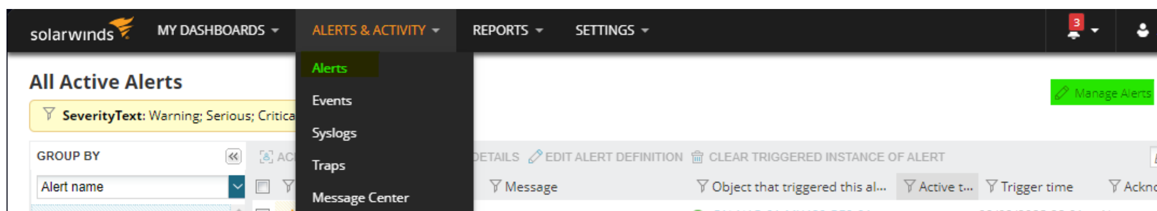


Figura 25: Tablero para el manejo de las alertas de SolarWinds

Luego, como se observa en la Figura 26, debe de seleccionarse la opción "*Add New Alert*" resaltada en verde para iniciar el proceso de creación de la alarma. Este proceso requerirá configurar los siguientes pasos:

1. Propiedades de la alerta
2. Condición de activación de la alerta
3. Condición de resolución de la alerta
4. Horario específico para que la alerta pueda activarse
5. Definición de la acción al momento de activación de la alerta

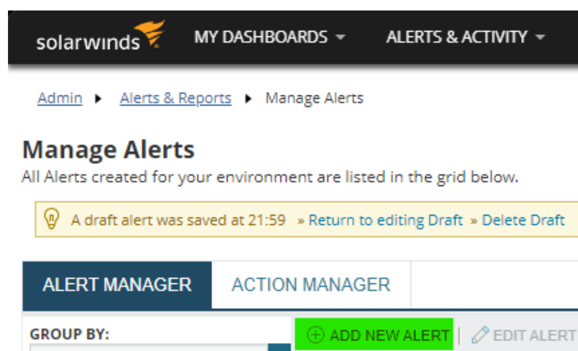


Figura 26: Creación de una nueva alerta

En el primera paso, como se ve en la Figura 27, debe de configurarse el nombre de la alerta, una descripción y cuánto tiempo debe de estar la condición de la alerta activa para que la acción se ejecute.

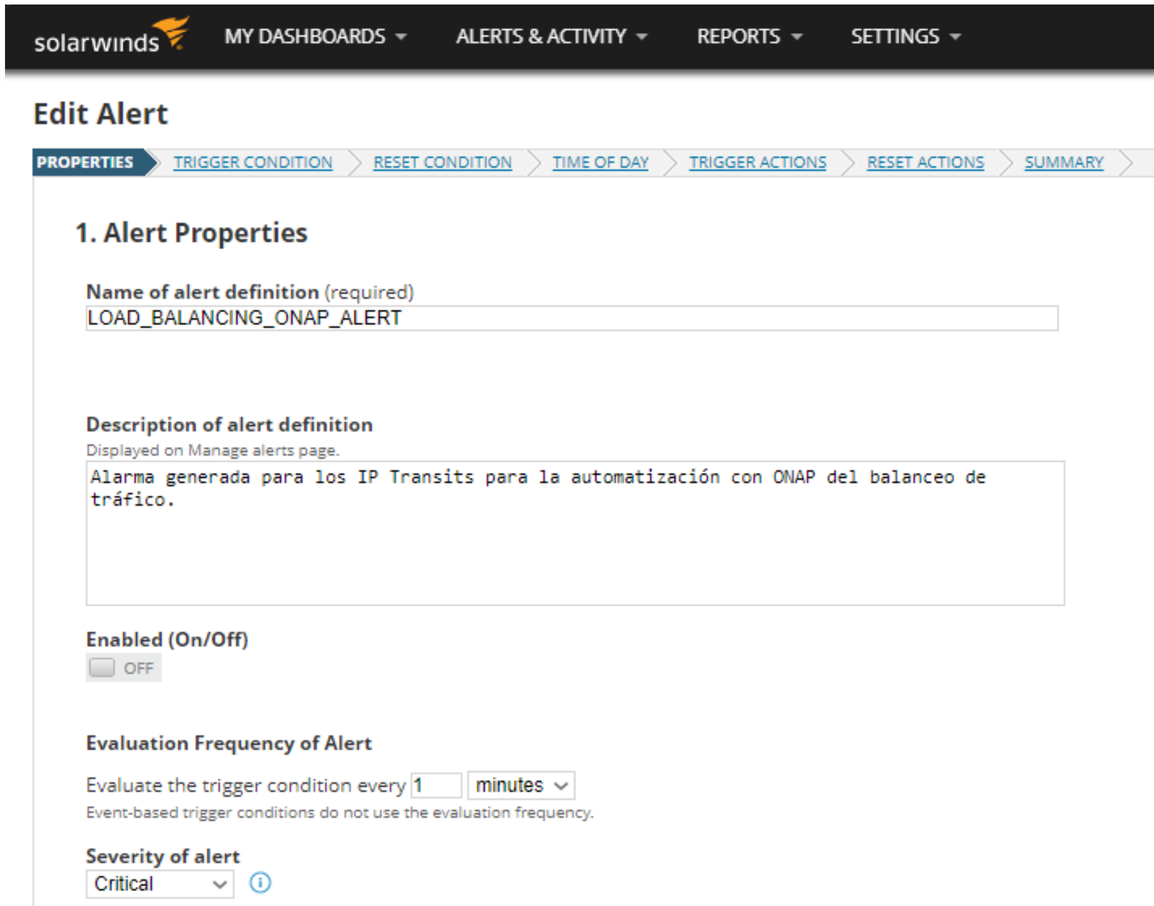


Figura 27: Definición de las propiedades de la alerta

Luego, como segundo paso, es necesario determinar la condición para que la alarma se active. En este caso, como puede verse en la Figura 28, se seleccionó que la alarma se active cuando haya una interfaz de IP Transit que sobrepase un porcentaje de utilización del 30 %. Es importante mencionar que se seleccionó un valor bajo para poder realizar pruebas en tiempo real.

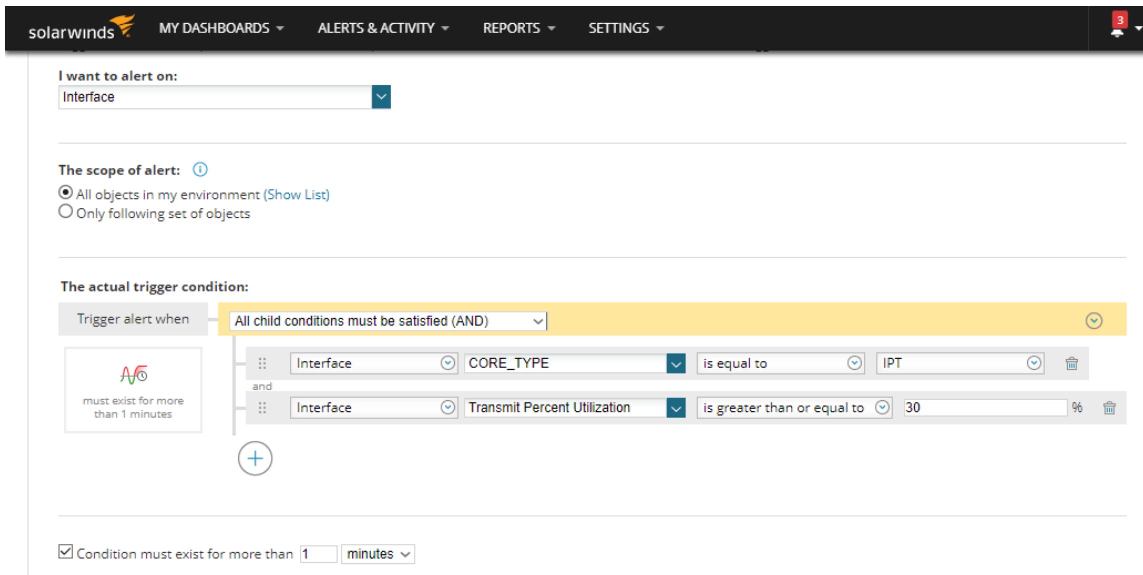


Figura 28: Definición de la condición que provoca que se levante la alerta

Después, tanto para el tercer paso como para el cuarto, se dejó la configuración que existe por defecto, que es que no haya ninguna acción al momento de que la alarma se resuelva (como se ve en la Figura 29) y que la alarma sea funcional durante todo el tiempo (como se ve en la Figura 30).

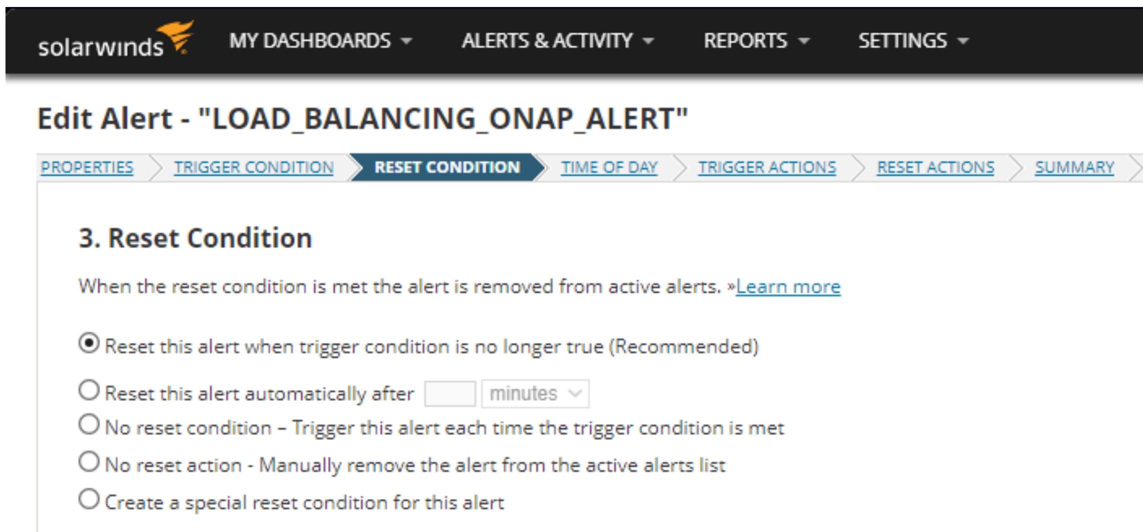


Figura 29: Definición de la condición para que la alerta se resuelva

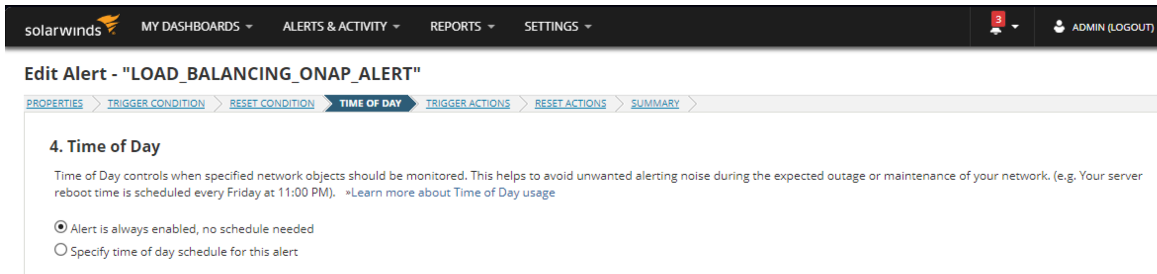


Figura 30: Configuración para las horas del día en las cuales la alerta tiene efecto

Finalmente, se debe de definir la acción que SolarWinds ejecutará cuando se levante la alarma. Como se ve en la Figura 31, primero se define qué aparecerá en el tablero de gestión de alarmas cuando la misma sea activada y se le asigna una acción. Como puede verse en la Figura 32, esta acción es la ejecución de un *script*, el cual se encarga de hacer una llamada a la API REST de Camunda para iniciar el proceso descrito en la Sección 8.1. SolarWinds cuenta con una forma nativa de hacer POST y GET a APIs a través de HTTP y HTTPS, sin embargo, no se pudo utilizar esta funcionalidad, ya que el servicio de Camunda utiliza el protocolo HTTPS con certificado auto firmado, por lo que SolarWinds detecta un problema de seguridad y no ejecuta la petición. Por esto, se encontró una solución alterna en la cual se utilizó un *script* de Python utilizando la misma función que se utilizó en el Código 8.12 para evitar los mensajes de advertencia de la librería “urllib3” de Python, logrando inicializar el flujo de Camunda de forma automática.

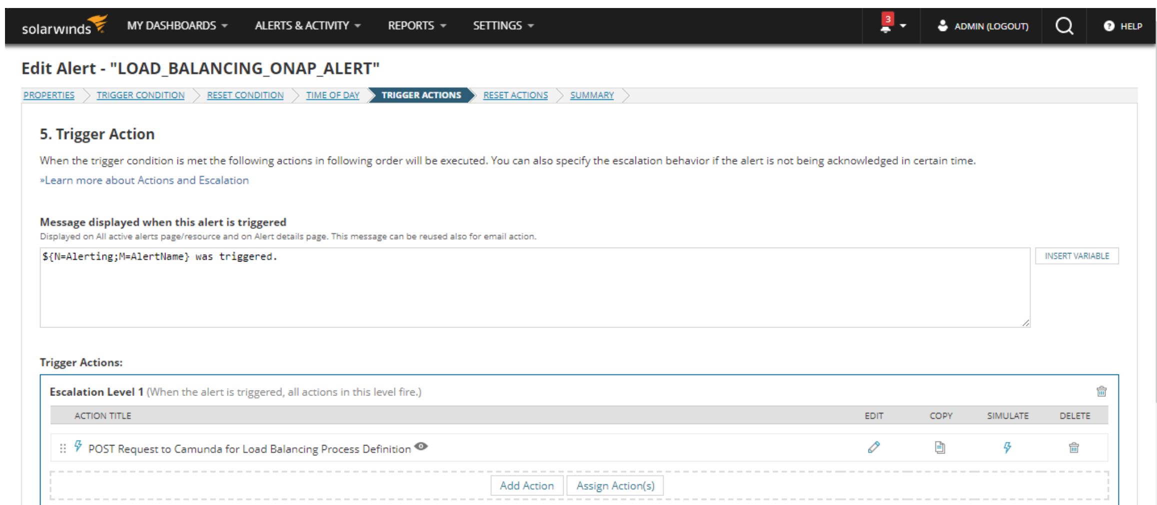


Figura 31: Definición de la acción a realizar cuando se levanta la alerta

Configure Action: Execute An External Program

Name of action

POST Request to Camunda for Load Balancing Process Definition

Execute an external program settings

Network path to external program

C:\Users\Administrator\Desktop\dist\camunda_script.exe "${N=SwisEntity;M=Alias}" "${N

INSERT VARIABLE

For example, use c:\test.bat, where c:\ is the disk on your main Orion poller.
You can also use the network address, such as \\machine-name\test.bat or \\127.0.0.1\test.bat

Windows authentication

Default user (SYSTEM)

Define user

Time of Day No additional schedule for this action needed

Execution settings Do not execute if acknowledged already.

SAVE CHANGES CANCEL

Figura 32: Detalles de la acción a realizar cuando se levanta la alerta

CAPÍTULO 9

Configuración de los equipos de red para realizar el balanceo de carga

Una vez obtenida la información del tráfico de Internet a través de las herramientas Kentik y SolarWinds, se procede a realizar la configuración de los equipos de red para realizar el balanceo de carga. Para esto, se debe de crear un flujo de Camunda y un CBA del CDS para la configuración de los equipos de red. A continuación, se muestra tanto el proceso de creación de estos flujos y CBAs, como la conexión a los equipos de red para la configuración de los mismos.

9.1. Flujo de Camunda para la configuración

Similar al caso del primer flujo descrito en el Capítulo 8, para realizar el flujo de Camunda para la configuración de los equipos de red, es necesario hacer tanto el diseño del *payload* del *endpoint* correspondiente al flujo de la API REST, como del flujo en sí bajo el estándar BPMN.

9.1.1. Diseño del *payload* para la REST API de Camunda

En el caso de este segundo flujo de automatización, como se ve en el Código 9.1, se utiliza la información obtenida del primer flujo para ejecutarse, ya que necesita la información del *Tier 1* de origen, que por la forma en que está diseñada la red, también es otra forma de identificar a la interfaz saturada original; y la información de los movimientos seleccionados. Nótese que para la información de los movimientos seleccionados, se utiliza una lista, puesto que es posible que el movimiento seleccionado conste de tener que mover varios prefijos a diferentes *Tiers 1*, por lo cual, dentro de la información que se necesita de cada uno está el *Tier 1* hacia el cual se va a mover, el prefijo y el ASN. El ASN es importante saberlo, puesto

que indica a qué país pertenece el prefijo; esta información permite saber qué equipos deben de configurarse para lograr el balanceo, ya que los prefijos son específicos de ciertos países. Al igual que en el primer flujo, la parte importante para Camunda es el *payload*, mientras que el resto de datos son realmente útiles para el CBA del CDS.

```
{
    "variables": {
        "payload": {
            "type": "String",
            "value": "{\"fromTier1\":\"\", \"selectedMove\": [{\"toTier1\":\"\",\"
    prefix\":\"\", \"asn\":\"\"}]}"
        },
        "cbaVersion": {
            "type": "String",
            "value": ""
        },
        "semaphore": {
            "type": "String",
            "value": ""
        },
        "connections": {
            "type": "String",
            "value": ""
        },
        "retry": {
            "type": "String",
            "value": ""
        },
        "logLevel": {
            "type": "String",
            "value": ""
        }
    }
}
```

Código 9.1: *Payload* para la ejecución del flujo de configuración

9.1.2. Diseño de los flujos en Camunda Modeler

En este caso, el flujo diseñado para Camunda es más complejo que el caso del primer flujo, puesto que una configuración para el balanceo de carga puede consistir en múltiples configuraciones en varios equipos, por lo cual, se tomó la decisión de separar al flujo en dos.

Para la primera parte del flujo, como se puede ver en la Figura 33, existen tres secciones: obtención de credenciales para herramientas externas, obtención de los equipos relacionados en el movimiento a realizar para el balanceo y la configuración en sí de los equipos. La primera sección es idéntica a la primera sección del primer flujo de automatización descrito en el Capítulo 8. La segunda sección consta de tres bloques que siguen el patrón *Construct - Put - Parse*; este patrón es utilizado en otros casos de uso y sirve para tener una forma ordenada y modular para poder construir la petición a la A&AI, hacer el requerimiento de tipo PUT o GET, y hacer un *parseo* de la información que devuelve la base de datos. Nótese que si el método que se utiliza es PUT, significa que se está haciendo una petición de tipo DSL. Las peticiones DSL son peticiones especiales que la A&AI permite hacer con el propósito de hacer búsquedas en la base de datos de grafos cuando se tienen ciertas características de un elemento, pero no se sabe cuál es o cuando pueden existir varios elementos que cumplan con las características buscadas. Un ejemplo de una petición para la cual no tendría sentido

utilizar un DSL es obtener la información de un PNF en específico, puesto que se sabe qué elemento es y se tiene el *endpoint* específico para consultar la información, por lo que aplicaría más realizar un GET (aunque es importante mencionar que sí es posible y sí funcionaría); por otro lado, un ejemplo donde sí aplica utilizar un DSL es cuando se quiere obtener la información de todos los PNFs que son de un país en específico. Este último ejemplo es justamente lo que se realiza en este flujo, ya que se cuenta con el ASN, el cual también indica el país en el que debe de realizarse la configuración (como se mencionó en la Subsección 9.1.1). Finalmente, la tercera sección consiste en iterar a través de los PNFs encontrados en la sección dos del flujo para ejecutar las configuraciones.

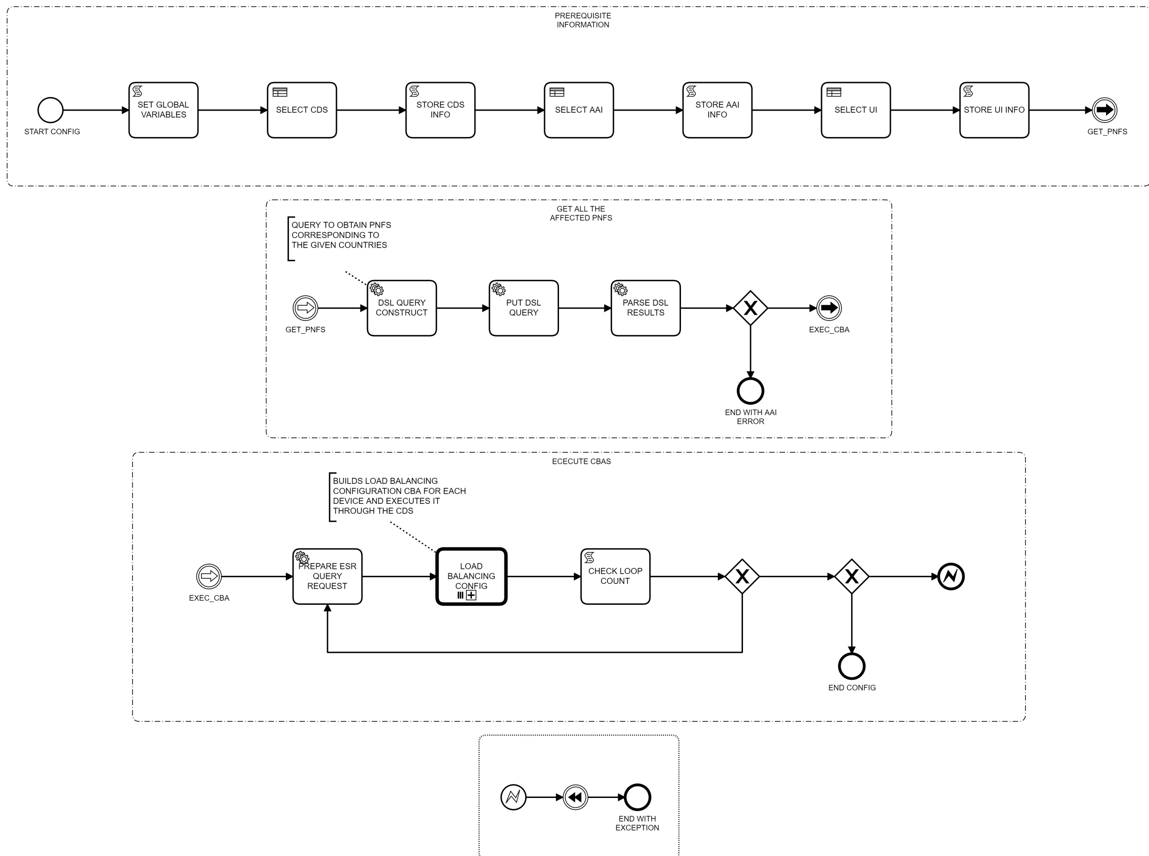


Figura 33: Modelo BPMN del flujo de Camunda para la configuración para el balanceo de carga.

Como se ve en la Figura 33, el bloque llamado “LOAD BALANCING CONFIG” es un rectángulo con contorno más grueso que el resto, lo cual significa que es una llamada a un flujo externo. Además de esto, el bloque también cuenta con un símbolo de tres líneas verticales paralelas, las cuales indican que la ejecución en paralelo de múltiples instancias de ese proceso externo es permitida; de esta forma, si se encuentran varios equipos que deben de configurarse para hacer el movimiento de un prefijo, todas las configuraciones puede realizarse de forma simultánea, permitiendo así agilizar el proceso. Por otro lado, también se puede observar en esta sección que hay un símbolo de un rombo con una equis en medio, el cual representa a una condicional o decisión. En este caso, el primer símbolo de decisión es utilizado para determinar si hay más equipos que tengan que configurarse, mientras que el segundo símbolo de decisión es utilizado para indicar si algo salió mal o no, lo cual puede

verse por el símbolo de un círculo con un rayo relleno adentro, el cual indica un evento de *try* - *catch* para detectar y manejar errores.

El bloque que es instanciado en la tercera sección del flujo puede verse a detalle en la Figura 34. Este flujo secundario, al igual que el principal mostrado en la Sección 9.1 y que el primer flujo mostrado en la Sección 8.1, cuenta con un proceso llamado "SET GLOBAL VARIABLES", dentro del cual se utilizan las variables pasadas en la instanciación del bloque en el flujo principal para mostrarlas en los *logs*, lo cual es útil para realizar pruebas. Seguido de esto, se tiene nuevamente un grupo de tres bloques que siguen el patrón *Construct* - *Put* - *Parse*. En este caso, el requerimiento a la base de datos que se realiza es para obtener las credenciales del equipo que se va a configurar; puesto que se sabe el equipo del cual se quiere obtener información y es único, se utiliza el método GET con el *endpoint* específico del PNF. Luego de este patrón, puede verse un bloque llamado "PREPARE STATE DATA REQUEST", el cual es un proceso utilizado para formar el *payload* del CDS para poder ejecutar el CBA a partir de una plantilla de un archivo JSON; este proceso es similar al realizado en la Subsección 8.1.2 para el primer flujo.

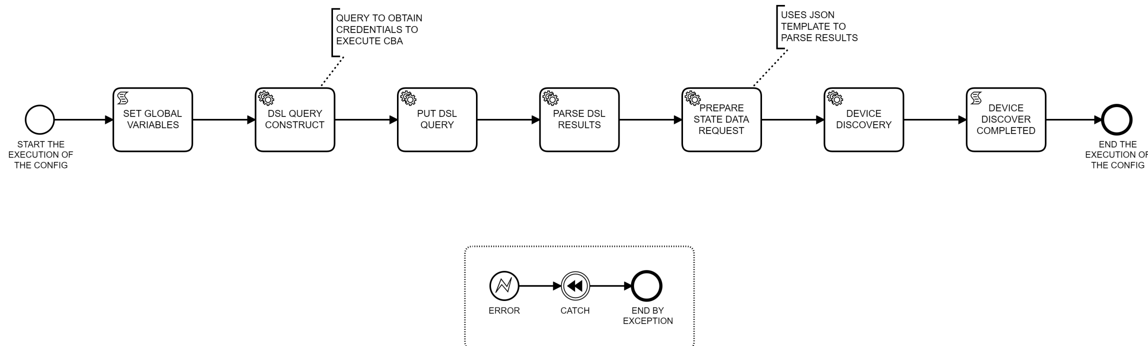


Figura 34: Modelo BPMN del flujo de Camunda la ejecución de la configuración por equipo.

9.1.3. Adición de la funcionalidad del flujo

Como se mencionó en las secciones anteriores, el funcionamiento del segundo flujo automatizado consiste de tres etapas. La primera etapa es idéntica a la primera etapa del primer flujo de Camunda descrito en la Subsección 8.1.3. Luego de esto, en la segunda etapa se menciona el uso del patrón *Construct* - *Put* - *Parse*, el cual es un patrón que consta de tres procesos; el primer proceso sirve para construir la petición a la A&AI y guardar ese URL en una variable lista para ejecutar la petición, este proceso es un *Service Task* implementado específicamente con un *Delegate Expression* y que utiliza la clase `QueryConstructUtils`, la cual es una clase diseñada específicamente para definir funciones para la construcción de las peticiones a la A&AI; el segundo proceso es utilizado para realizar la petición en sí a la base de datos, lo cual es implementado también con un *Service Task*, pero en este caso se hace uso de un *External Task*, en el cual se obtiene el URL guardado en la fase de *Construct* y se ejecuta, guardando el resultado en una variable de Camunda; finalmente, el tercer paso es utilizado para interpretar la información retornada por la petición a la base de datos, lo cual es implementado también con un *Delegate Expression*, dentro del cual, se obtiene la información devuelta por el paso anterior y se pasa a través de una función definida para

la clase `QueryParseUtils`, la cual es una clase creada específicamente para definir funciones para el *parseo* de información.

Como puede verse en el Código 9.2, se utiliza la fase de *Construct* para obtener las IPs de todos los equipos involucrados para realizar los movimientos a través de la petición que se hará a la base de datos (que en este caso es un DSL), la cual es obtenida con el uso de la función `QueryConstructUtils.constructInstallationQuery`, que puede verse más a detalle en el Código 15.4. Luego, como puede verse en el Código 9.3, se utiliza la fase de *Put* para realizar la petición HTTP a la API REST de la base de datos. Finalmente, como puede verse en el Código 9.4, la fase *Parse* es utilizada con la función `QueryParseUtils.parsePnfByCountry`, la cual es una función que puede verse más a detalle en el Código 15.6.

```
private void preparePnfByCountryQuery(DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "preparePnfByCountryQuery: ");
    String countries = getCountries(delegateExecution);
    try {
5       String tempUrl =
            delegateExecution.getVariable(DeviceConstants.CUSTOM_URL_AAI).toString();

        QueryConstructUtils.constructInstallationQuery(
            countries,
            tempUrl,
10          DslConstants.PNF_BY_COUNTRY_QUERY,
            delegateExecution);

        String prepareAaiPayload = delegateExecution
            .getVariable(DeviceConstants.AAI_REQ_PAYLOAD)
15          .toString();

        String resVariable = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_OUTPUT_KEY)
            .toString();
20      delegateExecution
            .setVariable(resVariable, prepareAaiPayload);
    } catch (Exception e) {
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR,
            "preparePnfByCountryQuery Exception Occurred !!");
25  }
}
```

Código 9.2: Fase de construcción de la petición en el patrón *Construct* - *Put* - *Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos

```
public void aaiQueryRestApiTask(ExternalTask externalTask,
    ExternalTaskService externalTaskService) {
    logger.log(Level.INFO, "aaiQueryRestApiTask : Inside aaiQueryRestApiTask");
4   int tempRetries = aaiRetries;
    Integer extRetries = externalTask.getRetries();
    try {
        String requestPayload = externalTask
            .getVariable(ExternalTaskConstant.AAI_REQ_PAYLOAD);
9       String aaiUrl = externalTask
            .getVariable(ExternalTaskConstant.AAI_REQ_URL);
        String username = externalTask
            .getVariable(ExternalTaskConstant.AAI_REQ_USERNAME);
        String password = externalTask
14          .getVariable(ExternalTaskConstant.AAI_REQ_PASSWORD);
        String methodType = externalTask
            .getVariable(ExternalTaskConstant.AAI_REQ_METHOD);
        String responseKey = externalTask
            .getVariable(ExternalTaskConstant.RESPONSE_KEY);
19
        if (methodType.equalsIgnoreCase("get")) {
```

```
            responseStr = aaiHttpClient
                .sendHttpsGet(aaiUrl, username, password);
        } else if (methodType.equalsIgnoreCase("put")) {
            responseStr = aaiHttpClient
                .sendHttpsPut(aaiUrl, username, password, requestPayload);
        }

        ObjectValue responseObject = Variables.objectValue(responseStr).create();
        externalTaskService.complete(externalTask,
            Collections.singletonMap("externalTaskAAIOutput", responseObject));
    } catch (Exception exp) {
        tempRetries = tempRetries - 1;
        if (extRetries == null) {
        externalTaskService.handleFailure(externalTask.getId(),
            "externalWorkerID", "AAI Query Failed", tempRetries, 10000);
        } else if (extRetries - 1 == 0) {
        externalTaskService.handleBpmnError(externalTask, "CBAError");
        } else {
        externalTaskService.handleFailure(externalTask, "errorMessage",
            "errorDetails", externalTask.getRetries() - 1,
            calculateRetryDelay(externalTask.getRetries()));
        }
    }
}
```

Código 9.3: Fase de envío de la petición en el patrón *Construct - Put - Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos

```
private void processPnfByCountryResponse(DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "processPnfByCountryResponse :");
    try {
        String ipResponse = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_EXTERNAL_TASK_AAI_OUTPUT)
            .toString();

        QueryParseUtils.parsePnfByCountry(ipResponse, delegateExecution);

    } catch (Exception e) {
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR,
            "processPnfByCountryResponse Exception Occurred !!");
    }
}
```

Código 9.4: Fase de transformación, filtrado y utilización de la información de la petición realizada en el patrón *Construct - Put - Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos

Finalmente, para la tercera y última etapa del flujo, se mencionó que se utiliza otro flujo externo para instanciarlo múltiples veces para que corra de forma paralela. Dentro de este flujo se mencionó nuevamente el uso del patrón *Construct - Put - Parse* como segundo paso luego de obtener las variables globales. Como puede verse en el Código 9.5, se utiliza la fase de *Construct* para obtener las credenciales del equipo que se va a configurar, las cuales son obtenidas con el uso de la función `QueryConstructUtils.constructEsrQuery`, la cual es una función que puede verse más a detalle en el Código 15.5. Luego, la fase de *Put* es idéntica a la utilizada en el Código 9.3. Finalmente, como puede verse en el Código 9.4, la fase *Parse* es utilizada con la función `QueryParseUtils.parseExternalEsrResponse`, la cual es una función que puede verse más a detalle en el Código 15.7.

```
private void prepareEsrInfoQuery(DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "prepareEsrInfoQuery :");
    try {
```

```
        String tempEsrIpv4 = null;
        String esrTempUrl = delegateExecution
            .getVariable(DeviceConstants.CUSTOM_URL_AAI)
            .toString();
        if (delegateExecution.getVariable(ProcessDeviceConstants.IPV4_SYSTEM) !=
            null) {
        tempEsrIpv4 = delegateExecution
            .getVariable(ProcessDeviceConstants.IPV4_SYSTEM)
            .toString();
        }
        QueryConstructUtils.constructEsrQuery(tempEsrIpv4, esrTempUrl,
            delegateExecution);
        String esrAaiPayload = delegateExecution
            .getVariable(DeviceConstants.AAI_REQ_PAYLOAD)
            .toString();

        String resEsrVariable = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_OUTPUT_KEY)
            .toString();
        delegateExecution
            .setVariable(resEsrVariable, esrAaiPayload);
    } catch (Exception e) {
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR,
            "prepareEsrInfoQuery Exception Occurred !!");
    }
}
```

Código 9.5: Fase de construcción de la petición en el patrón *Construct - Put - Parse* para la obtención de las credenciales del equipo a configurar

```
private void processEsrInfoResponse(DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "processEsrInfoResponse :");
    try {
        String esrIpResponse = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_EXTERNAL_TASK_AAI_OUTPUT)
            .toString();
        QueryParseUtils.parseExternalEsrResponse(esrIpResponse, delegateExecution);
    } catch (Exception e) {
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR,
            "processEsrInfoResponse Exception Occurred !!");
    }
}
```

Código 9.6: Fase de transformación, filtrado y utilización de la información de la petición realizada en el patrón *Construct - Put - Parse* para la obtención de las credenciales del equipo a configurar

Luego, como siguiente paso del flujo de automatización auxiliar, se crea con la información encontrada un *payload* para poder ejecutar el CBA de configuración. La creación de este *payload* puede verse en el Código 9.7.

```
private void prepareConfigureRequest(DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "prepareConfigureRequest :");
    try {
        String payloadKey = "";
        if (delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_PAYLOAD_KEY) != null) {
            payloadKey = delegateExecution
                .getVariable(ProcessDeviceConstants.VAR_PAYLOAD_KEY)
                .toString();
        }
        String stateTempPayload = "{}";
        if (delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_PAYLOAD) != null) {
            stateTempPayload = delegateExecution
                .getVariable(ProcessDeviceConstants.VAR_PAYLOAD)
```

```
                .toString();
        }
        delegateExecution
            .setVariable("processInstanceId",
                delegateExecution.getVariable("instanceid"));
        String baseAaiUrl = delegateExecution
            .getVariable(DeviceConstants.AAI_BASIC_REQ_URL)
            .toString();
        delegateExecution
            .setVariable(DeviceConstants.AAI_REQ_URL, baseAaiUrl);

        StringBuilder tempBuiler = new StringBuilder();
        tempBuiler.append("json/").append(payloadKey).append(".json");
        String payloadStateFileName = tempBuiler.toString();
        JsonPathWrap payLoadBuilder = new JsonPathWrap();
        String finalStatePayload = payLoadBuilder.constructJson(
            payloadStateFileName,
            stateTempPayload,
            delegateExecution,
            null);
        String resStateVariable = delegateExecution
            .getVariable(ProcessDeviceConstants.VAR_OUTPUT_KEY)
            .toString();
        delegateExecution
            .setVariable(resStateVariable, finalStatePayload);
    } catch (Exception e) {
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR,
            "prepareConfigureRequest Exception Occurred !!");
    }
}
```

Código 9.7: Construcción del *payload* para la ejecución del CBA

Como último paso del flujo de automatización auxiliar, se implementó un *Service Task* de tipo *External Task* para ejecutar el CBA con el *payload* ya formado. Al igual que en la Subsección 8.1.3, este *External Task* está suscrito al *topic* "externalRestApiTask".

9.2. CBA para la configuración

9.2.1. Configuración de archivos necesarios para el funcionamiento del CBA

En este trabajo de graduación se ha expuesto un caso de uso que consta de dos grandes flujos: el descubrimiento de capacidades para realizar movimientos de tráfico y la ejecución de la configuración para realizar los movimientos, lo cual puede lograrse a través de varios acercamientos. Un primer acercamiento podría ser crear dos CBAs totalmente separados para cada flujo, y otro acercamiento (y el utilizado en este trabajo de graduación) es utilizar el mismo CBA, pero definiendo flujos diferentes, así como se hizo en la Subsección 9.2.1. En los códigos 9.8 y 9.9 se muestran los fragmentos de las definiciones correspondiente al flujo de configuración; estos fragmentos de código se agregan a las llaves "*workflows*" y "*node_-templates*" del JSON mostrado en el Código 7.6 correspondientemente. Al igual que en el primer flujo definido para el descubrimeinto de capacidades disponibles, se utiliza en este flujo el macro para ingresar parámetros definido en el Código 8.9, mas no es necesario crear uno nuevo gracias al acercamiento de desarrollar ambos flujos en el mismo CBA. Nótese que el acercamiento escogido se utilizó porque sigue siendo ordenado y modular tener estos flujos

en el mismo CBA porque son para el funcionamiento del mismo caso de uso, sin embargo, no se recomienda hacer esto para flujos que sirven para cosas totalmente diferentes y sin relación entre sí, ya que se perdería el orden.

```
{
    "bgp-load-balancer": {
        "steps": {
            "execute-script": {
                "description": "",
                "target": "execute-bgp-load-balancer"
            }
        },
        "inputs": {},
        "outputs": {}
    }
}
```

Código 9.8: Definición del *Workflow* de configuración

```
{
    "execute-bgp-load-balancer": {
        "type": "component-remote-script-executor",
        "interfaces": {
            "ComponentRemoteScriptExecutor": {
                "operations": {
                    "process": {
                        "inputs": {
                            "selector": "*py-executor",
                            "blueprint-name": "bgp-load-balancer",
                            "blueprint-version": "",
                            "blueprint-action": "BGPLoadBalancerExecutor",
                            "timeout": {
                                "get_input": "py-exec-grpc-timeout"
                            },
                            "request-data": "*parameters"
                        }
                    }
                }
            }
        }
    }
}
```

Código 9.9: Definición del nodo para la ejecución del código en el *python executor* para la configuración

9.2.2. Diseño del *payload* para la REST API del CDS

En el caso del *payload* para el flujo de configuración que se observa en el Código 9.10, puede observarse que se envía la información de credeciales del equipo que se va a configurar, la cual fue obtenida a través del patrón *Construct - Put - Parse* descrito en la Subección 9.1.3, y las credenciales de la A&AI, las cuales fueron obtenidas a través del bloque DMN. Por otro lado, también se incluye la información del *Tier 1* del cual se está liberando carga, que proviene de la alarma de SolarWinds. Además de esto, puede verse que se incluye la información del movimiento seleccionado proveniente del primer flujo de automatización descrito en la Sección 9.1. Finalmente, puede verse que se manda en el *payload* parámetros específicos para la ejecución del código dentro del *pod* de Python, tanto para el uso de *logs*, como para la ejecución asíncrona de ciertas partes del código.

```
{
    "commonHeader": {
        "originatorId": "",
        "requestId": "",
        "subRequestId": ""
    },
    "actionIdentifiers": {
        "blueprintName": "bgp-load-balancer",
        "blueprintVersion": "",
        "actionName": "bgp-load-balancer",
        "mode": "sync"
    },
    "payload": {
        "bgp-load-balancer-request": {
            "implementation": {
                "timeout":
            },
            "cds-py-exec-pod-ip": "cds-py-executor",
            "py-exec-grpc-timeout": "",
            "workflow-name": "bgp-load-balancer",
            "skip-input-params-keys-validation": [],
            "input-params": {
                "pnf-access-information": {
                    "cloud-owner": "",
                    "cloud-region-name": "",
                    "vendor": "",
                    "version": "",
                    "user-name": "",
                    "password": ""
                },
                "aai-resources-access-info": {
                    "aai-service-host-name": "aai",
                    "aai-base-url": "https://{}:{}/aai/v23",
                    "aai-service-port": "",
                    "aai-username": "",
                    "aai-password": "",
                    "aai-api-timeout":
                },
                "async-access-info": {
                    "semaphore": "",
                    "connections": "",
                    "retry": ""
                },
                "from-tier1": "",
                "selected-move": "",
                "log-level": ""
            }
        }
    }
}
```

Código 9.10: *Payload* para la constula al servicio cds-blueprints-processor-http para la ejecución del CBA de configuración

9.2.3. Algoritmo para la generación de las plantillas de configuración

Al igual que en el primer flujo, en el Código 9.9 puede verse que existe un objeto debajo de la llave “operations” con nombre “process”, el cual indica qué función debe de ejecutarse, mientras que el valor de la llave “blueprint-action” indica que la clase que deberá de utilizarse es “BGPLoadBalancerExecutor”, la cual está definida dentro del archivo “bgp_load_balancer_executor.py”. En el Código 9.11 puede verse un fragmento del código que implementa

la función “process” de la clase “BGPLoadBalancerExecutor” antes mencionada.

```
# Imports
...

class BGPLoadBalancerExecutor(AbstractScriptFunction):

    def __init__(self):
        # Init method
        ...

    def process(self, execution_request):
        is_connection = False
        try:
            common_header =
                json.loads(json_format.MessageToJson(execution_request.commonHeader))
            input_json =
                json.loads(json_format.MessageToJson(execution_request.payload))[
                CbaCommonKeywords.MAIN_CBA_PAYLOAD]
            skip_input_vars = input_json.get(CbaCommonKeywords.SKIP_INPUT_KEY)
            input_params = input_json.get(CbaCommonKeywords.INPUT_PARAMS)
            workflow_name = input_json.get(CbaCommonKeywords.CBA_WORKFLOW_NAME)
            final_resp_data = {
                "py-executor-originator-id": common_header.get("originatorId"),
                "py-executor-request-id": common_header.get("requestId"),
                "py-executor-subrequest-id": common_header.get("subRequestId"),
                "bgp-load-balancer-status" : {
                    "bgp-load-balancer-response" : {},
                    "response-code":"",
                    "response-data":"",
                    "custom-response-attributes":[],
                    "status":"success"
                }
            }

            input_params_obj = GenericInputParameterValidator(
                workflow_name,
                input_params,
                skip_input_vars
            )
            validated_info = input_params_obj.process_validation()

            if validated_info.get('status'):
                host = input_params.get("pnf-access-information").get("ip-address")
                user = input_params.get("pnf-access-information").get("user-name")
                passwd = input_params.get("pnf-access-information").get("password")
                port = input_params.get("pnf-access-information").get("port")
                pnf_input_data = {
                    "pnf-device-ip-address": host,
                    "pnf-device-access-port": port,
                    "pnf-device-username": user,
                    "pnf-device-password": passwd
                }

                self.aai_resources_access_info =
                    input_params.get("aai-resources-access-info")
                aai_host =
                    self.aai_resources_access_info.get("aai-service-host-name")
                aai_port = self.aai_resources_access_info.get("aai-service-port")
                self.aai_url = (self
                    .aai_resources_access_info
                    .get("aai-base-url")
                    .format(aai_host, aai_port))
                pnf_data_input_params = input_params.get("pnf-access-information")
                async_input_params = input_params.get("async-access-info")
                from_tier1 = input_params.get("from-tier1")
                selected_move = json.loads(input_params.get("selected-move"))
                self.logger.info(f"Selected Move: {selected_move}")
```

```
            router_device = DiscoverJuniperRouterDevice(pnf_input_data)
            router_device_service = JuniperRouterDiscoveryService(
                self.aai_resources_access_info,
                router_device,
                pnf_data_input_params,
                async_input_params
            )

            country = self.get_country(host)
            prefixes_for_country = [(prefix['prefix'], prefix['toTier1'])
                for prefix in selected_move
                if country == prefix['asn']]

            result, response, is_connection = (router_device_service
                .discover_config_info(prefixes_for_country, from_tier1))
            jinja_response = self.queryHandler.parse_plain_text(
                self.configuration_existing_term_template,
                response=response
            )

            if result:
                final_resp_data["bgp-load-balancer-status"] =
                    self.resp_obj.discover_device_response(
                    CbaCommonKeywords.RESULT_SUCCESS_STATUS,
                    CbaCommonKeywords.SUCCESS_STATUS_CODE,
                    CbaCommonKeywords.PNF_DISCOVERY_SUCCESS_MSG + host
                )
                execution_response = executor_utils.success_response(
                    execution_request,
                    final_resp_data,
                    CbaCommonKeywords.BP_SUCCESS_STATUS_CODE
                )

        else:
            ...

    except Exception as err:
        ...

    yield execution_response
```

Código 9.11: Clase para la ejecución del CBA de configuración

Como puede observarse, el código puede separarse en cuatro partes. La primera parte, al igual que el primer flujo para descubrimeinto de capacidades disponibles, consta de la obtención y *parseo* de la información proveniente de la llamada gRPC. La segunda parte consiste en el uso de la información del PNF para obtener la información del país al que pertenece, la cual es utilizada para poder filtrar los movimientos que deben de hacerse en ese equipo en específico, puesto que pueden haber varios prefijos involucrados en el movimiento para el balanceo y que no todos correspondan al equipo que se está configurando; esta consulta se realiza de forma asíncrona hacia la A&AI a través de la función `get_country(self, ip_address)`, la cual puede verse más detalladamente en el Código 15.11. La tercera parte consiste en utilizar una instancia de la clase "JuniperRouterDiscoveryService" para poder acceder al equipo físico para obtener información sobre los sistemas lógicos, la política en la que se encuentra actualmente el prefijo y las comunidades que existen. Los sistemas lógicos obtenidos se utilizan para identificar el sistema lógico de Internet; la política en la que se encuentra el prefijo se utiliza para poder eliminar el prefijo de esa interfaz para colocarlo en una nueva; y las comunidades existentes se utilizan para formar el comando que agrega la comunidad correspondiente al *Tier 1* hacia el cual se va a mover el prefijo. El código de esta clase puede

verse más detallado en la Sección 15.6. Finalmente, la cuarta parte consiste en la generación de las plantillas de configuración para cada uno de los prefijos que aplican al PNF a través del uso del motor de plantillas Jinja; en el Código 9.12 puede observarse la plantilla utilizada.

Como puede observarse en el Código 9.12, el motor de plantillas permite pasar un diccionario de contexto, el cual, en este caso se llama “response”. Este parámetro se pasa al momento de llamar la función que utiliza Jinja, como puede verse en el Código 9.11, en donde puede observarse que se utiliza una función definida en la clase `QueryHandler`, la cual puede ser vista más a detalle en la Sección 15.6.

```
{% for move in response['moves'] -%}
set logical-systems {{ response['logical-system'] }} policy-options policy-statement
    {{ move['policy-statement'] }} term {{ move['term'] }}_temp from protocol bgp
set logical-systems {{ response['logical-system'] }} policy-options policy-statement
    {{ move['policy-statement'] }} term {{ move['term'] }}_temp from route-filter {{
    move['prefix'] }} exact
{% for community in move['communities'] -%}
set logical-systems {{ response['logical-system'] }} policy-options policy-statement
    {{ move['policy-statement'] }} term {{ move['term'] }}_temp then community add {{
     community }}
{% endfor -%}
set logical-systems {{ response['logical-system'] }} policy-options policy-statement
    {{ move['policy-statement'] }} term {{ move['term'] }}_temp then accept
{% endfor %}
```

Código 9.12: Plantilla de configuración de los equipos locales Juniper para el balanceo de tráfico de internet

CAPÍTULO 10

Integración de herramientas de monitoreo

Uno de los objetivos específicos de este trabajo de graduación es el uso del *stack* tecnológico de Grafana dentro del caso de uso desarrollado. Para cumplir este objetivo, se utilizaron tres de las soluciones ofrecidas por Grafana Labs:

- Grafana: Es la herramienta principal de Grafana Labs, la cual es utilizada para la visualización de datos a través de una interfaz gráfica, la cual permite crear y organizar tableros para la fácil interpretación de la información.
- Loki: Es un sistema de agregación de *logs*, el cual está diseñado para poder obtener, guardar y hacer consultas de *logs* provenientes de los aplicativos y la infraestructura.
- Prometheus: Es un sistema de monitoreo de código abierto, el cual está diseñado para obtener métricas de los aplicativos y la infraestructura. Este módulo no es realmente de Grafana Labs, sin embargo, Grafana Labs promueve su uso y diseña sus soluciones para funcionar desde el inicio con Prometheus.

10.1. Creación de tableros en Grafana

Con estas herramientas, se logró crear un tablero con información sobre la salud del ambiente sobre el cual está corriendo el caso de uso y otro sobre los *logs* que produce Camunda y el CDS. Para esto, se siguieron los siguientes pasos:

1. Ingresar a la página principal de Grafana, la cual puede verse en la Figura 35.
2. Seleccionar la opción "*Dashboards*" en el menú lateral izquierdo, la cual puede verse en la Figura 36.

3. Seleccionar la opción "*New*" en la esquina superior derecha, la cual puede verse en la Figura 37. En esta opción puede escogerse realizar un nuevo folder o un nuevo tablero, en este caso, primero se escogió la opción de nuevo folder para poder tenerlo más ordenado y luego se repitió el proceso dos veces para crear dos tableros dentro de este folder.

4. Entrar a un tablero creado, el cual puede verse en la Figura 38.

5. Crear un panel dentro del tablero, lo cual puede verse en la Figura 39.

6. Editar el panel creado para que tenga una gráfica adecuada para mostrar la información deseada, lo cual puede verse en la Figura 40.

7. Editar el *query* realizado dentro del panel para poder obtener y mostrar la información deseada, lo cual puede verse en la Figura 41.

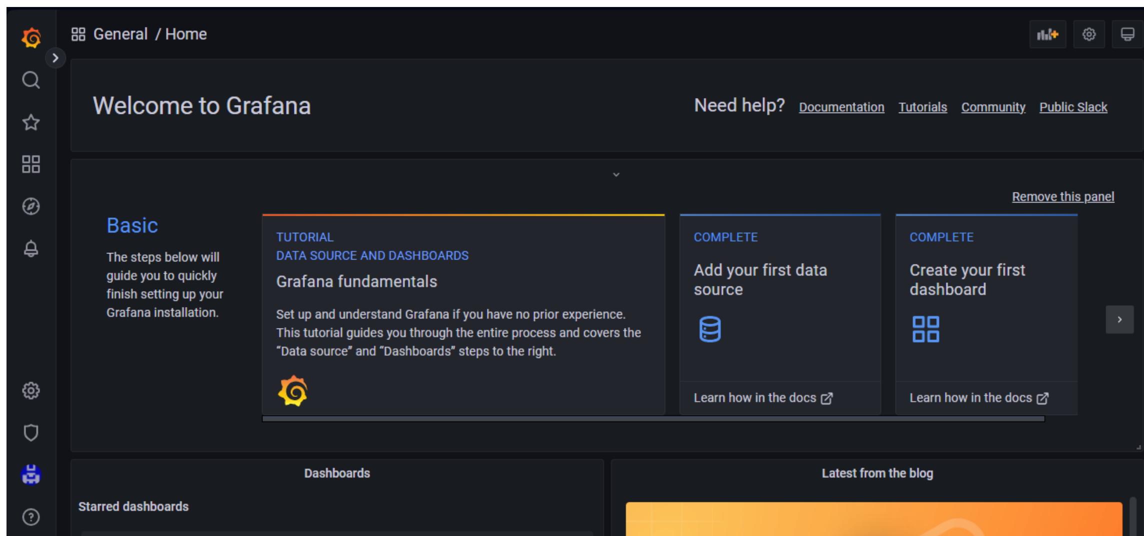


Figura 35: Página principal de Grafana.

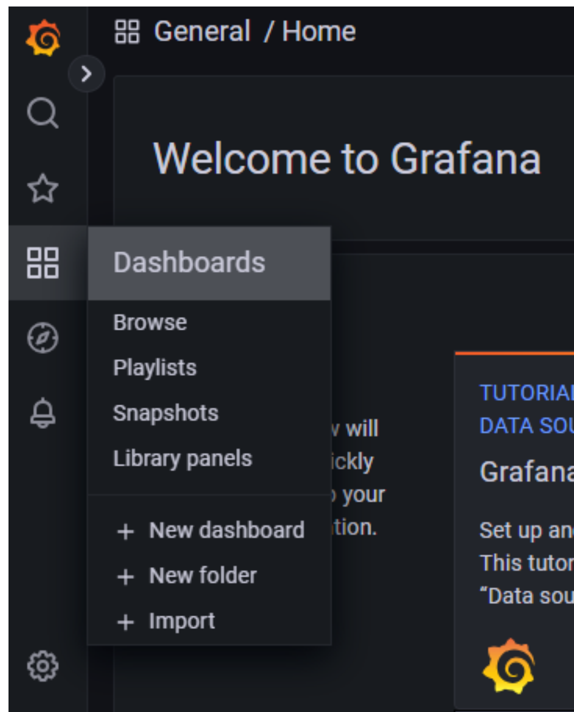


Figura 36: Opción de Grafana para ver y crear tableros.

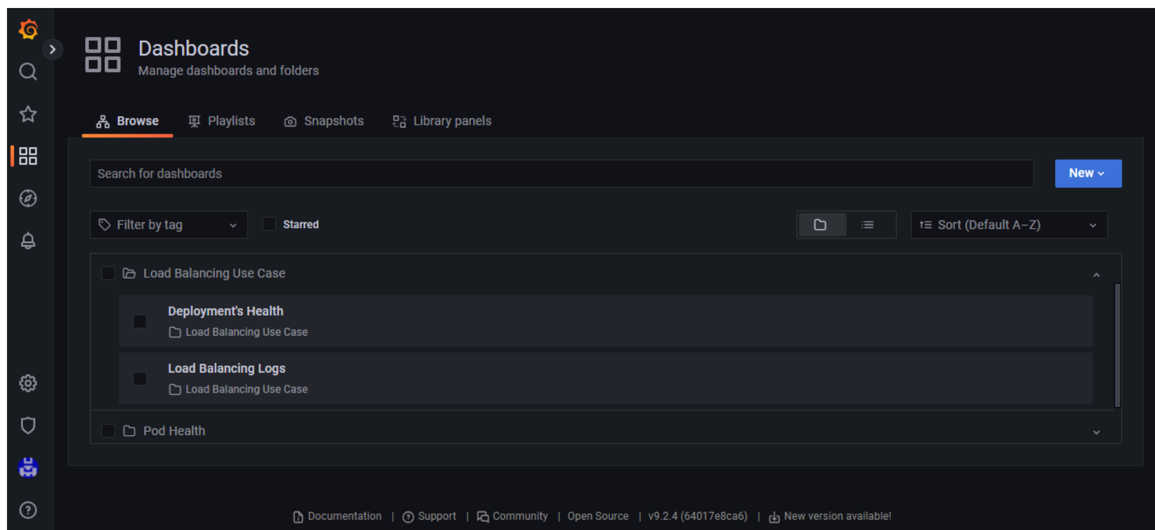


Figura 37: Folders de Grafana con los tableros.

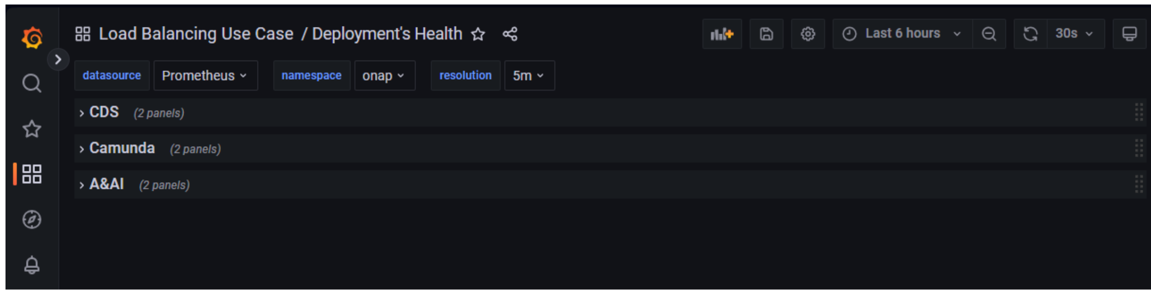


Figura 38: Vista principal de un tablero de Grafana.

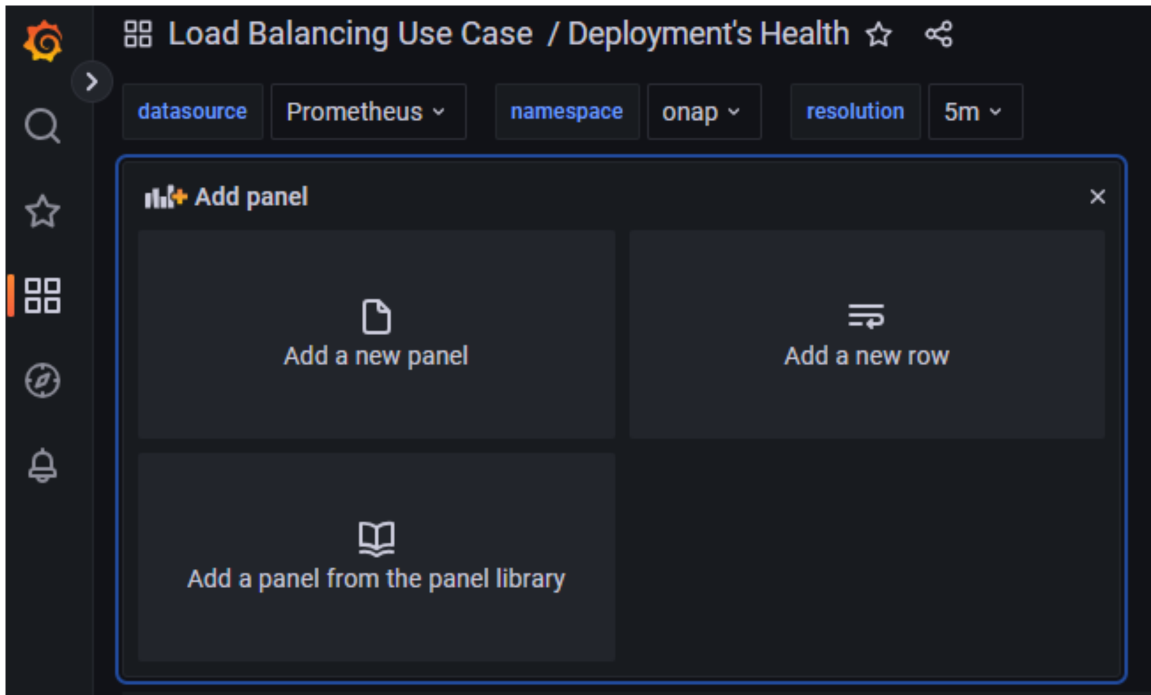


Figura 39: Creación de un nuevo panel para un tablero de Grafana.

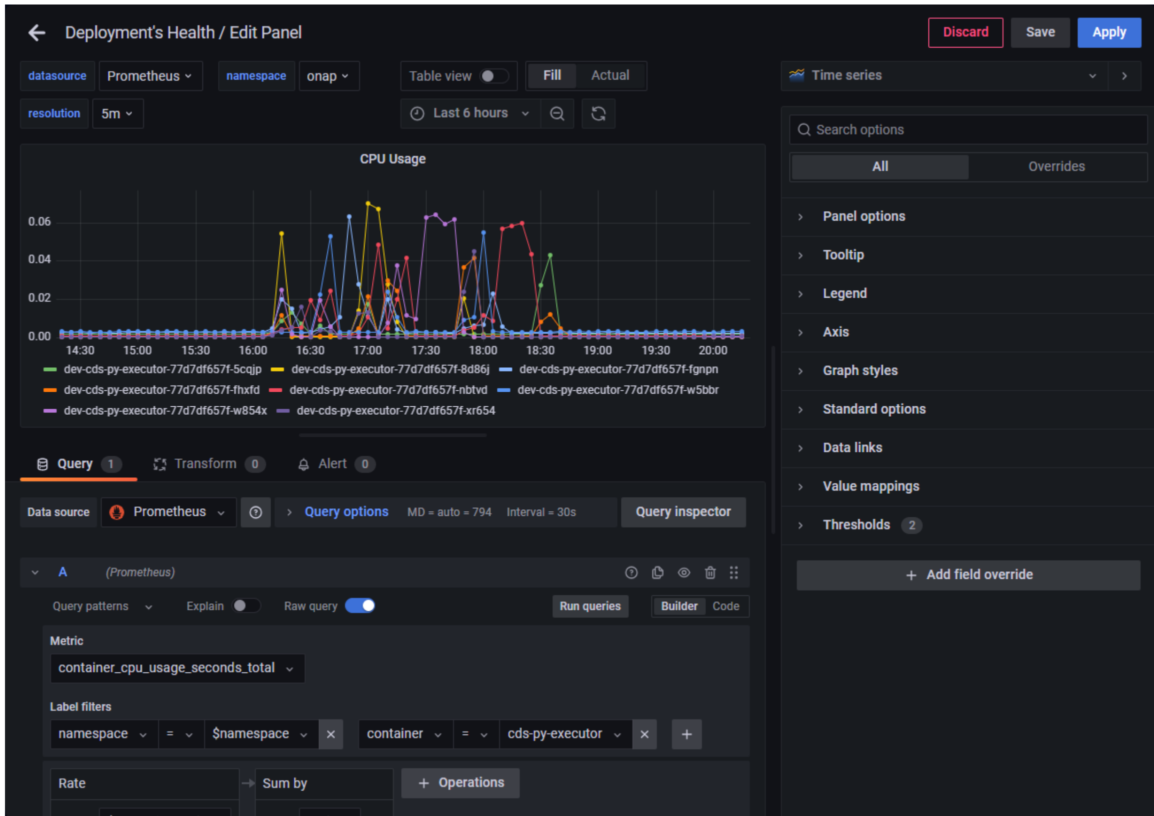


Figura 40: Edición de un panel para un tablero de Grafana.

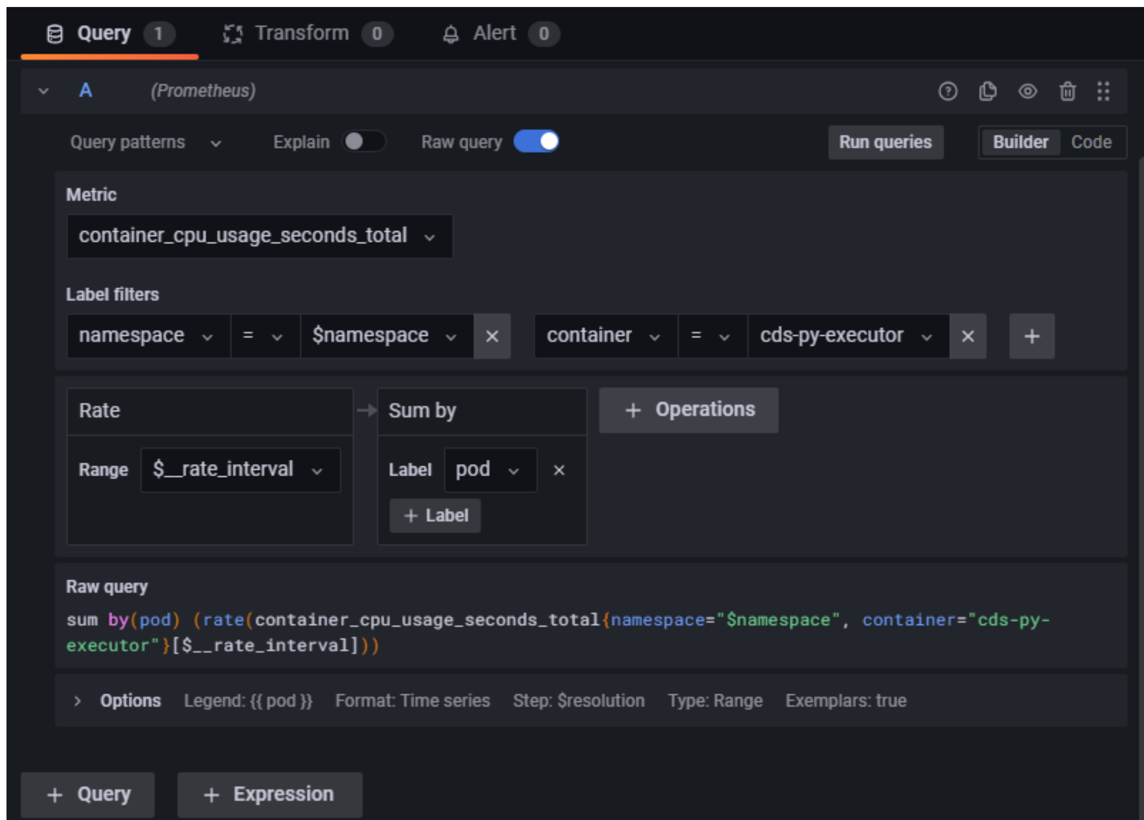


Figura 41: Edición de un *query* para la obtención de información para un panel.

Como se mencionó antes, con estos pasos se logró crear dos tableros, uno para monitorear la salud de la red de la red y otro para monitorear los *logs* de Camunda y el CDS. El primer tablero puede verse en la Figura 42, mientras que el segundo tablero se muestra en las figuras 43 y 44 para poder distinguir los *logs* de Camunda de los del CDS. Como puede verse en el primer tablero, la información que se está monitoreando es el uso de CPU y de memoria RAM que tiene cada *pod* en específico, separado por módulo de ONAP (CDS, Camunda y A&AI). Para este tablero se utilizó como fuente de información a Prometheus, el cual se encarga de estar *polleando* la información de la infraestructura y su uso a un nivel de granularidad que permite desglosar la información según *pod*. Por otro lado, para el segundo tablero, puede verse que tanto para Camunda como para el CDS, se agregó el prefijo "Use case: Load Balancing - " a los *logs* para poder distinguirlos y utilizar este filtro en el *query* de la información del panel (como se mostró en la Figura 41). Para este tablero se utilizó como fuente de información a Loki, el cual se encarga de estar *polleando* los *logs* de los aplicativos para poder mandárselos a Grafana.

Figura 42: Tablero de Grafana para el monitoreo de la salud del ambiente.

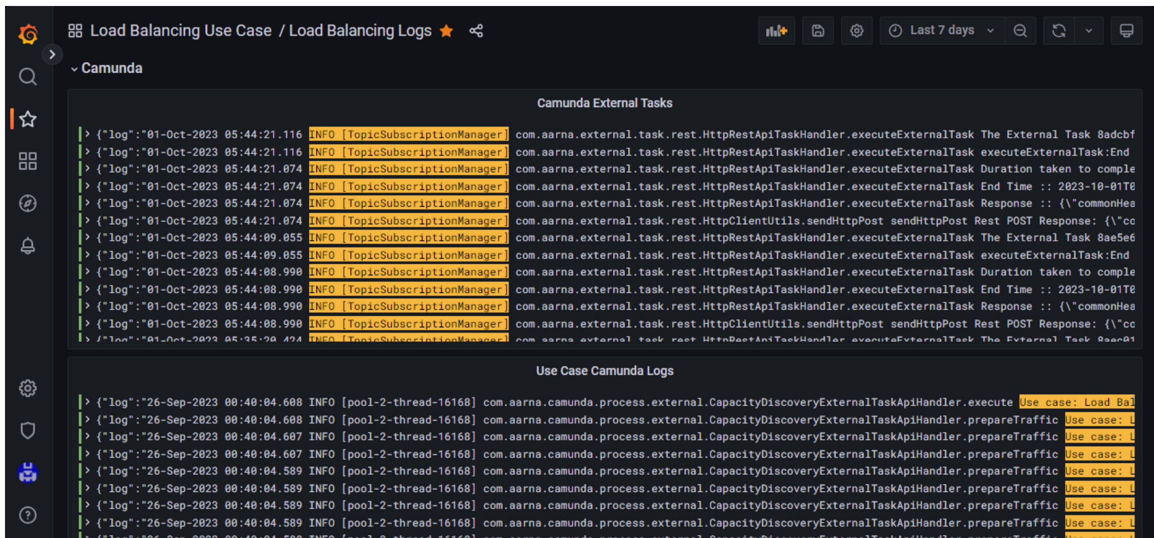


Figura 43: Tablero de Grafana para ver los *logs* de Camunda.

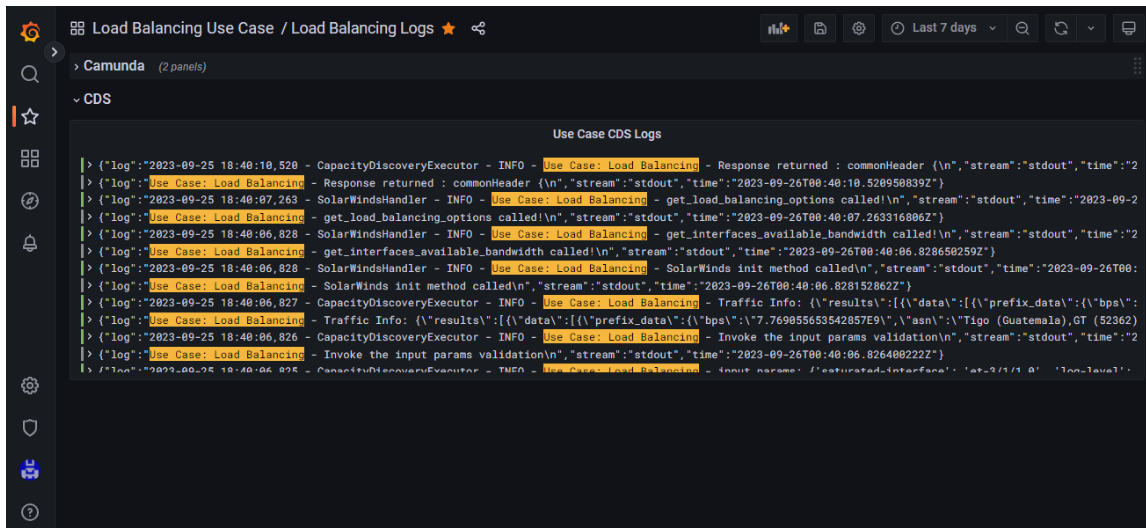


Figura 44: Tablero de Grafana para ver los *logs* del CDS.

CAPÍTULO 11

Funcionamiento integral del caso de uso

A lo largo de este trabajo de graduación se ha mencionado y explicado el desarrollo y funcionamiento de dos flujos para lograr realizar el balanceo de tráfico de internet. Además de esto, es evidente que de las respuestas que da el primer flujo, debe de escogerse una para utilizarla en el segundo flujo. Esto está pensado de esta forma por muchos motivos, como por ejemplo, al inicio del uso de la herramienta, es preferible que la herramienta solo sugiera cambios y que no los ejecute automáticamente para poder realizar pruebas de efectividad y rendimiento; por otro lado, dar opciones permite que un ingeniero pueda introducir un factor de decisión humana para los casos en los que haya condiciones excepcionales y que no sigan realmente un patrón que pueda programarse; y finalmente, el punto anterior también abre a la oportunidad de, en fases futuras del proyecto, utilizar algoritmos de *Machine Learning* que permita escoger la mejor opción en base a un historial de qué opciones han sido escogidas por los ingenieros.

A pesar de estos puntos, en este trabajo de graduación es necesario ver el funcionamiento integral de la solución. Por este motivo, se desarrolló una aplicación con Python utilizando el *framework* FastAPI. Esta aplicación levanta una REST API, la cual cuenta con un *endpoint* de descubrimiento de capacidades y otro para realizar la configuración, similar a los de Camunda. De forma interna, estos *endpoints* ejecutan el *POST request* a la API de Camunda; en el caso del primer flujo, la única diferencia con hacerlo directamente hacia Camunda es que se agrega una bandera indicando que la petición viene de esta aplicación, esta bandera es utilizada dentro del flujo de Camunda para realizar el *POST request* hacia el segundo *endpoint*; en el caso del segundo flujo, la diferencia a ejecutar el proceso de configuración directamente con Camunda es que la entrada de este flujo es el resultado del flujo anterior, dentro del cual vienen las opciones para poder realizar balanceo, de las cuales se utiliza la primera opción para poder construir el *payload* del flujo de configuración de Camunda para así poder ejecutarlo ya con un movimiento seleccionado de forma automática.

11.1. Desarrollo de REST API para la ejecución completa del caso de uso

Como se mencionó anteriormente, la REST API implementada en este trabajo fue desarrollada con el *framework* FastAPI de Python. En el Cuadro 5 se muestra la estructura de directorios de la aplicación.

Cuadro 5: Árbol de directorios y archivos del proyecto bpmn-commons

En el caso de esta aplicación, casi todo el funcionamiento se encuentra en el archivo `main.py`, sin embargo, puesto que se deseaba separar ciertas funcionalidades en otros archivos, se decidió crear un directorio `app/`, dentro del cual se encuentra el archivo `__init__.py`, permitiendo que se ejecute la aplicación a través del folder como un paquete en lugar del archivo directamente; por el uso que se le está dando a esta aplicación, este archivo puede estar vacío. Por otro lado, en el Código 11.1 se ve el contenido del archivo `main.py`.

```
import requests
import json
import aiohttp
import asyncio
import copy

from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

from .schemas import CamundaStartProcess, CamundaCompleteTask
from .constants import Constants

app = FastAPI()

origins = ["*"] # accept any domain in the CORS

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.get("/api/v1")
async def root():
    return {"message": "Load Balancing API Gateway"}

@app.post("/api/v1/capacities")
async def find_capacities(process_start: CamundaStartProcess):
    result = await camunda_start_process(process_start.dict(),
        "PROCESS_TRAFFIC_LOAD_BALANCER")
    return {
        "message": "Find capacities",
        "camunda response": result
```

```python
    }

@app.post("/api/v1/configurations")
async def find_configurations(process_response: CamundaCompleteTask):
    discovery_response = process_response.dict()\
                        ['payload']['capacity-discovery-response']\
                        ['response-data']['capacity-discovery-status']\
                        ['capacity-discovery-response']
    selected_move = discovery_response['move-groups'][0][0]
    from_tier1 = discovery_response['from-tier1']
    variables = json.dumps({
        "fromTier1": from_tier1,
        "selectedMove": [
            {
                "toTier1": move['to-tier1'],
                "prefix": move['prefix'],
                "asn": move['asn']
            } for move in selected_move
        ]
    })

    payload = copy.copy(Constants.CONFIG_PAYLOAD)
    payload['variables']['payload']['value'] = variables

    result = await camunda_start_process(payload, "PROCESS_CONFIGURE_LOAD_BALANCER")
    return {
        "message": "Find capacities",
        "variables": variables
    }

async def camunda_start_process(payload, process):
    async with aiohttp.ClientSession() as session:
        url = Constants.BASE_CAMUNDA_URL + f"/process-definition/key/{process}/start"
        headers = {
            "Accept": "*/*",
            "Content-Type": "application/json",
            "Authorization": "Basic c291c2VyOm15cGFzc3dvcmQ="
        }

        print(payload)

        async with session.post(url=url, headers=headers, data=json.dumps(payload),
            verify_ssl=False, timeout=1000) as resp:
            result = await resp.json()

    return result
```

Código 11.1: Archivo main.py para el funcionamiento del *gateway*

Como puede verse en el Código 11.1, la aplicación cuenta con tres *endpoints*:

- `/api/v1`: Este *endpoint* es el *default* de la aplicación, el cual se utiliza para verificar que la aplicación está corriendo correctamente.
- `/api/v1/capacities`: Este *endpoint* es el que se utiliza para ejecutar el flujo de descubrimiento de capacidades disponibles.
- `/api/v1/configurations`: Este *endpoint* es el que se utiliza para ejecutar el flujo de configuración.

Como puede observarse, los *endpoints* `/api/v1/capacities` y `/api/v1/configurations` utilizan el método HTTP POST, por lo pueden tener un *payload* de entrada; en FastAPI pueden

definirse estos *payloads* con clases de Python. En el Código 11.2 puede verse el contenido del archivo `schemas.py`, en donde pueden verse las clases de Python utilizadas para la definición de la estructura esperada de los *payloads* de cada *endpoint*.

```
from pydantic import BaseModel
from typing import Union

class CamundaStartProcess(BaseModel):
    variables: dict

class CamundaCompleteTask(BaseModel):
    commonHeader: dict
    actionIdentifiers: dict
    correlationUUID: Union[str, None] = None
    status: dict
    payload: dict
```

Código 11.2: Definición de los esquemas para los *payloads* de entrada de los *endpoints*

En el caso del *endpoint* `/api/v1/capacities`, se recibe un objeto de tipo `CamundaStartProcess`, el cual recibe un diccionario como entrada; en este caso, el diccionario es obligatorio, ya que no se está usando la función `Union` de la librería `typing`. Este diccionario es exactamente el mismo que se pasa al *endpoint* original de Camunda, pero con una variable extra llamada "fromPortal". Por otro lado, en el caso del *endpoint* `/api/v1/configurations`, se recibe un objeto de tipo `CamundaCompleteTask`, el cual recibe cuatro diccionarios y un *string* como entrada; en este caso, los diccionarios son obligatorios, mientras que el *string* es opcional, ya que se está usando la función `Union` para poder indicar que el valor puede ser `None`. Nótese que estas entradas son las mismas que las salidas del CDS; esto es así, porque lo que devuelve el CDS del flujo de descubrimiento de capacidades es lo que se manda a este segundo flujo.

11.2. Simulación de la alarma de SolarWinds

Para poder realizar estas pruebas integrales, se configuró una nueva alarma (siguiendo los pasos para la creación de una alarma de SolarWinds mostrados en la Subsección 8.4.1). En esta ocasión, la API que se creó a través del código de FastAPI utiliza HTTP en lugar de HTTPS, por lo que la función nativa de SolarWinds para realizar *POST requests* sí funciona y puede configurarse como se ve en la Figura 45. En el Código 11.3 puede verse el contenido del *payload* de la alarma; como se mencionó antes, este *payload* es idéntico al mostrado en el código 8.1, pero con la bandera extra indicando que va a utilizarse la solución integral.

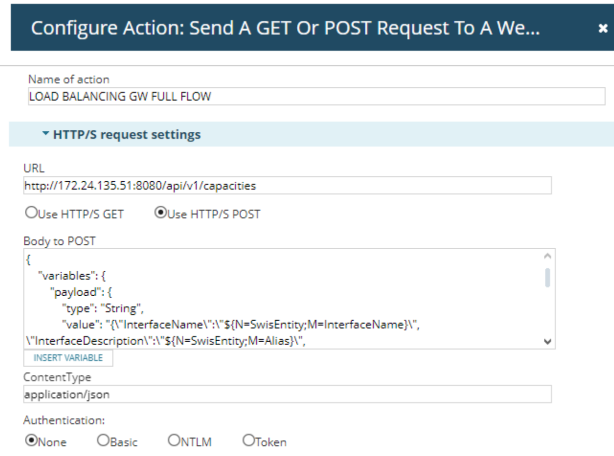


Figura 45: Vista de la alarma de SolarWinds para la simulación de la ejecución del flujo integral a través del aplicativo en FastAPI.

```
{
    "variables": {
        "payload": {
            "type": "String",
            "value": "{\"InterfaceName\":\"${N=SwisEntity;M=InterfaceName}\", \"
    InterfaceDescription\":\"${N=SwisEntity;M=Alias}\", \"pnf_ipv4_address\":\"${N=
    SwisEntity;M=Node.IP_Address}\"}"
        },
        "cbaVersion": {
            "type": "String",
            "value": "13.11.2023-6"
        },
        "semaphore": {
            "type": "String",
            "value": "5"
        },
        "connections": {
            "type": "String",
            "value": "2"
        },
        "retry": {
            "type": "String",
            "value": "3"
        },
        "logLevel": {
            "type": "String",
            "value": "info"
        },
        "threshold": {
            "type": "String",
            "value": "0.15"
        },
        "maxMoves": {
            "type": "String",
            "value": "5"
        },
        "fromPortal": {
            "type": "Boolean",
            "value": true
        }
    }
}
```

Código 11.3: *Payload* para la ejecución del flujo integral a través del aplicativo en FastAPI

Luego, se utilizó la capacidad de SolarWinds para simular las alarmas con datos reales del momento de cada capacidad. En las figuras 46 y 47 puede verse cómo se realiza esta simulación. Una vez se creó la alarma, se presiona el botón de simulación, luego se busca la interfaz sobre la cual se quiere correr la simulación y finalmente se ejecuta la simulación.

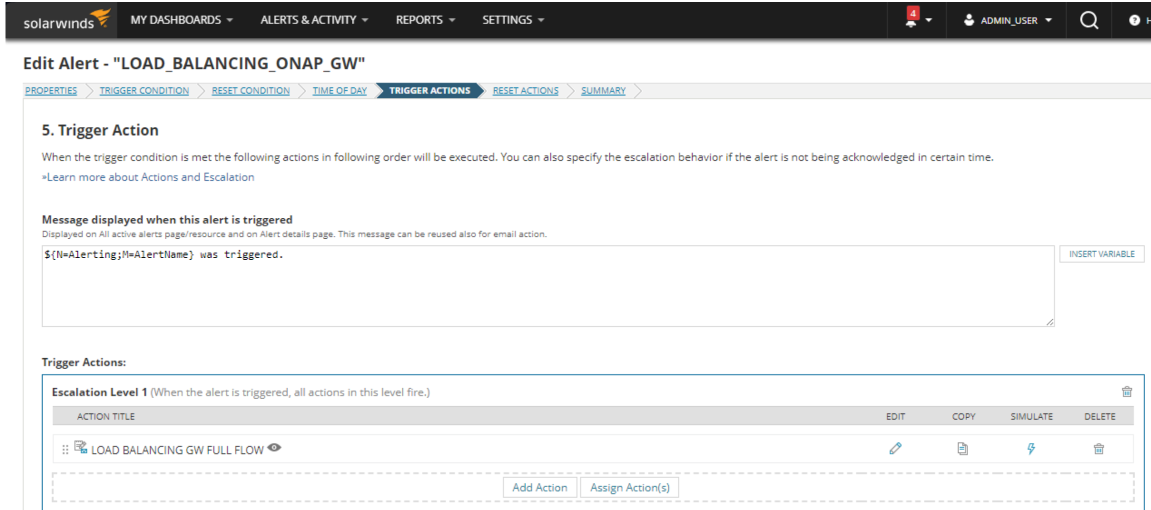


Figura 46: Vista de la alarma de SolarWinds para la simulación de la ejecución del flujo integral a través del aplicativo en FastAPI.

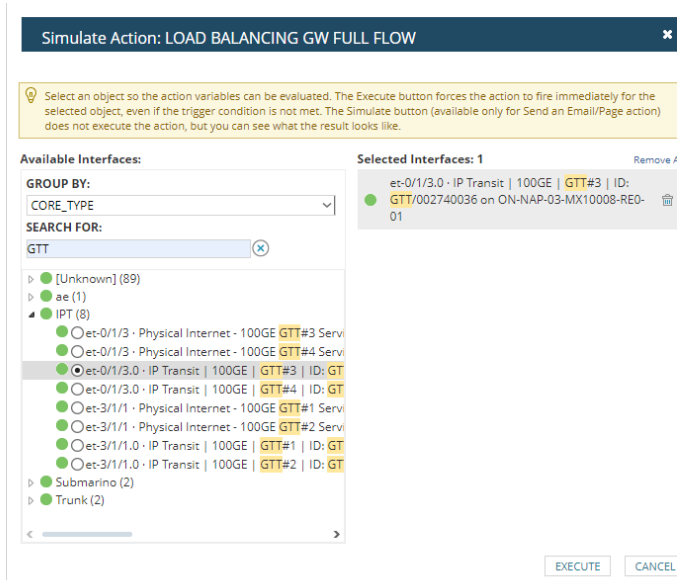


Figura 47: Ejecución de la simulación del flujo integral sobre una interfaz específica.

11.3. Análisis de resultados

Para esta demostración del funcionamiento integral del caso de uso, se realizaron 20 simulaciones con SolarWinds, como puede verse en el Cuadro 6. El tiempo de ejecución

para el flujo de descubrimiento de capacidades presenta un promedio de aproximadamente 37.797s, con una desviación estándar de 142.435s. Esta desviación estándar tan grande se debe a que en la prueba número 15 presentó un tiempo de ejecución de dos órdenes de magnitud más grande que el resto de tiempos, lo cual se debe a que esta prueba requirió mover 12 prefijos de los 14 disponibles que habían en ese momento, siendo el doble del segundo movimeinto más grande.

Esta diferencia de tiempo se debe a que la cantidad de posibilidades de movimientos que el sistema debe de verificar está dada por la Ecuación 11.1, en donde α es la cantidad máxima de prefijos que tiene la interfaz saturada de los cuales se está escogiendo mover, n es la cantidad de prefijos que se está tratando de mover, β es la cantidad de interfaces disponibles para poder realizar el movimiento y x es el tamaño del grupo en el cual se consiguió hacer un balanceo exitoso. Esta ecuación puede separarse en tres partes, la primera parte es la cantidad de combinaciones posibles de prefijos que pueden moverse, la cual está dada por la expresión $\frac{\alpha!}{n! \times (\alpha - n)!}$, que es la ecuación de la operación "n escoge k" utilizada en el algoritmo. Luego, la segunda parte de la ecuación representa cómo pueden moverse los prefijos hacia las interfaces disponibles, la cual está dada por la expresión β^n, la cual surge porque en el código se utiliza dos veces la función "product" de Python, lo cual puede verse a más detalle en la Sección 15.5. Finalmente, la tercera parte de la ecuación es que las primeras dos partes están dentro de una sumatoria, lo cual se debe a que la cantidad de opciones calculada de las primeras dos partes de la ecuación son las opciones para grupos individuales, sin embargo, conforme se van probando grupos, los tiempos se van acumulando. Nótese que estas son la cantidad máxima de opciones que se tienen cuando se llegan a buscar grupos de tamaño x, el mejor caso estaría dado por $f(x-1)+1$, puesto que sería la iteración completa del tamaño anterior más uno porque se encuentra en la primera opción disponible del tamaño final. Esto puede generar una gran diferencia para resultados con los mismos tamaños de grupos. Estas pruebas utilizaron como valor fijo 14 interfaces de destino posibles, puesto que la red utilizada cuenta con 15 y la interfaz saturada en sí es eliminada de las opciones de destino; el resto de datos pueden variar según las condiciones de la red en el momento que se corren las pruebas.

$$f(x) = \sum_{n=1}^{x} \frac{\alpha!}{n! \times (\alpha - n)!} \times \beta^n \tag{1}$$

Cantidad de posibles movimientos a validar para un grupo de tamaño x

Existen dos casos en los cuales se puede llegar a necesitar mover tantos prefijos a la vez:

1. Si la cantidad de tráfico que se está intentando mover de la interfaz saturada es muy grande.
2. Si el consumo de cada prefijo de forma individual es muy bajo.

Puesto que el balanceo de tráfico de internet se dará cuando haya mucha demanda y las interfaces se saturen, indicando que los prefijos están consumiendo bastante ancho de banda, el segundo caso está eliminado y no debería de ocurrir una vez este caso de uso

se encuentre en producción. Por otro lado, la alarma de SolarWinds está configurada para que la acción se ejecute luego de que la interfaz saturada lleve más de 1 minuto alarmada. Puesto que en 1 minuto la interfaz, bajo situaciones ordinarias, no es capaz de subir tanto, el primer caso no es probable. Por esto, el tiempo de la prueba 15 puede considerarse como un caso excepcional, por lo que se calculó que el promedio del tiempo de ejecución para el flujo de descubrimiento de capacidades sin tomar en cuenta esta prueba es de aproximadamente 6.102s, con una desviación estándar aproximada de 0.330s.

Por otro lado, el tiempo de ejecución para el flujo de configuración presenta un promedio de aproximadamente 83.417s con una desviación estándar de aproximadamente 124.705s. En este caso, la desviación estándar grande se debe al tiempo de respuesta de los equipos de la red en donde se realizan las configuraciones. En el caso de las pruebas 13, 15, 16, 18, 19 y 20 se logró identificar que alguno de los *routers* involucrados en el balanceo estaba lento y respondía más lento que los demás. Esto se logró identificar gracias a que, dependiendo de en dónde tiene que realizarse el balanceo (Guatemala, El Salvador, etc.) hay varios equipos que funcionan como espejos; en el caso de las primeras 12 pruebas, todos los equipos que funcionaban como espejos entre sí respondieron en el mismo tiempo aproximadamente, mientras que en las pruebas antes mencionadas eso no pasó, sino que había uno o más equipos que estaban respondiendo considerablemente más lento que antes. Al eliminar estas pruebas para determinar cómo se comportaría el sistema cuando los equipos estén respondiendo de forma óptima, se encontró que el tiempo de ejecución promedio es de aproximadamente 15.490s con una desviación estándar aproximada de 0.933s.

Finalmente, el tiempo de ejecución total del caso de uso presenta un promedio de aproximadamente 121.356s con una desviación estándar aproximada de 252.241s con los datos fuera de serie y de 21.783s con desviación estándar aproximada de 4.388s sin los datos fuera de serie. Esto quiere decir que en el caso en que los equipos de red estén en buenas condiciones, el tiempo promedio en que el sistema podría dar opciones para el balanceo de tráfico es de aproximadamente 21.783s. Acorde a ingenieros del NOC del ISP para el cual se está desarrollando este trabajo, un ingeniero con mucha experiencia puede tardarse un mínimo de 15 minutos, mientras que alguien que acaba de entrar a la posición puede tardarse hasta 2 horas. Esto quiere decir que el sistema desarrollado en este trabajo de graduación muestra un *speedup* de entre 41.3 y 330.5 veces, dependiendo de la experiencia del ingeniero que se esté comparando.

No. Prueba	Tiempo del flujo de descubrimiento (s)	Tiempo del flujo de configuración (s)	Tiempo total (s)	Prefijos movidos
1	6.059	15.427	22.663	4
2	9.606	22.322	33.105	2
3	5.654	11.207	16.933	2
4	5.667	19.096	24.769	6
5	4.818	15.518	20.347	3
6	7.414	11.776	19.213	1
7	5.450	12.674	18.132	2
8	4.911	17.924	22.847	4
9	6.678	14.122	20.802	2
10	4.999	13.378	18.396	2
11	5.731	13.789	19.537	2
12	6.405	16.078	22.567	2
13	5.700	202.838	208.569	3
14	6.324	21.241	27.598	3
15	642.92	498.725	1141.673	12
16	5.924	176.409	182.381	1
17	5.712	12.313	18.053	1
18	5.143	179.860	185.012	4
19	5.368	184.415	189.805	4
20	5.465	209.234	214.720	4

Cuadro 6: Tiempos de ejecución de las pruebas de funcionamiento integral del caso de uso

CAPÍTULO 12

Conclusiones

- Se conectó la plataforma ONAP a las herramientas de monitoreo de red SolarWinds y Kentik a través de APIs, siguiendo el estándar *Cloud Native* para aplicaciones con arquitectura de microservicios.
- Se logró implementar un flujo de automatización que, en conjunto con SolarWinds y Kentik, permite la recolección de datos de tráfico de red y su uso para determinar posibilidades para balancear el tráfico.
- Se logró implementar un flujo de automatización que muestra la plantilla de configuración utilizada para balancear el tráfico de red, a través de la comprensión del flujo completo de balanceo de cargas que realiza un ISP a nivel de BGP.
- Se diseñó un tablero de información en la *stack* tecnológico de Grafana, en el cual se puede visualizar los *logs* de los procesos que están corriendo en la plataforma ONAP al ejecutar los flujos implementados.
- Se diseñó un tablero de información en la *stack* tecnológico de Grafana, en el cual se puede visualizar la salud del ambiente utilizado para la implementación de la plataforma ONAP.
- El funcionamiento integral final del aplicativo desarrollado únicamente logra implementar una semiautomatización, puesto que la empresa proveedora de servicios para la cual se está realizando el desarrollo decidió que desea probar los resultados de los flujos de automatización antes de que estos se ejecuten en producción, debido a la sensibilidad de los procesos que se realizan. Por esto, el proyecto únicamente muestra las plantillas de configuración de los equipos, mas no las aplica directamente.

CAPÍTULO 13

Recomendaciones

- Puesto que la información utilizada para generar los tableros en Grafana provienen de los *logs* de los servicios, se recomienda que se diseñe un estandar para los mismos, de tal forma que luego sea sencillo transformar la data y sea posible generar tableros con un mayor impacto.
- En este trabajo se encontró que la herramienta SolarWinds no es capaz de hacer requerimientos POST o GET a servidores HTTPS que cuentan con un certificado autofirmado, por lo que se mostró una solución alternativa en donde se ejecutó un *script* de Python, dentro del cual se utilizó un comando que permite obviar este tipo de certificados. Sin embargo, se identificó que es posible modificar el código fuente de SolarWinds para poder hacer una corrección permanente, por lo que, si la empresa cuenta con una licencia de SolarWinds y se le paga soporte, se recomienda escalar el tema y reportarlo para que se solucione de una forma más directa y que la función nativa de SolarWinds de realizar requerimientos POST y GET de los resultados esperados.
- En este trabajo de graduación fue una gran limitante la disponibilidad para poder realizar configuraciones a los equipos para ir validando de una forma más continua el funcionamiento de la aplicación. Por lo que se recomienda que se levante un laboratorio, sobre el cual sea más sencillo realizar dichas configuraciones sin repercusiones porque algo salga mal. Nótese que para que esto funcione, sería necesario que el ambiente de desarrollo de ONAP tenga conectividad con los equipos del laboratorio, por lo que se recomienda levantar una VPN a través de la cual los servidores puedan tener dicha conectividad o que se levante una versión más simple de ONAP para el laboratorio.
- Este trabajo de graduación, gracias a la naturaleza de la plataforma ONAP, permite poder integrar la metodología de desarrollo e integración continua (CI/CD). Por esto, se recomienda utilizar los *scripts* de Bash mostrados en la Sección 7.6 para poder enriquecer y subir los CBAs cada vez que haya un cambio en el código fuente. Esta implementación puede llegar a ser un reto bastante grande, puesto que, al menos en este caso, los servidores se encuentran en una nube privada y podría ser un riesgo de seguridad darle acceso desde GitLab a los mismos, por lo que tendría que encontrarse

alguna solución alterna, como levantar *runners* de GitLab locales en la misma nube privada o algo similar.

- Como se mostró en la Sección 11.3, el algoritmo de decisión desarrollado crece rápidamente ante incrementos pequeños en el tamaño de los grupos de prefijos. Por esto, se recomienda agregar una forma de marcar aquellos grupos de prefijos que no son válidos porque saturarían las interfaces de destino para que no se utilicen en grupos de tamaños mayores. Esto mejoraría considerablemente el rendimiento del algoritmo sin comprometer que algún prefijo que pudo haber sido válido no se tome en cuenta.

CAPÍTULO 14

Bibliografía

[1] T. Guario, “Prototyping a Network Service based on the ONAP Platform,” Tesis de Maestría, Universidad Turin, 2019.

[2] F. E. L. R. Fernandes, “Implementing Network Level High-Availability and Load-Balancing on OpenStack, using SDN and NFV,” Tesis de Maestría, Técnico Lisboa, 2019.

[3] A. Networks, *E2E Network Services Automation*. dirección: https://www.aarnanetworks.com/case-studies/tigo.

[4] Kentik, *FPT Telecom uses network observability from Kentik to deliver an amazing customer experience*. dirección: https://www.kentik.com/resources/casestudy-fpt-telecom/.

[5] J. F. Kurose y K. W. Ross, *Computer networking: A Top-Down Approach*. Pearson Education Limited, 2022.

[6] Cloudflare. dirección: https://www.cloudflare.com/learning/ddos/what-is-layer-7/.

[7] Cloudflare. dirección: https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/.

[8] Imperva, *What is OSI Model | 7 Layers Explained | Imperva*. dirección: https://www.imperva.com/learn/application-security/osi-model/.

[9] W. Mair, “How does the internet work,” 1996.

[10] IETF, *Internet standards*. dirección: https://www.ietf.org/standards/.

[11] Fortinet, *What Is the Network Edge?* Dirección: https://www.fortinet.com/resources/cyberglossary/network-edge.

[12] Huawei, *What is Access Network(Basic Understanding)*. dirección: https://forum.huawei.com/enterprise/en/what-is-access-network-basic-understanding/thread/779431-100181.

[13] T. Communications, *What is a core network? How does the core network work?* Dirección: https://www.tatacommunications.com/knowledge-base/network-core-network-explained/.

[14] Gartner, *Definition of NAP (Network Access Point) - Gartner Information Technology Glossary*. dirección: https://www.gartner.com/en/information-technology/glossary/nap-network-access-point.

[15] *Network Edge Intelligence for the Emerging Next-Generation Internet*, dic. de 2010. dirección: https://www.mdpi.com/1999-5903/2/4/603.

[16] ARIN, *Autonomous System Numbers*. dirección: https://www.arin.net/resources/guide/asn/.

[17] Juniper, *BGP Overview*. dirección: https://www.juniper.net/documentation/us/en/software/junos/bgp/topics/topic-map/bgp-overview.html.

[18] Cisco, *BGP Neighbor States > BGP Fundamentals | Cisco Press*, ene. de 2018. dirección: https://www.ciscopress.com/articles/article.asp?p=2756480&seqNum=4.

[19] BGP.us, *Differences between iBGP and eBGP*. dirección: https://www.bgp.us/ibgp-and-ebgp/.

[20] Imperva, *Border Gateway Protocol | BGP Routing vs. DNS Routing*. dirección: https://www.imperva.com/learn/ddos/border-gateway-protocol-bgp/.

[21] Linode, *An Explanation of BGP Networking*. dirección: https://www.linode.com/docs/guides/an-explanation-of-bgp-networking/.

[22] D. Medhi y K. Ramasamy, "Chapter 9 - BGP," en *Network Routing (Second Edition)*, ép. The Morgan Kaufmann Series in Networking, D. Medhi y K. Ramasamy, eds., Second Edition, Boston: Morgan Kaufmann, 2018, págs. 286-333, ISBN: 978-0-12-800737-2. DOI: https://doi.org/10.1016/B978-0-12-800737-2.00011-9. dirección: https://www.sciencedirect.com/science/article/pii/B9780128007372000119.

[23] Y. Rekhter, T. Li y S. Hares, "A Border Gateway Protocol 4 (BGP-4)," RFC Editor, RFC 4271, ene. de 2006. dirección: https://www.rfc-editor.org/rfc/rfc4271.txt.

[24] *Submarine Cable Map*. dirección: https://www.submarinecablemap.com/.

[25] Amazon, *What is an API? - Application Programming Interfaces Explained - AWS*. dirección: https://aws.amazon.com/what-is/api/.

[26] Redhat, *What is a REST API?* Dirección: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[27] IBM, *¿Qué es una API REST?* Dirección: https://www.ibm.com/mx-es/topics/rest-apis.

[28] P. Sturgeon, *Understanding RPC, REST and GraphQL*, ene. de 2018. dirección: https://apisyouwonthate.com/blog/understanding-rpc-rest-and-graphql/.

[29] Google, *REST vs. RPC: what problems are you trying to solve with your APIs?* Dirección: https://cloud.google.com/blog/products/application-development/rest-vs-rpc-what-problems-are-you-trying-to-solve-with-your-apis.

[30] gRPC Authors, *Introduction to gRPC*, feb. de 2023. dirección: https://grpc.io/docs/what-is-grpc/introduction/.

[31] C. Richardson, *Microservices Patterns With Examples in Java*. Manning Publications, nov. de 2018.

[32] IBM, *What is Virtualization?* Dirección: https://www.ibm.com/topics/virtualization.

[33] VMWare, *Containers vs. Virtual Machines (VMs): What's the Difference?* Dirección: https://www.ibm.com/cloud/blog/containers-vs-vms.

[34] W. Works, *Virtual Machines vs. Containers: Virtual Solutions for Two Different Problems*, dic. de 2016. dirección: https://www.weave.works/blog/virtual-machines-vs-containers/.

[35] VMWare, *What is Container Orchestration?* Dirección: https://www.vmware.com/topics/glossary/content/container-orchestration.html.

[36] Kubernetes, *Production-Grade Container Orchestration*. dirección: https://kubernetes.io/.

[37] *ONAP: Open Network Automation Platform*, ene. de 2023. dirección: https://www.onap.org/.

[38] *Overview — onap london documentation*. dirección: https://docs.onap.org/en/latest/platform/overview/index.html.

[39] *ONAP's Architecture*, mayo de 2021. dirección: https://www.onap.org/architecture.

[40] *Architecture — onap master documentation*. dirección: https://docs.onap.org/projects/onap-aai-aai-common/en/istanbul/platform/architecture.html.

[41] *CONTROLLER DESIGN STUDIO (CDS) — onap master documentation*. dirección: https://docs.onap.org/projects/onap-ccsdk-cds/en/istanbul/index.html.

[42] A. Brogi, J. Soldani y P. Wang, "TOSCA in a Nutshell: Promises and Perspectives," en *Service-Oriented and Cloud Computing*, M. Villari, W. Zimmermann y K.-K. Lau, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, págs. 171-186, ISBN: 978-3-662-44879-3.

[43] *Controller Blueprint Archived Designer Tool (CBA) — onap master documentation*. dirección: https://docs.onap.org/projects/onap-ccsdk-cds/en/istanbul/cba/index.html.

[44] Camunda, *How to choose the right Camunda architecture*. dirección: https://camunda.com/blog/2020/04/how-to-choose-the-right-camunda-architecture/.

[45] Camunda, *Camunda Cockpit*. dirección: https://docs.camunda.org/manual/7.19/webapps/cockpit/.

CAPÍTULO 15

Anexos

15.1. Información general útil del proyecto

Servicio	***Endpoint***	Acción	Descripción
Camunda	/PROCESS_TRAFFIC_LOAD_BALANCER/start	POST	Iniciar el flujo de descubrimiento
	/PROCESS_CONFIGURE_LOAD_BALANCER/start	POST	Iniciar el flujo de configuración
CDS	/api/v1/blueprint-model/enrich	POST	Realizar proceso de enriquecimiento del CBA
	/api/v1/blueprint-model	POST	Subir y guardar el CBA enriquecido para poder utilizarse
	/api/v1/execution-service/process	POST	Ejecutar un CBA
A&AI	/aai/v23/dsl?format=simple	POST	Obtener equipos correspondientes a un país
Kentik	/api/v5/query/topxdata	POST	Obtener información de una vista configurada en Kentik
SolarWinds	/SolarWinds/InformationService/v3/Json/Query	GET	Obtener información de la base de datos de SolarWinds

Cuadro 7: APIs utilizadas en el proyecto

15.2. Utilidades para el uso de Kentik

En el Código 15.1 se muestra el *payload* que se utiliza para realizar la petición a la API REST de Kentik para obtener la información del tráfico a través de capacidades de IP Transit. Este *payload* es generado de forma automática por Kentik, como se mostró en la Sección 8.2. Dentro de la información que contiene este *payload*, puede observarse que se incluye la información que se quiere sacar en la llave “aggregateTypes”, que en este caso es el consumo de ancho de banda. Luego, en la llave “device_name” se puede observar que se incluyen todos los equipos que se agregaron en “Data Sources”, que en este caso son todos

los equipos que se encuentran en el IXP y que se conectan a los *Tier 1*. Después, en la llave "dimension" pueden verse las dimensiones que se están utilizando para agrupar el tráfico; para este caso de uso, se separa el consumo de ancho de banda por prefijo, interfaz, número de AS y el equipo en el IXP. Finalmente, en la llave "filters" se muestran todos los filtros que se están realizando; para este caso de uso se filtra para que todo el tráfico que se muestre en la vista sea aquel que está saliendo de los ASNs que pertenecen a alguna operación del ISP.

```
{
    "version": 4,
    "queries": [
        {
5           "bucket": "Left +Y Axis",
            "isOverlay": false,
            "query": {
                "all_devices": false,
                "aggregateTypes": [
10                  "avg_bits_per_sec",
                    "p95th_bits_per_sec",
                    "max_bits_per_sec"
                ],
                "aggregateThresholds": {
15                  "avg_bits_per_sec": 0,
                    "p95th_bits_per_sec": 0,
                    "max_bits_per_sec": 0
                },
                "bracketOptions": "",
20              "hideCidr": false,
                "cidr": 32,
                "cidr6": 128,
                "customAsGroups": false,
                "cutFn": {},
25              "cutFnRegex": {},
                "cutFnSelector": {},
                "depth": 75,
                "descriptor": "",
                "device_name": [
30                  "mx960_nap01",
                    "mx960_nap02",
                    "mx10k8-nap03",
                    "mx10k8-nap04"
                ],
35              "device_labels": [],
                "device_sites": [],
                "device_types": [],
                "fastData": "Auto",
                "filterDimensionsEnabled": false,
40              "filterDimensionName": "Total",
                "filterDimensionOther": false,
                "filterDimensionSort": false,
                "filterDimensions": {},
                "aggregateFiltersEnabled": false,
45              "aggregateFiltersDimensionLabel": "",
                "aggregateFilters": [],
                "hostname_lookup": true,
                "isOverlay": false,
                "isPreviousPeriod": false,
50              "lookback_seconds": 300,
                "from_to_lookback": 3600,
                "generatorDimensions": [],
                "generatorPanelMinHeight": 250,
                "generatorMode": false,
55              "generatorColumns": 1,
                "generatorQueryTitle": "{{generator_series_name}}",
                "generatorTopx": 8,
                "kmetrics": {},
                "matrixBy": [],
```

```
60                  "metric": [
                        "bytes"
                    ],
                    "minsPolling": 0,
                    "forceMinsPolling": false,
65                  "reAggInterval": "auto",
                    "reAggFn": "none",
                    "mirror": false,
                    "mirrorUnits": true,
                    "outsort": "avg_bits_per_sec",
70                  "overlay_day": -7,
                    "overlay_timestamp_adjust": false,
                    "period_over_period": false,
                    "period_over_period_lookback": 1,
                    "period_over_period_lookback_unit": "week",
75                  "query_title": "",
                    "source": "",
                    "secondaryOutsort": "",
                    "secondaryTopxSeparate": false,
                    "secondaryTopxMirrored": false,
80                  "show_overlay": false,
                    "show_total_overlay": false,
                    "starting_time": null,
                    "ending_time": null,
                    "sync_all_axes": false,
85                  "sync_axes": false,
                    "sync_extents": true,
                    "show_site_markers": false,
                    "topx": 10,
                    "update_frequency": 0,
90                  "use_kmetrics": false,
                    "use_log_axis": false,
                    "use_secondary_log_axis": false,
                    "use_alt_timestamp_field": false,
                    "viz_type": "stackedArea",
95                  "aggregates": [
                        {
                            "value": "avg_bits_per_sec",
                            "column": "f_sum_both_bytes",
                            "fn": "average",
100                         "label": "Bits/s Sampled at Ingress + Egress Average",
                            "unit": "bytes",
                            "group": "Bits/s Sampled at Ingress + Egress",
                            "origLabel": "Average",
                            "sample_rate": 1,
105                         "raw": true,
                            "name": "avg_bits_per_sec"
                        },
                        {
                            "value": "p95th_bits_per_sec",
110                         "column": "f_sum_both_bytes",
                            "fn": "percentile",
                            "label": "Bits/s Sampled at Ingress + Egress 95th Percentile"
    ,
                            "rank": 95,
                            "unit": "bytes",
115                         "group": "Bits/s Sampled at Ingress + Egress",
                            "origLabel": "95th Percentile",
                            "sample_rate": 1,
                            "name": "p95th_bits_per_sec"
                        },
120                     {
                            "value": "max_bits_per_sec",
                            "column": "f_sum_both_bytes",
                            "fn": "max",
                            "label": "Bits/s Sampled at Ingress + Egress Max",
125                         "unit": "bytes",
                            "group": "Bits/s Sampled at Ingress + Egress",
```

```
                        "origLabel": "Max",
                        "sample_rate": 1,
                        "name": "max_bits_per_sec"
                    }
                ],
                "filters": {
                    "connector": "All",
                    "filterGroups": [
                        {
                            "name": "",
                            "named": false,
                            "connector": "All",
                            "not": false,
                            "autoAdded": "",
                            "filters": [
                                {
                                    "filterField": "inet_family",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "=",
                                    "filterValue": "4"
                                },
                                {
                                    "filterField": "i_input_snmp_alias",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "ILIKE",
                                    "filterValue": "IP Transit"
                                },
                                {
                                    "filterField": "i_input_snmp_alias",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "NOT ILIKE",
                                    "filterValue": "Submarina"
                                }
                            ],
                            "saved_filters": [],
                            "filterGroups": []
                        },
                        {
                            "name": "",
                            "named": false,
                            "connector": "Any",
                            "not": false,
                            "autoAdded": "",
                            "filters": [
                                {
                                    "filterField": "dst_as",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "=",
                                    "filterValue": "52362"
                                },
                                {
                                    "filterField": "dst_as",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "=",
                                    "filterValue": "23243"
                                },
                                {
                                    "filterField": "dst_as",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "=",
                                    "filterValue": "26617"
                                },
```

```
                                {
                                    "filterField": "dst_as",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "=",
                                    "filterValue": "52262"
                                },
                                {
                                    "filterField": "dst_as",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "=",
                                    "filterValue": "262197"
                                },
                                {
                                    "filterField": "dst_as",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "=",
                                    "filterValue": "28036"
                                },
                                {
                                    "filterField": "dst_as",
                                    "metric": "",
                                    "aggregate": "",
                                    "operator": "=",
                                    "filterValue": "27742"
                                }
                            ],
                            "saved_filters": [],
                            "filterGroups": []
                        }
                    ]
                },
                "dimension": [
                    "InterfaceID_src",
                    "dst_route_prefix_len",
                    "AS_dst",
                    "i_device_id"
                ]
            }
        }
    ]
}
```

Código 15.1: *Payload* para la API REST de Kentik que obtiene la información del tráfico a través de capacidades de IP Transit

15.3. Utilidades para el uso de Camunda

15.3.1. Definición de constantes

En todo el proyecto de Camunda se utilizan varias constantes para el funcionamiento de los flujos. Tanto el Código 15.2 como en el Código 15.3 son clases de Java definidas con el propósito de guardar estas constantes, con el fin de que si el valor cambia en algún momento, este se ajuste únicamente en estas clases y no en cada instancia en el código, haciéndolo más fácil de mantener. El Código 15.2 muestra constantes que sirven para la conexión a las herramientas externas, como la A&AI, Kentik y SolarWinds. Por otro lado, el Código 15.3 contiene a las constantes que son utilizados por los archivos BPMN para indicar qué método

se ejecuta en un *Delegate Expression*, el JSON que se utilizará para el *payload*, entre otros datos.

```java
public final class DeviceConstants {

private DeviceConstants() {
      throw new UnsupportedOperationException("This is a utility class and cannot be
          instantiated");
    }

    public static final String AAI_REQ_PAYLOAD = "aaiRequestPayload";
    public static final String AAI_REQ_URL = "aaiUrlOutput";
    public static final String AAI_BASIC_REQ_URL = "aaiBasicUrl";
    public static final String AAI_CUSTOM_REQ_URL = "aaiCustomUrlOutput";
    public static final String AAI_REQ_USERNAME = "aaiUsername";
    public static final String AAI_REQ_PASSWORD = "aaiPassword";
    public static final String AAI_REQ_METHOD = "aaiReqMethod";
    public static final String AAI_ACTION_PUT = "PUT";
    public static final String AAI_ACTION_GET = "GET";
    public static final int AAI_TIME_OUT = 2700000;
    public static final String AAI_ERROR = "AAI Request Error";

    public static final String KENTIK_URL = "kentikUrl";
    public static final String KENTIK_USERNAME = "kentikUsername";
    public static final String KENTIK_PASSWORD = "kentikPassword";

    public static final String SOLARWINDS_HOST = "solarwindsHost";
    public static final String SOLARWINDS_USERNAME = "solarwindsUsername";
    public static final String SOLARWINDS_PASSWORD = "solarwindsPassword";

    public static final String AAI_CLOUD_INFRASTRUCTURE_LIST_URL =
        "cloud-infrastructure/cloud-regions/cloud-region";
    public static final String AAI_ESR_SYSTEM_INFO_LIST_URL =
        "/esr-system-info-list";
    public static final String AAI_ESR_SYSTEM_INFO_LIST_IP_FILTER = "?ip-address=";
    public static final String AAI_PNF_EQUIP_VENTOR_FILTER_QUERY = "/network/pnfs"
              +
                  "?equip-vendor=USER_INPUT_EQUIP_VENDOR&equip-type=USER_INPUT_EQUIP_TYPE";
    public static final String AAI_PNF_MAC_ADDR_FILTER_QUERY =
        "/network/pnfs/pnf/:pnf-name";
    public static final String AAI_DSL_SIMPLE_URL =
        "/dsl?format=simple&nodesOnly=true";
    public static final String AAI_ESR_SYSTEM_QUERY =
        "/cloud-infrastructure/cloud-regions/cloud-region"
                      + "/:cloud-owner/:cloud-region-id/esr-system-info-list";
    public static final String AAI_CLOUD_REGION =
        "/cloud-infrastructure/cloud-regions";
    public static final String PNF_INFO_LIST = "pnfInfoList";
    public static final String CUSTOM_URL_AAI = "aaiCustomUrl";

    public static final String NODE_TYPE = "node-type";
    public static final String ESR_SYSTEM_INFO = "esr-system-info";
}
```

Código 15.2: Clase para la definición de constantes útiles para el funcionamiento de los flujos de Camunda

```java
public final class ProcessDeviceConstants {

    private ProcessDeviceConstants() {
      throw new UnsupportedOperationException("This is a utility class and "
          + "cannot be instantiated");
    }

    public static final String VAR_PAYLOAD_KEY = "payload_key";
    public static final String VAR_RESPONSE_KEY = "response_key";
    public static final String VAR_STATUS_KEY = "status_key";
```

```
    public static final String VAR_OUTPUT_KEY = "output_key";
    public static final String VAR_PROCESS_INSTANCE_KEY = "process_instance_key";
    public static final String VAR_PROCESS_KEY = "process_key";
    public static final String VAR_PORTAL_URL = "portal_url";
    public static final String VAR_DRY_RUN = "dry_run";
    public static final String VAR_PAYLOAD = "payload";
    public static final String VAR_EXTERNAL_TASK_OUTPUT = "externalTaskOutput";
    public static final String VAR_AAI_STATUS_KEY = "aai_status_key";
    public static final String VAR_EXTERNAL_TASK_AAI_OUTPUT =
        "externalTaskAAIOutput";
    public static final String CBA_ERROR = "CBAError";
    public static final String CBA_STATUS_OUT = "cbaStatusOut";
    public static final String IPV4_SYSTEM = "ipv4System";
    public static final String IPV4_SYSTEMS = "ipv4Systems";
    public static final int ERROR_MAX_SIZE = 2000;
}
```

Código 15.3: Clase para la definición de constantes útiles para el funcionamiento de los flujos de Camunda

15.3.2. Métodos para la construcción de las peticiones a la base de datos

En la Subsección 9.1.3 se habló del patrón *Construct - Put - Parse*. En el Código 15.4 se puede ver el método utilizado para construir la consulta a la A&AI para obtener todos los equipos que pertenecen a un país para poder realizar las configuraciones. Por otro lado, en el Código 15.5 puede verse el método utilizado para construir la consulta a la A&AI para obtener las credenciales de un equipo en específico para poder entrar al equipo por medio de Netconf.

```
public static void constructInstallationQuery(
    String country,
    String basicUrl,
    String query,
    DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "constructInstallationQuery method");
    String uriQuery = basicUrl + DeviceConstants.AAI_DSL_SIMPLE_URL;
    String countryQuery = query;
    countryQuery = countryQuery.replace("INPUT_COUNTRY", country);
    delegateExecution
        .setVariable(DeviceConstants.AAI_REQ_PAYLOAD, countryQuery);
    delegateExecution
        .setVariable(DeviceConstants.AAI_REQ_URL, uriQuery);
    delegateExecution
        .setVariable(DeviceConstants.AAI_REQ_METHOD, DeviceConstants.AAI_ACTION_PUT);
}
```

Código 15.4: Fase de construcción de la petición en el patrón *Construct - Put - Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos

```
public static void constructEsrQuery(
    String ipv4,
    String basicUrl,
    DelegateExecution delegateExecution) {
    logger.log(Level.INFO, "constructEsrQuery method");
    String uriQuery = basicUrl + DeviceConstants.AAI_DSL_SIMPLE_URL;
    try {
        logger.log(Level.INFO, "ipv4 : {0}", ipv4);
        String esrQuery = DslConstants.AAI_DSL_ESR_SYSTEM_INFO_REGION;
        esrQuery = esrQuery.replace("USER_INPUT_ESR_SYSTEM_INFO", ipv4);
        logger.log(Level.INFO, "Query {0}", esrQuery);
        delegateExecution
```

```
            .setVariable(DeviceConstants.AAI_REQ_PAYLOAD, esrQuery);
        delegateExecution
            .setVariable(DeviceConstants.AAI_REQ_URL, uriQuery);
        delegateExecution
            .setVariable(DeviceConstants.AAI_REQ_METHOD,
                DeviceConstants.AAI_ACTION_PUT);
    } catch (Exception jsonExp) {
        throw new BpmnError(ProcessDeviceConstants.CBA_ERROR,
            "constructEsrQuery Exception Occurred !!");
    }
}
```

Código 15.5: Fase de construcción de la petición en el patrón *Construct - Put - Parse* para la obtención de las credenciales del equipo que se va a configurar

15.3.3. Métodos para el *parseo* de información

Luego de la construcción y ejecución de las consultas a la base de datos de ONAP, se requiere, siguiendo el patrón *Construct - Put - Parse*, la etapa de interpretación y manipulación (o *parseo*) de los datos. En el código 15.6 puede observarse el método utilizado para obtener los equipos encontrados para cada país involucrado. Después, en el Código 15.7 puede verse el método utilizado para sacar la información necesaria para realizar las configuraciones del equipo.

```
public static void parsePnfByCountry(String esrJsonResponse, DelegateExecution
    delegateExecution) {
    logger.log(Level.INFO, "parsePnfByCountry method");
    EsrSystemInfo[] esrSystemInfo = null;
    try {
        ObjectMapper mapper = new ObjectMapper();
        JsonNode tempNode = mapper.readTree(esrJsonResponse);
        JsonNode resultStr = tempNode.path("results");
        JsonNode tempStrNode;
        JsonNode tempPropNode;
        EsrSystemInfo esrInfo = null;
        esrSystemInfo = new EsrSystemInfo[resultStr.size()];
        String tempEsrUrl = null;
        String[] tempEsrArray;

        for (int i = 0; i < resultStr.size(); i++) {
            tempStrNode = resultStr.get(i);
            if (tempStrNode.get(DeviceConstants.NODE_TYPE) != null
                && tempStrNode.get(DeviceConstants.NODE_TYPE).textValue()
                    .equalsIgnoreCase(DeviceConstants.ESR_SYSTEM_INFO)) {
                tempEsrUrl = tempStrNode.get("url").textValue();
                tempEsrArray = tempEsrUrl.split("/");
                tempPropNode = tempStrNode.get("properties");
                esrInfo = mapper.readValue(tempPropNode.toPrettyString(),
                    EsrSystemInfo.class);
                esrInfo
                    .setEsrRegionId(tempEsrArray[7]);
                esrInfo
                    .setEsrCloudOwner(tempEsrArray[6]);
                esrInfo
                    .setDeviceVendor(deviceVendorStr(esrInfo.vendor.toLowerCase()));

                if (esrInfo.password.equalsIgnoreCase("N/A")) {
                esrInfo.password = "";
                }
                esrSystemInfo[i] = esrInfo;
            }
        }
```

```
            delegateExecution
                .setVariable(DeviceConstants.PNF_INFO_LIST, esrSystemInfo);
            delegateExecution
                .setVariable(ProcessDeviceConstants.CBA_STATUS_OUT, true);
        } catch (JsonProcessingException jsonExp) {
            throw new BpmnError(ProcessDeviceConstants.CBA_ERROR,
                "parsePnfByCountry JsonProcessingException Occurred !!");
        } catch (Exception exp) {
            throw new BpmnError(ProcessDeviceConstants.CBA_ERROR,
                "Exception in parsePnfByCountry !!");
        }
    }
```

Código 15.6: Fase de *parseo* de la petición en el patrón *Construct* - *Put* - *Parse* para la obtención de equipos locales de cada operación involucrados en los movimientos

```
public static void parseExternalEsrResponse(String queryResponse, DelegateExecution
    delegateExecution) {
    logger.log(Level.INFO, "parseExternalEsrResponse method");
    try {
        ObjectMapper mapper = new ObjectMapper();
        JsonNode tempNode = mapper.readTree(queryResponse);
        JsonNode resultStr = tempNode.path("results");
        EsrSystemInfo tempEsrInfo = new EsrSystemInfo();

        for (JsonNode tempStrNode : resultStr) {
        String nodeType = tempStrNode.path("node-type").textValue();
            if (nodeType.equalsIgnoreCase("esr-system-info")) {
                handleEsrSystemInfo(tempStrNode, mapper, delegateExecution,
                    tempEsrInfo);
                delegateExecution
                    .setVariable("esrStatus", "success");
            } else if (nodeType.equalsIgnoreCase("cloud-region")) {
                handleCloudRegion(tempStrNode, delegateExecution, tempEsrInfo);
            }
        }
        delegateExecution
            .setVariable("esrDeviceInfo", tempEsrInfo);

    } catch (IOException jsonExp) {
        handleException("Json Exception :: " + jsonExp.getMessage());
    } catch (Exception exp) {
        handleException("Exception :: " + exp.getMessage());
    }
}
```

Código 15.7: Fase de *parseo* de la petición en el patrón *Construct* - *Put* - *Parse* para la obtención de las credenciales del equipo que se va a configurar

15.4. Utilidades para el uso de SolarWinds desde un CBA

Como se mencionó en la Subsección 8.3.3, se utiliza la API de SolarWinds para obtener la información de las interfaces de los equipos que se encuentran en el IXP y que conectan con algún *Tier 1*. En el Código 15.8 se ven las dos consultas utilizadas para esto; la primer consulta es utilizada para sacar la información de todas las interfaces de *IP Transit* de todos los equipos del IXP y la segunda consulta es utilizada para sacar la información de una interfaz en específico.

```
class SWQLConstants():
    INTERFACE_INPUT_AVAILABLE_BANDWIDTH = """
```

```
            SELECT I.Caption, I.Inbps, I.InPercentUtil, I.Speed, N.IPAddress
            FROM Orion.NPM.Interfaces I
            JOIN Orion.Nodes N ON I.NodeID = N.NodeID
            WHERE
                I.Caption LIKE '%IP Transit%' AND
                I.Caption NOT LIKE '%Submarina%'
        """
    SPECIFIC_INTERFACE_INPUT_AVAILABLE_BANDWIDTH = """
            SELECT I.Caption, I.Inbps, I.InPercentUtil, I.Speed, N.IPAddress
            FROM Orion.NPM.Interfaces I
            JOIN Orion.Nodes N ON I.NodeID = N.NodeID
            WHERE
                I.Name LIKE '%{}%' AND
                N.IPAddress = '{}'
        """
```

Código 15.8: Clase que contiene las peticiones a la base de datos de SolarWinds

15.5. Detalle del algoritmo para determinar los posibles movimientos para el balanceo de tráfico

En los códigos 15.9 y 15.10 se puede ver el detalle de cómo funciona el algoritmo mencionado en la Subsección 8.3.3. En el caso del Código 15.9 se muestra cómo se utiliza la función *combinations* de la librería *itertools* de Python para obtener todas las combinaciones posibles de prefijos que se pueden mover. Luego, puede apreciarse cómo se utiliza la función *product* de la misma librería para obtener todas las combinaciones posibles de interfaces a las que se pueden mover los prefijos, que luego es agrupada por prefijo. Esto resulta en una lista de de listas, en donde cada lista interna consta de una tupla que tiene el prefijo y la interfaz a la que se puede mover. Finalmente, con el segundo uso de la función *product* se obtienen todas las combinaciones posibles de movimientos que se pueden realizar de todos los prefijos a todas interfaces. Además de esto, en esta parte del código se hacen los filtros de verificar que el consumo de ancho de banda el grupo de movimientos a realizar sí liberaría la interfaz saturada, que el país al que pertenece el prefijo sí tiene la capacidad de mover el prefijo a la interfaz de destino y si se alcanzó ya el máximo de movimientos para devolverlos y terminar el algoritmo.

Finalmente, en el Código 15.10 se muestra cómo se verifica que los movimientos que se están realizando no sobrepasen el umbral de tráfico de la interfaz a la que se está moviendo el prefijo y que esta condición se cumpla para todos los prefijos involucrados en el grupo de movimientos.

```
def get_possible_moves(self, prefixes_from_sat, from_bps, from_threshold,
    ip_transit_interfaces, possible_moves):
    for k in range(1, len(prefixes_from_sat)+1):
        for _combination in combinations(prefixes_from_sat, k):
            valid_moves = []

            combined_bps = sum(prefix['prefix-bps'] for prefix in _combination)
            new_from_bps = from_bps - combined_bps
            if new_from_bps > from_threshold:
                continue

            move_options = product(list(_combination), ip_transit_interfaces)
            move_options_list = [move_option for move_option in move_options]

```

```
            for idx, move_option in enumerate(move_options_list):
                if move_option[1]['tier1'] not in self.destinations_by_country[
                    self.asn_conversion[move_option[0]['asn']]
                ]:
18                  move_options_list.pop(idx)

            grouped_lists = [list(group) for _, group in groupby(move_options_list,
                key=lambda x: x[0]['prefix'])]
            move_combinations = product(*grouped_lists)
            valid_moves = self.check_move_combination(move_combinations,
                valid_moves, new_from_bps)
23          if valid_moves:
                possible_moves['move-groups'].append(valid_moves)
            for move_group in possible_moves['move-groups']:
                total_moves += len(move_group)
                if total_moves >= self.maximum_moves:
28                  return possible_moves
    return possible_moves
```

Código 15.9: Método para la creación de las posibles combinaciones entre interfaces y prefijos para realizar el balanceo

```
1   def check_move_combination(self, move_combinations, valid_moves, new_from_bps):
        asn_conversion = {
            ...
        }
        for move_combination in move_combinations:
6           validity = 0
            repeated_destination = {}

            for move in move_combination:
                prefix_data = move[0]
11              to_interface = move[1]

                to_interface_key = f"{to_interface['pnf_ipv4']} -
                    {to_interface['interface_name']}"
                to_interface_accu_bps = repeated_destination.get(to_interface_key,
                    to_interface['bps'])

16              suppossed_traffic = prefix_data['prefix-bps'] + to_interface_accu_bps
                if suppossed_traffic >= to_interface['threshold']:
                    break
                validity += 1
                repeated_destination[to_interface_key] = suppossed_traffic
21
            if validity == len(move_combination):
                valid_moves.append(
                    [
                        {
26                          'from-new-bps': new_from_bps,
                            'prefix': move[0]['prefix'],
                            'prefix-bps': move[0]['prefix-bps'],
                            'asn': asn_conversion[move[0]['asn']],
                            'to-pnf': move[1]['pnf_ipv4'],
31                          'to-interface': move[1]['interface_name'],
                            'to-tier1': move[1]['tier1'],
                            'to-bps': repeated_destination[f"{move[1]['pnf_ipv4']} -
                                {move[1]['interface_name']}"]
                        } for move in move_combination
                    ]
36              )
            if len(valid_moves) >= self.maximum_moves:
                return valid_moves
        return valid_moves
```

Código 15.10: Método para la obtención de los posibles movimientos que se pueden realizar para balancear la carga

15.6. Detalle del algoritmo para generar las plantillas de configuración de los equipos

En el Código 15.11 se observa la función utilizada para obtener el país al que pertenece el equipo en el que se está realizando la configuración con el fin de intentar armar las plantillas de configuración para el movimiento de solo los prefijos que aplican al país, ya que en el grupo de movimientos pueden haber múltiples movimientos que corresponden a diferentes países.

```python
def get_country(self, ip_address):
    self.logger.info("get_country called")
    dsl = DSLConstants.GET_COUNTRY_BY_ADDRESS
    url = URLConstants.SIMPLE_DSL_URL
    context = {"IP_ADDRESS": ip_address}
    dsl_url, dsl_body = self.queryHandler.craft(
        dsl,
        url,
        self.aai_url,
        context)

    response = asyncio.run(
        self.queryHandler.dsl_execute(
            dsl_url,
            dsl_body,
            self.aai_resources_access_info)).json()

    try:
        country = response["results"][0]["properties"]["country"]
    except AttributeError:
        country = ""

    return country
```

Código 15.11: Método para la obtención del país al que pertenece un PNF en específico

En el Código 15.12 puede verse la clase de Python que tiene el algortímo para obtener la información de los equipos a través de Netconf y que utiliza esa información para interpretarla y modificarla para poder luego formar las plantillas de configuración. Como puede verse, esta clase cuenta con cuatro funciones principales: el descubrimiento general de la información de configuración (la cual será explicada a más detalle luego), la obtención de los *routers* lógicos, la obtención de las políticas y la obtención de las comunidades BGP. La obtención de los *routers* lógicos es utilizada porque estos equipos separan sus funciones en *routers* lógicos, donde uno es utilizado para gestionar el tráfico relacionado a internet y otro es utilizado para gestionar el tráfico de enlaces de datos (tráfico privado). Por otra parte, la obtención de las políticas sirva para poder determinar en qué política se está diciendo que el prefijo deseado salga por un *Tier 1* específico indicado por una comunidad. Finalmente, la obtención de las comunidades sirve para saber qué comunidades tiene el equipo, para asegurarse que el prefijo se está moviendo a una comunidad existente.

```python
class JuniperRouterDiscoveryService:

    def __init__(self, aai_params, juniper_router, pnf_data, async_details):
        ...

    def logger_for_decorator(self):
        return self.logger

    def discover_config_info(self, prefixes, from_tier1):
```

```
    ...

    @timer(logger_for_decorator)
    @error_handler(logger_for_decorator)
    def get_logical_pnfs(self):
        logical_pnfs = self.juniper_device.discover_logical_systems()
        return logical_pnfs

    @timer(logger_for_decorator)
    @error_handler(logger_for_decorator)
    def get_policy_statement(self, logical_system, prefix, current_config = True,
        to_tier1 = None, policy = None):
        policy_attributes = self.juniper_device.discover_policies(logical_system,
            prefix, current_config, to_tier1, policy)
        return policy_attributes

    @timer(logger_for_decorator)
    @error_handler(logger_for_decorator)
    def get_communities(self, logical_system, to_tier1):
        communities = self.juniper_device.discover_communities(logical_system,
            to_tier1)
        return communities
```

Código 15.12: Clase para el descubrimiento de información de los equipos de la red

En el Código 15.13 se puede observar la función que obtiene la información de configuración general de los equipos antes mencionada. Como puede verse, inicialmente se obtienen los *routers* lógicos, los cuales se filtran para quedarse únicamente con el responsable del tráfico de internet. Luego, se utiliza el estándar de los nombres de los *Tier 1* en las interfaces y la expresión regular estándar de los nombres de las comunidades para poder determinar qué comunidad debería de tener el prefijo originalmente que se quiere mover según la información obtenida de Kentik; nótese que si esta política no se encuentra se indica que hay un error y se termina el algoritmo. Después, se sigue el paso anterior para determinar la comunidad que debería de existir en alguna política para que el movimiento sea posible; nótese que en este caso, si no existe ninguna política con esta comunidad no es un problema, siempre y cuando la comunidad exista, porque parte del flujo es tomar en cuenta que si esto pasa, debería de configurarse una política temporal nueva. Finalmente, el sistema se encarga de ir acumulando esta información en un arreglo, el cual es el que luego se pasará como contexto a la función de Jinja para que genere la configuración deseada.

```
def discover_config_info(self, prefixes, from_tier1):
    from_tier1_name = from_tier1.split("#")[0].lower()
    from_tier1_nap = from_tier1.split("#")[1]
    from_tier1_pattern =
        re.compile(r'{0}.*-{1}--\d+-prepend'.format(from_tier1_name, from_tier1_nap))

    final_response = {
        "logical-system": "",
        "moves": []
    }
    connection_established = False
    try:
        err_response = CbaCommonKeywords.CONNECTION_FAILURE
        connection = self.juniper_device.open_connection()

        if not connection:
            result = False
            connection_established = False
            return result, err_response, connection_established

        result = True
```

```
        err_response = None
        connection_established = True

        result, logical_system_response = self.get_logical_pnfs()
        logical_systems = [element for element in logical_system_response if 'INT'
            in element.upper()]

        if not logical_systems:
            return

        logical_system = logical_systems[0]
        final_response["logical-system"] = logical_system

        for prefix in prefixes:
            to_tier1_name = prefix[1].split("#")[0].lower().replace(" ", "")
            to_tier1_nap = prefix[1].split("#")[1]
            to_tier1_pattern =
                re.compile(r'{0}-{1}--\d+-prepend'.format(to_tier1_name,
                to_tier1_nap))

            result, policies_response = self.get_policy_statement(logical_system,
                prefix[0])

            communities = [
                community['community-name'] for community in
                    policies_response[0]['term'][0]['then']['community']
            ]

            from_tier1_community = [
                community for community in communities if
                    from_tier1_pattern.findall(community)
            ][0]
            if not from_tier1_community:
                err_response = list(CbaCommonKeywords.ERROR_CODES)[2]
                result = False
                return result, err_response, connection_established

            result, policies_response_to = self.get_policy_statement(
                logical_system,
                prefix[0],
                current_config = False,
                to_tier1 = prefix[1],
                policy = policies_response[0]['name'])

            communities_to = None if not policies_response_to else \
            [
                community['community-name'] \
                for community in
                    policies_response_to[0]['term'][0]['then']['community']
            ]

            to_tier1_community = self.get_communities(logical_system, prefix[1])[1] \
                if not communities_to \
                else [community \
                    for community in communities_to \
                    if to_tier1_pattern.findall(community)][0]

            communities = [
                to_tier1_community if community == from_tier1_community \
                else community for community in communities
            ]
            new_term_name = to_tier1_name + "_ONAP_temp"
            move = {
                "policy-statement": policies_response[0]['name'],
                "term": new_term_name if not policies_response_to else
                    policies_response_to[0]['term'][0]['name'],
                "prefix": prefix[0],
                "communities": communities
```

```
            }
            final_response['moves'].append(move)

        return result, final_response, connection_established

    except Exception as err:
        ...

    finally:
        ...
```

Código 15.13: Método para la obtención, interpretación y uso de la información de los equipos de red para generar las plantillas de configuración

En el Código 15.14 se muestran las funciones utilizadas para poder realizar consultas de tipo DSL a la A&AI y las funciones de Jinja para poder generar las plantillas de configuración dado un diccionario de contexto.

```
# Imports
...

class QueryHandler():
    def __init__(self):
        # Init method
        ...

    def craft(self, dsl, url, aai_url, context):
        self.logger.info("craft called")
        dsl_body = dsl
        for key, value in context.items():
            dsl_body = dsl_body.replace(key, value)
        dsl_url = url.replace("AAI_URL", aai_url)
        return (dsl_url, dsl_body)

    async def dsl_execute(self, dsl_url, dsl_body, aai_resources_access_info,
        session=None, semaphore=None):
        self.logger.info("execute called")
        aai_object = AAIServiceUtility(aai_resources_access_info)
        if session and semaphore:
            async with semaphore:
                return await aai_object.put_async(dsl_url, dsl_body, session)
        else:
            return aai_object.put(dsl_url, dsl_body)

    def parse_plain_text(self, template, data=None, **input_data):
        self.logger.info("parse called")
        os.chdir(os.path.dirname(os.path.dirname(os.path.dirname(__file__))))
        file_loader = FileSystemLoader('Templates')
        env = Environment(loader=file_loader)
        template = env.get_template(template)
        render_data = None
        if data:
            render_data = template.render(data)
        if input_data:
            render_data = template.render(input_data)
        return render_data
```

Código 15.14: Clase con métodos útiles para la ejecución de DSL *queries* y *parseo* de la información

15.7. Utilidades diversas para los *scripts* de Python que son parte del CBA

Para un funcionamiento más escalable y fácil de mantener, se crearon dos decoradores para las funciones de Python, los cuales pueden verse en los códigos 15.15 y 15.16, los cuales se encargan de realizar programación defensiva general y tomar tiempos para evaluación del rendimiento de la aplicación respectivamente.

```python
def error_handler(logger_func):
    """
    Handle errors in the decorated function
    """
    def erro_decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            try:
                func_response = func(*args, **kwargs)
                return True, func_response
            except Exception as err:
                logger = logger_func(args[0])
                logger.error(f"Error processing function {func.__name__}: {err}")
                return False, None
        return wrapper
    return erro_decorator
```

Código 15.15: Decorador para el manejo de errores

```python
def timer(logger_func):
    """
    Print the runtime of the decorated function
    """
    def timer_decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            start_time = time.perf_counter()
            value = func(*args, **kwargs)
            end_time = time.perf_counter()
            run_time = end_time - start_time
            logger = logger_func(args[0])
            logger.info(f"Finished {func.__name__!r} in {run_time:.4f} secs")
            return value
        return wrapper
    return timer_decorator
```

Código 15.16: Decorador para medir el tiempo de ejecución de las funciones