



TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Optimización del despliegue de SDNs por medio de Algoritmos Bioinspirados

Autor

Luis María Chaves López

Directores

Antonio Miguel Mora García
Juan Francisco Valenzuela Valdés



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

—
Granada, 16 de Junio de 2025

Optimización del despliegue de SDNs: Mediante Algoritmos Bioinspirados.

Luis María Chaves López

Palabras clave: Redes Definidas por Software (SDN), balanceo de carga, OCH, SH, optimización bioinspirada, visualización de topología

Resumen

Este Trabajo Fin de Grado presenta el desarrollo y evaluación de un algoritmo bioinspirado basado en Ant Colony Optimization (*ACO*), con enfoque en su variedad Ant Colony System (*ACS*). El principal objetivo es optimizar el balanceo de carga en Redes Definidas por Software (SDN). El objetivo de este trabajo es mejorar la eficiencia de la red considerando métricas clave como la carga, delay y pérdida de paquetes, adaptándose dinámicamente a las condiciones de la red en tiempo real. Para la consecución del objetivo, se ha implementado una plataforma experimental que combina Mininet, controlador Ryu y una interfaz web, que permite la visualización interactiva de la red. Además, se comparan los resultados del algoritmo propuesto con métodos clásicos como Dijkstra, evaluando métricas de rendimiento y calidad de servicio. Los resultados demuestran que el enfoque propuesto ofrece mejoras significativas en el equilibrio de carga y estabilidad de red.

Optimization of SDNs deployment: By means of Bio-inspired Algorithms.

Luis María Chaves López

Keywords: Software-Defined Networking (SDN), load balancing, Ant Colony Optimization (ACO), Ant Colony System (ACS), bio-inspired optimization, topology visualization

Abstract

This Final Degree Project presents the development and evaluation of a bio-inspired algorithm based on Ant Colony Optimization (ACO), focusing on its ACS variety. The main objective is to optimize load balancing in Software Defined Networks (SDN). The goal is to improve network efficiency by considering key metrics such as load, delay and packet loss, dynamically adapting to network conditions in real time. To achieve this goal, an experimental platform has been implemented combining Mininet, Ryu controller and a web interface, which allows interactive visualization of the network. In addition, the results of the proposed algorithm are compared with classical methods such as Dijkstra, evaluating performance and quality of service metrics. The results show that the proposed approach offers significant improvements in load balancing and network stability.

Yo, **Luis María Chaves López**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 77171058D, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Luis María Chaves López

Granada, 16 de junio de 2025 .

D. **Antonio Miguel Mora García** , Profesor del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

D. **Juan Francisco Valenzuela Valdés** , Profesor del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado *Optimización del despliegue de SDNs por medio de Algoritmos Bioinspirados*, ha sido realizado bajo su supervisión por **Luis María Chaves López**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 15 de Junio de 2025.

Los directores:

Antonio Miguel Mora García

Juan Francisco Valenzuela Valdés

Agradecimientos

Me gustaría agradecer a mis familiares y amigos por todo el apoyo recibido durante la realización de este proyecto. En especial a mis padres y a mi hermana. Además, quiero agradecer y hacer una mención especial a mi tutor Antonio Mora, sin él este proyecto no habría sido posible, gracias por ayudarme siempre que lo he necesitado.

Índice general

1. Introducción	1
1.1. Motivación	2
1.1.1. Limitaciones de los métodos de balanceo de carga convencionales en entornos SDN	3
1.1.2. Algoritmos Bioinspirados	4
1.1.3. ¿Por qué los algoritmos bioinspirados son tan interesantes respecto a otros métodos de optimización?	4
1.2. Objetivos	6
1.2.1. Objetivos secundarios	6
1.3. Estructura	7
2. CONCEPTOS PRELIMINARES	9
2.1. SDN	9
2.2. Algoritmos ACO	10
2.2.1. Funcionamiento general de ACO	11
2.2.2. Explicando ACO a través de un ejemplo	11
2.3. Openflow	12
2.3.1. ¿Cómo funciona OpenFlow? [1]	13
3. DEFINICIÓN DEL PROBLEMA	14
4. Estado del arte	16
4.1. The Load Balancing of SDN based on ACO with Job Classification	16
4.1.1. Algoritmo Planteado	17
4.1.2. Comparación con el TFG	17
4.2. Joint route-server load balancing in software defined networks using ACO	17
4.2.1. Comparación con el TFG	18
4.3. Dynamic Load Balancing of Software-Defined Networking Based on Genetic-Ant Colony Optimization	19
4.3.1. Algoritmo planteado	19
4.3.2. Comparación con el TFG	19

4.4. Conclusiones	20
5. Planificación	22
5.1. Fases de proyecto	22
5.2. Presupuesto estimado	24
6. HERRAMIENTAS Y DATOS	26
6.1. Mininet	26
6.1.1. ¿En qué consiste una red de Mininet?	26
6.2. Controlador SDN Ryu	27
6.3. Flask	28
6.3.1. Comunicación entre Flask y el controlador RYU	29
6.3.2. Visualización de la red con D3	30
7. Utilidad de Visualización de la Red	31
7.1. Tecnologías utilizadas	31
7.2. Acciones Disponibles para el Usuario	32
7.2.1. Visualización de la topología	34
7.3. Potencial de Uso y Evaluación	36
8. Algoritmos Desarrollados	37
8.1. Selección del siguiente nodo	37
8.2. Estrategia de actualización de feromonas	38
8.3. Cálculo de la carga y coste del camino	39
8.4. Encontrar el mejor camino	39
8.4.1. Cálculo del umbral y actualización de la lista de caminos	40
8.4.2. Encontrar el mejor camino	40
8.5. Descripción general del algoritmo	40
8.6. Enfoque ACS	41
8.6.1. Exploración frente a Explotación ($q0$)	41
8.6.2. Actualización Local de Feromonas (ϕ)	41
8.6.3. Ventajas del enfoque ACS	42
9. Experimentos	43
9.1. Configuración	43
9.1.1. Parámetros	43
9.1.2. Configuraciones	45
9.1.3. Resultados de la comparativa de configuraciones	49
9.2. Comparación de algoritmos	51
10. Conclusiones	55
Bibliografía	61

A. Manual de usuario	62
A.1. Introducción	62
A.2. Requisitos y herramientas necesarias	62
A.3. Guía de uso de la aplicación	62
A.4. Modificar parámetros del algoritmo	64
Glosario	65

Índice de figuras

2.1. Openflow en SDN. Fuente: [1]	12
2.2. Arquitectura de Openflow. Fuente: [1]	13
5.1. Diagrama de Gantt con la cronología de la realización de las diferentes tareas	24
6.1. Estructura de una red Mininet. Fuente [2]	27
6.2. Arquitectura de Ryu. Fuente [3]	28
6.3. Flujo de comunicación entre el navegador, Flask y el contro- lador Ryu (elaboración propia)	29
6.4. Visualización inicial de la topología de red con D3.js	30
7.1. Visión general de la web	32
7.2. Botones de acción sobre la instantánea	33
7.3. Cambiar nodo origen y nodo destino	33
7.4. Modificación de delay del enlace a 10 ms	33
7.5. Generación de carga de host X a host Y	34
7.6. Topología de red	35
7.7. Tabla de métricas al clicar en el enlace	35
7.8. Topología y tabla	36
8.1. Selección de nodo. Fuente [4]	38
9.1. Visualización de la topología para las pruebas	49
9.2. Costes promedio de cada configuración	50
9.3. Desviación típica en los costes de cada configuración	50
9.4. Tiempo promedio de cada configuración	51
9.5. Coste promedio de cada algoritmo	52
9.6. Coste de cada ejecución	52
9.7. Tiempo promedio de cada algoritmo	53
9.8. Carga promedio	53
9.9. Delay promedio	54
A.1. Lanzamiento de la aplicación	63
A.2. Visión de la aplicación	63

A.3. Inicio del controlador	64
A.4. Ejecución de la topología	64

Índice de cuadros

5.1. Estimación de horas y coste del proyecto (estudiante: 15€/h, tutor: 40€/h)	25
9.1. Parámetros utilizados para la ejecución del algoritmo ACO . .	46
9.2. Resumen de configuraciones modificadas para evaluación . . .	46
9.3. Comparación de Coste, Carga y Retardo Total por Algoritmo	51

Glosario de Acrónimos

Application to Controller Plane Interface A-CPI. 10

Ant Colony Optimization (Optimización por Colonias de Hormigas)
ACO. 2

Ant Colony System (Sistema de Colonias de Hormigas) ACS. 6

Application Programming Interface API. 12

D3.js Data-Driven Documents JavaScript. 30

GA Genetic Algorithm (Algoritmo Genético). 19

Hypertext Transfer Protocol HTTP. 28

OCH Optimización basada en Colonias de Hormigas. 2

QoE Quality of Experience. 15

Quality of Service *QoS*. 5

RR Round Robin. 4

SDN Software Defined Network. 1

SH Sistema de Hormigas. 2

Trabajo Fin de Grado TFG. 7

Transmission Control Protocol TCP. 13

Transport Layer Security TLS. 10

TSP Travelling Salesman Problem (Problema del viajante de comercio). 11

Web Server Gateway Interface WSGI. 29

Capítulo 1

Introducción

El crecimiento de las redes ha dificultado su gestión [5], por lo que recientemente se ha propuesto el concepto de Software Defined Network (SDN) para abordar este problema, al separar el plano de control del plano de datos, permite una **gestión centralizada** de la red.

Dado que los recursos de la red son limitados, optimizar su utilización resulta crucial. En este contexto, el balanceo de carga permite mejorar la distribución equilibrada de la carga entre múltiples recursos, con el objetivo de maximizar la fiabilidad y la eficiencia de los recursos de la red [6] [7]. Los controladores SDN pueden crear un balanceo de carga óptimo en comparación con las redes tradicionales, gracias a su visión global de la red.

El problema del balanceo de carga puede ser solucionado usando diferentes técnicas metaheurísticas bioinspiradas, debido a su naturaleza **NP-complete** [8]. Por lo tanto, para resolver el problema del balanceo de carga en SDN, las técnicas metaheurísticas bioinspiradas son métodos cruciales.

El principal propósito de las redes SDN es hacer que el manejo de las redes sea más flexible y sencillo mediante la separación de la infraestructura de la red en dos capas lógicas: capa de control y de datos [9] [10]. De esta manera, el controlador de la red se mueve desde dispositivos de transferencia para ser gestionado por un controlador lógico centralizado con el objetivo de que el software pueda implementar funciones de red.

El problema del balanceo de carga

Uno de los desafíos clave en las redes SDN es el balanceo de carga. Consiste en el proceso de **distribuir el tráfico de la red entre múltiples rutas o servidores** con el objetivo de evitar la congestión, **maximizar la utilización de los recursos disponibles** y una mejorar el rendimiento general[11]. escalabilidad y robustez de la red, asegurando que ningún recurso de la red se sobrecargue mientras haya otros inactivos o poco cargados.

Para resolver el problema del balanceo de carga hay dos principales ca-

tegorías de algoritmos [5]:

- **Los algoritmos estáticos:** necesitan saber la información previa del sistema. La principal ventaja de los algoritmos estáticos es su simple implementación, pero la viabilidad de un equilibrio inadecuado es alta [12] [13]. Por otro lado, la mayor desventaja de los algoritmos estáticos, es que el estado actual del sistema no se tiene en cuenta a la hora de tomar la decisión, por lo que no es un enfoque adecuado en sistemas en el que el estado de la carga no puede predecirse con antelación [5].
- **Los algoritmos dinámicos:** se basan en el estado actual del sistema y se emplean principalmente para gestionar las cargas de procesamiento cambiantes. Las tareas pueden transferirse de un nodo sobrecargado a otro con poca carga mediante los algoritmos dinámicos[5].

El problema del balanceo de carga se vuelve más complejo en redes dinámicas, donde los flujos cambian frecuentemente y los enlaces pueden sufrir una degradación temporal o fallos de congestión temporal [14]. Aquí es donde las técnicas clásicas (Dijkstra, RR) se quedan cortas ya que son algoritmos estáticos que no tienen en cuenta el estado en tiempo real de la red. Por otro lado, es la principal ventaja de los algoritmos dinámicos ya que dependen del estado actual del sistema. Los mecanismos dinámicos nos permiten lograr un mejor rendimiento y obtener soluciones más eficientes y precisas [5].

Algoritmos bioinspirados

Como comentamos recientemente, el problema del balanceo de carga en SDN es un problema **NP-complete** [15], es decir, que no se puede resolver de forma exacta en un tiempo razonable para redes de gran tamaño. Para ello se han propuesto técnicas metaheurísticas bioinspiradas, que permiten encontrar buenas soluciones en un tiempo razonable.

Una de las más destacables y prometedoras es ACO (Ant Colony Optimization (Optimización por Colonias de Hormigas)). Esta técnica se basa en el comportamiento real de las colonias de hormigas para encontrar rutas óptimas hacia fuentes de alimento, utilizando feromonas como mecanismo de retroalimentación. Esto aplicado al contexto de las redes, ACO permite encontrar rutas eficientes considerando métricas como pérdida de paquetes, carga y retardo.

1.1. Motivación

Las redes modernas se enfrentan a una presión creciente para ofrecer **flexibilidad, escalabilidad y una gestión dinámica** [8]. Con el crecimiento

exponencial del tráfico de datos y los cambiantes requisitos de la red. Las arquitecturas tradicionales basadas en hardware, a menudo se ven superadas, debido a la **complejidad de su configuración manual**, la propensión a errores y la falta de adaptabilidad a las demandas cambiantes del tráfico.

Las Redes Definidas por Software (SDN), surgen como respuesta a este problema, posibilitando aplicar técnicas más avanzadas como la **monitorización continua** del estado de la red, **la reconfiguración dinámica** de rutas y servicios en tiempo real.

No obstante, el verdadero reto no reside en **recopilar información del el plano de datos**, sino en **tomar decisiones óptimas**, como enrutar el tráfico de manera de que minimice la congestión, el equilibrio de las cargas en los enlaces, la reducción de latencias y la pérdida de paquetes. [8]

1.1.1. Limitaciones de los métodos de balanceo de carga convencionales en entornos SDN

Los algoritmos tradicionales para el balanceo de carga, son a menudo concebidos para entornos estáticos, siendo insuficientes para la naturaleza dinámica y compleja de los despliegues modernos de SDN [8] [16].

A continuación describimos las limitaciones de algunos algoritmos de balanceo de carga más comunes:

Algoritmo Dijkstra

Es un algoritmo clásico para encontrar la ruta más corta entre los nodos de un grafo, basándose en los pesos de arista (positivos). Sin embargo tiene muchas limitaciones en SDN. [17] [18]

Es un método estático, con escasa correspondencia con el estado de la red con el tiempo real, ya que el algoritmo Dijkstra se basa en heurísticas predefinidas y estrategias de optimización estáticas. Principalmente considera métricas fijas, como el número de saltos o coste de los enlaces, dejando a un lado el estado en tiempo real de los enlaces, como es la carga en dicho enlace. Con este enfoque nos puede llevar a unas soluciones subóptimas donde el tráfico es enviado continuamente a la ruta “más corta”, provocando una congestión de la red, sin tener en cuenta otras rutas menos utilizadas. [16]

También es incapaz de optimizar múltiples métricas, ya que es típicamente un algoritmo de objetivo único, como puede ser la ruta más corta, siendo nulo para satisfacer los requisitos de Calidad de Servicio de los servicios modernos que requieren de optimización simultánea de diferentes métricas (pérdida de paquetes, retardo o ancho de banda). [16]

Algoritmo Round Robin

Es una manera simple de distribuir las peticiones de los clientes por todo el grupo de servidores. Con este método, las peticiones de los clientes se

dirigen a los servidores disponibles de forma cíclica, funciona mejor cuando los servidores tienen capacidades computacionales y de almacenamiento prácticamente idénticas. [19] [20]

El mayor inconveniente de Round Robin (RR) es esa suposición, de que todos los servidores o enlaces tienen capacidades de cómputo idénticas y pueden manejar equivalentes cargas. No tiene mecanismos para distribuir un mayor número de solicitudes a servidores con mayor capacidad. [19]

No tiene detección de fallos, es decir, que continúan enviando solicitudes a los servidores aunque no estén funcionales. Al estar configurado estáticamente, al no adaptarse la carga del servidor en tiempo real ni a las condiciones dinámicas de la red, conduce a un balanceo de carga desigual, provocando una sobrecarga de conmutadores con una menor capacidad.[20]

Las capacidades de los algoritmos tradicionales no causan sólo un problema aislado, sino que se va encadenando una cascada de efectos negativos. La congestión provoca un aumento de la latencia, una mayor pérdida de paquetes y reducción del rendimiento [21]. Por eso los métodos estáticos son incapaces de adaptarse al tráfico dinámico.

1.1.2. Algoritmos Bioinspirados

Como ya hemos comentado en la sección anterior, los algoritmos tradicionales tienen muchas limitaciones para la optimización del balanceo de carga en SDN, debido a su naturaleza cambiante[22]. Por ello los algoritmos bioinspirados emergen como una solución muy prometedora para resolver este complejo problema.

En concreto, el algoritmo Ant Colony Optimization (ACO) destaca por su capacidad de:

- Aprender progresivamente los caminos más eficientes.
- Poder adaptarse a los cambios del entorno
- Balancear la exploración de nuevos caminos con la explotación de rutas ya descubiertas.

[15]

1.1.3. ¿Por qué los algoritmos bioinspirados son tan interesantes respecto a otros métodos de optimización?

Originalmente, los algoritmos bioinspirados fueron diseñados para resolver problemas tan complejos que los métodos tradicionales tardarían una cantidad de tiempo inviable para encontrar la solución óptima. Problemas

con una cantidad de posibilidades que ni el ordenador más potente del mundo podría revisarlas en una cantidad de tiempo razonable [15]. Estos son los *problemas NP-Complete* para los que los algoritmos bioinspirados obtienen una solución.

Lo más interesante es que estos se basan en las leyes físicas y en los mecanismos que la naturaleza ha perfeccionado durante millones de años, para resolver los tres mayores obstáculos que un ser vivo se puede enfrentar [15]: la reproducción (como se propagan y mejoran sus características), la búsqueda de alimento (encontrar los recursos necesarios) y la adaptabilidad al entorno (como evolucionan para sobrevivir en diferentes condiciones).

Con estos algoritmos podemos obtener soluciones casi óptimas a problemas complejos en un tiempo razonable. Además, muchos de los algoritmos bioinspirados se basan en la evolución de las poblaciones de individuos, siendo cada individuo una solución concreta de problemas [15]. Esta estructura los hace altamente paralelizables, permitiendo reducir considerablemente el tiempo de obtención de la solución casi óptima, aprovechando las ventajas de la programación paralela en una o varias máquinas.

Este trabajo tiene como objetivo aprovechar el control centralizado de SDN con la inteligencia adaptativa y distribuida de los algoritmos bioinspirados. Esta sinergia crea un gran marco para la gestión de la red que supera las capacidades de los enfoques tradicionales.

Las Redes Definidas por Software proporcionan una visión global de la red y una capacidad de ajustar dinámicamente las reglas de reenvío, mientras que los algoritmos bioinspirados, en nuestro caso ACO, nos proporciona la inteligencia para realizar estos ajustes de forma adaptativa basándose en las condiciones de la red en tiempo real [23]. Con el uso de este tipo de algoritmos conseguimos explorar espacios de soluciones complejos para encontrar rutas óptimas y asignación de recursos.

La necesidad de optimizar el balanceo de carga deriva directamente de la insuficiencia de métodos actuales para manejar las redes modernas [8]. Como ya detallamos anteriormente los algoritmos tradicionales (*Dijkstra*, *RR*), no están “*equipados*” para tratar con el tráfico dinámico de los entornos SDN actuales, ya que conducen a la congestión y mala utilización de los recursos de la red [11]. El crecimiento exponencial del tráfico de la red exige soluciones que puedan adaptarse instantáneamente previniendo cuellos de botella y mantener los *QoS* (Quality of Service).

Con el masivo crecimiento de usuarios móviles, el *big data*, la computación en la nube y el *Internet de las cosas*, se exige una alta calidad de servicio de comunicación y una gestión eficiente del tráfico de la red [24]. En concreto, los centros de datos experimentan un crecimiento exponencial del tráfico, lo que hace que la optimización y balanceo de carga sea indispensable hoy en día.

Más allá del rendimiento técnico, la eficiencia del balanceo de carga en *SDN* tiene un gran impacto económico [25]. Se ha observado/investigado que *SDN* “ahorra dinero” y “mantiene las redes funcionando sin problemas”, ya que se reduce el trabajo manual y automatizado. [26]

Además de aplicar la sinergia entre *SDN* y los algoritmos bioinspirados, este proyecto contribuye más allá de los enfoques anteriores en varios aspectos clave. En primer lugar, se propone una mejora sobre el algoritmo ACO en el balanceo de carga de redes *SDN* mediante la incorporación del enfoque ACS (Ant Colony System (Sistema de Colonias de Hormigas)), que permite tomar decisiones más inteligentes durante el proceso de selecciones de caminos.

En segundo lugar se ha desarrollado una herramienta web interactiva y en tiempo real, que permite visualizar no sólo la topología de la red, sino sus métricas, con la posibilidad de modificar dichas métricas, generar carga y observar el impacto de las decisiones del algoritmo sobre la red.

Finalmente, se ha llevado a cabo una fase experimental competa en un entorno *SDN* realista, comparando diferentes configuraciones del algoritmo, y distintos algoritmos de enrutamiento (como Dijkstra, LLBACO base y LLBACO mejorada) permitiendo validar de forma objetiva los beneficios del enfoque desarrollado.

1.2. Objetivos

Este Trabajo de Fin de Grado se enfoca en **desarrollar e integrar un algoritmo ACO que optimice el balanceo de carga en redes SDN**, permitiendo analizar rutas alternativas en función del estado en tiempo real de la red, mejorando así la distribución del tráfico.

1.2.1. Objetivos secundarios

Para poder lograr el objetivo principal, se definen los siguientes objetivos secundarios:

1. Investigar el estado del arte sobre el balanceo de carga en *SDN* y los algoritmos ACO.
2. Familiarizarse con el entorno de desarrollo y simulación de redes *SDN*, mediante las herramientas *Mininet* y el controlador *Ryu*.
3. Desarrollar un monitor/controlador en *Ryu* que proporcione las métricas de retardo, carga y pérdida de paquetes necesarias para el algoritmo ACO.
4. Implementar el algoritmo ACO adaptado al entorno *SDN*, e integrarlo con el controlador *Ryu* para que tome decisiones de enrutamiento basadas en las métricas obtenidas.

5. Evaluar los resultados del algoritmo implementado. Primero, comprobar que los resultados sean correctos y coherentes; después, probar diferentes configuraciones para determinar cuál es la óptima.
6. Comparar con otros algoritmos. Es necesario analizar algoritmos que ya se utilizan para el balanceo de carga, con el fin de evaluar la utilidad del enfoque propuesto.
7. Crear una herramienta o una aplicación web que permita a los usuarios visualizar la topología y el correcto funcionamiento del algoritmo.

1.3. Estructura

A continuación se describe brevemente de qué tratará cada capítulo:

- **Capítulo 2 - Conceptos preliminares:** Se explican los conceptos teóricos fundamentales necesarios para poder entender el resto del trabajo, como las *Redes Definidas por Software* y el algoritmo *Ant Colony Optimization*.
- **Capítulo 3 - Definición del problema:** Se realiza una formulación formal del problema que se va a resolver en este proyecto.
- **Capítulo 4 - Estado del arte:** Se analizarán diversos trabajos previos relacionados con la optimización del balanceo de carga en *SDN* mediante algoritmos bioinspirados, comparándolos con el enfoque planteado en este TFG (Trabajo Fin de Grado).
- **Capítulo 5 - Planificación:** Se detallará una planificación temporal del trabajo, mediante un diagrama de Gantt y un análisis de recursos y esfuerzo.
- **Capítulo 6 - Herramientas utilizadas:** Se presentan las diferentes herramientas utilizadas para el desarrollo e implementación del proyecto.
- **Capítulo 7 - Utilidad de Visualización de la Red:** Se explica la aplicación web desarrollada para que el usuario interactúe con la red.
- **Capítulo 8 - Algoritmos desarrollados:** Se describe el diseño y funcionamiento del algoritmo propuesto, detallando los pasos y sus componentes clave.
- **Capítulo 9 - Experimentación y resultados:** Se exponen los experimentos realizados para evaluar la propuesta, junto con los resultados obtenidos y un análisis crítico.

- **Capítulo 10 - Conclusiones:** Se resumen los logros alcanzados, se reflexiona sobre las aportaciones del trabajo y se propone una línea de trabajos futuros que extiendan esta investigación.

Capítulo 2

CONCEPTOS PRELIMINARES

Antes de sumergirnos con la optimización del despliegue de redes *SDN* por medio de los algoritmos Bioinspirados, es fundamental comprender los conceptos teóricos que sustentan este trabajo. En esta sección se presentan los principios esenciales de las *Redes Definidas por Software* y del algoritmo *Ant Colony Optimization* (ACO), así como otros elementos clave que permiten contextualizar la solución propuesta.

2.1. SDN

Las Redes Definidas por Software (SDN) representan un nuevo enfoque en la gestión de redes, ofreciendo una arquitectura más flexible, dinámica y manejable [27]. Todas estas características las hacen muy adecuadas para la naturaleza dinámica de las aplicaciones de red de hoy en día. La gran innovación es que desacopla el plano de control del plano de datos de un dispositivo de red [9] [10].

En este modelo, el plano de control es trasladado de los dispositivos de la red a un controlador lógico centralizado que actúa como sistema operativo de la red[28]. Proporcionando una visión global de la red, que a diferencia de las redes tradicionales, puede realizar cambios globales de una manera centralizada. Uno de los protocolos más representativos en este contexto es Openflow, el cual permite una comunicación estandarizada entre el controlador y los dispositivos del plano de datos.

Estructura de una red SDN

La arquitectura tradicional de SDN está compuesta de tres capas principales[5]:

- **Capa de datos (data plane):** Es la capa inferior, y está formada por dispositivos de red físicos o virtuales como switches, routers y puntos

de acceso [25]. Estos dispositivos no toman decisiones, sino que simplemente reenvían paquetes según las reglas instaladas por el controlador. Se comunican con la capa de control a través de interfaces como el protocolo OpenFlow, que permite acceder al plano de reenvío de forma directa y segura [29]. Además, es habitual el uso de conexiones seguras como TLS (Transport Layer Security) para proteger dicha comunicación.

- **Capa de control (control plane):** Es la capa intermedia compuesta por controladores SDN basados en software. Estos controladores centralizan la lógica de control de red, monitorean el comportamiento del tráfico y gestionan la infraestructura mediante la interfaz con el plano de datos [25]. También exponen interfaces hacia las aplicaciones de red, conocidas como A-CPI (*Application to Controller Plane Interface*), que permite a las aplicaciones obtener una visión abstracta del estado de la red y modificar su comportamiento [30]. Un controlador típico puede incluir componentes funcionales como virtualizadores, coordinadores, balanceadores de carga, sistemas de detección de intrusos o monitorización de red.
- **Capa de aplicación (application plane):** Es la capa superior, donde se desarrollan e implementan las aplicaciones que utilizan los servicios ofrecidos por el controlador SDN. Estas aplicaciones pueden incluir funciones de ingeniería de tráfico, balanceo de carga, enrutamiento, seguridad o virtualización [25]. Se comunican con el plano de control mediante la interfaz A-CPI, accediendo a una visión global y abstracta de la red para la toma de decisiones.

2.2. Algoritmos ACO

Ant colony optimization (ACO) es una técnica metaheurística ampliamente utilizada para encontrar las soluciones a complejos problemas de optimización. ACO fue inspirada por la observación del comportamiento de las colonias de hormigas reales[31] al buscar rutas eficientes entre su nido y fuentes de alimento [32].

Las hormigas son insectos sociales que viven en colonias, aunque tienen habilidades individuales limitadas, logran comportamientos altamente a nivel colectivo [31]. Una característica interesante del comportamiento de las colonias de hormigas es cómo pueden encontrar los caminos más cortos entre el hormiguero y la comida. Esta cooperación se produce a través de unas sustancias llamadas feromonas.

Cuando una hormiga encuentra comida, en su recorrido de vuelta deposita las feromonas que todas pueden oler [33]. Este rastro permite a las hormigas volver a su hormiguero desde la comida,

Cuanto más corta y frecuentemente recorrida sea esa ruta, habrá una mayor concentración de feromonas en esa ruta, lo que intensificará que más hormigas elijan esa opción. Por otro lado, las rutas menos óptimas ven desaparecer sus rastros de feromonas, por evaporación, reduciendo su atractivo. La acción continuada de la colonia da lugar a un rastro de feromona que permite a las hormigas encontrar un camino cada vez más corto desde el hormiguero a la comida [33].

2.2.1. Funcionamiento general de ACO

1. **Inicialización:** se asigna una cantidad inicial de feromonas a cada posible camino solución.
2. **Construcción de soluciones:** las hormigas van generando soluciones, eligiendo el siguiente nodo hacia su destino, con una probabilidad basada en la feromona y una heurística del problema (distancia, carga...).
3. **Actualización de feromonas:** las rutas más prometedoras refuerzan sus feromonas, mientras que las demás sufren evaporación
4. **Iteración:** el proceso se repite durante un número determinado de iteraciones (que realizan cada hormiga).

[34]

2.2.2. Explicando ACO a través de un ejemplo

La mejor manera de entender cómo funciona ACO, es viendo como se aplica al clásico **Travelling Salesman Problem** (Problema del viajante de comercio) (TSP).

Se dispone de un conjunto $N = 1, \dots, n$ de ciudades, que han de ser visitadas una sola vez, volviendo a la ciudad de origen, y recorriendo la menor distancia posible [35]

- Cada hormiga representa un posible viajante que construye una ruta ciudad por ciudad. [36]
- La elección de la próxima ciudad se hace de forma probabilística, ponderada por:
 - La cantidad de feromona en la ruta entre dos ciudades
 - Una heurística, como el inverso de la distancia (las ciudades más cercanas son las preferidas)
- Tras completar la ruta, las hormigas mejor evaluadas refuerzan las hormonas de sus trayectorias [36]

- Este proceso iterativo, va permitiendo que las peores rutas sean eliminando (su concentración de feromonas será menor) y dejando paso a las rutas óptimas. [36]

Tras haberse consolidado el concepto en el ejemplo del viajante de comercio, es mucho más sencillo traspasarlo a la selección de rutas óptimas en redes SDN.

- Las ciudades serán sustituidas por *switches*.
- Las distancias sustituidas por las métricas de la red como *delay*, *carga* y *pérdida de paquetes*.
- La colonia tiene como objetivo minimizar el coste total desde un nodo origen a un nodo destino en base a las métricas obtenidas por el controlador.

2.3. Openflow

OpenFlow es un protocolo de comunicación de red fundamental en la arquitectura de redes definidas por software, se utiliza como un protocolo *southbound*, facilitando la comunicación entre el controlador SDN, que actúa como el cerebro de la red, y los dispositivos de reenvío de tráfico, como conmutadores y routers [37]. De hecho, OpenFlow se considera elAPI (Application Programming Interface) *southbound* estándar de la industria para la implementación de entornos SDN [38]. Su función principal es permitir la separación de las funciones de control y reenvío de la red, un principio central de SDN. [1].

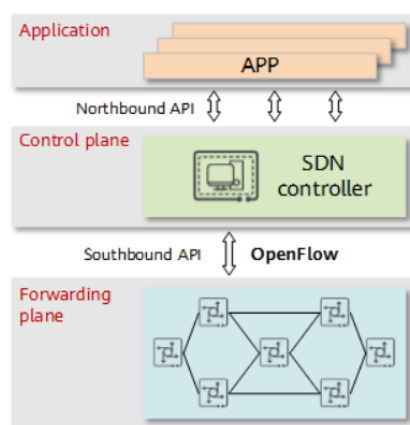


Figura 2.1: Openflow en SDN. Fuente: [1]

2.3.1. ¿Cómo funciona OpenFlow? [1]

Controlador : controla la red de manera centralizada para implementar las funciones del plano de control.

Conmutador : es el responsable del reenvío de la capa de datos, intercambia mensajes con el controlador a través de un canal seguro para recibir entradas de reenvío e informar de su estado

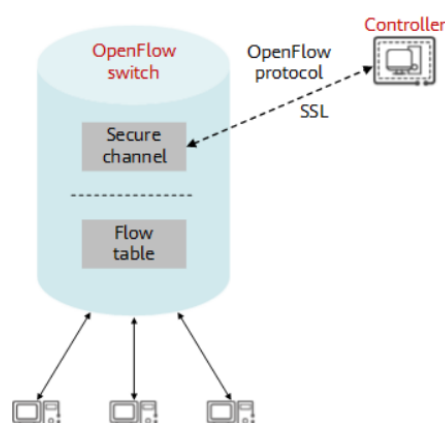


Figura 2.2: Arquitectura de Openflow. Fuente: [1]

Canal seguro

Se establece un canal seguro entre el controlador y el conmutador de Openflow. A través de este canal el controlador gestiona el conmutador, y recibe retroalimentación del conmutador.

Los mensajes deben cumplir con el formato especificado por el protocolo Openflow. El canal seguro es cifrado mediante seguridad de capa de transporte (TLS), pero puede ejecutarse directamente sobre TCP (Transmission Control Protocol) (en Openflow 1.1).

Tipos de mensajes:

Mensaje de Controlador a Conmutador : enviado del controlador al conmutador para obtener el estado del conmutador.

Mensaje Asíncrono : enviado por el conmutador OpenFlow al controlador para actualizar eventos de red o cambios de estado al controlador.

Mensaje Síncrono : enviado sin solicitud por el conmutador OpenFlow o el controlador. Se utiliza principalmente para establecer una conexión y detectar si el par está en línea.

Capítulo 3

DEFINICIÓN DEL PROBLEMA

Las Redes Definidas por Software son una de las tecnologías más utilizadas en la actualidad, y permiten el diseño y administración de las redes de una forma dinámica [37]. Existen diversas cuestiones a la hora de realizar dicho despliegue, que pueden afectar en mayor o menor medida al rendimiento de la red. En este TFG se atacará uno de esos problemas, como es el balanceo de carga. De modo que se desarrollará el algoritmo de Optimización de Hormigas (ACO) que permita optimizar el despliegue de una SDN con el fin de maximizar su rendimiento en función del balanceo de carga de la red software.

Enunciado formal del problema

Dado un grafo dirigido $G = (V, E)$, donde:

- V representa el conjunto de nodos de la red (switches SDN),
- E representa el conjunto de enlaces entre nodos, cada uno t:
 - $c(e)$: carga del enlace e (load), es la cantidad de ancho de banda utilizado de un enlace de red [39].
 - $d(e)$: retardo estimado (delay) en ms, es el tiempo que tarda un paquete de datos en viajar desde su origen hasta su destino a través de la red [40].
 - $l(e)$: pérdida de paquetes (packet loss) en %, indica el porcentaje de paquetes enviados a través de una red que no logran llegar a su destino [41].
- Un nodo origen $a \in V$ y un nodo destino $b \in V$,

Se busca determinar el camino óptimo $P = \{a \rightarrow b\}$ que minimice coste, basado en las métricas anteriores. En consecuencia, el algoritmo debe ser capaz de:

- Adaptarse a cambios dinámicos en el estado de la red en tiempo real.
- Tomar decisiones de enrutamiento que distribuyan eficientemente la carga de tráfico.
- Minimizar simultáneamente el delay, la carga y la pérdida de paquetes.
- Ser implementable y ejecutable sobre un controlador SDN real, como Ryu.

Para ello, se ha utilizado una variante del algoritmo de optimización por colonia de hormigas (ACS), debido a su capacidad para explorar múltiples rutas, adaptarse al entorno y encontrar soluciones cercanas al óptimo en problemas de alta complejidad.

Relevancia del problema

La eficiencia del balanceo de carga no solo afecta al rendimiento de la red, sino que incide directamente en el uso racional de los recursos y la calidad de experiencia del usuario (Quality of Experience (QoE)). En contextos como centros de datos, redes móviles o infraestructuras cloud, su impacto es aún más significativo.

Este TFG busca contribuir con una solución realista, flexible y eficaz, evaluada sobre un entorno de simulación práctico, que permita una comparación directa con enfoques tradicionales y ponga de manifiesto las ventajas de las técnicas bioinspiradas en entornos SDN.

Capítulo 4

Estado del arte

El problema del balanceo de carga en las Redes Definidas por Software (SDN) se ha convertido en un área de investigación activa, dada la creciente necesidad de gestionar eficientemente los recursos en las redes complejas y dinámicas. Una línea muy prometedora de soluciones se basa en algoritmos bioinspirados, concretamente en Ant Colony Optimization (ACO) y sus variantes. En este capítulo analizaremos diferentes trabajos en este campo, destacando sus similitudes con el presente TFG, así como las limitaciones que presentan frente al enfoque propuesto.

4.1. The Load Balancing of SDN based on ACO with Job Classification

Este artículo [42] propone su enfoque jerárquico para el balanceo de carga en SDN, combinando una arquitectura de controladores distribuidos con una optimización basada en el algoritmo ACO.

La arquitectura propuesta consta de un controlador central cuyo trabajo es monitorear toda la red, encargado de coordinar varios sub controladores, cada uno de los cuales gestiona una subred homogénea compuesta por nodos con características similares.

La innovación en la que se centra este artículo es la **clasificación de tareas** antes de la asignación. Las tareas están clasificadas según su demanda de CPU, y son asignadas a subredes especializadas. Posteriormente, dentro de cada subred, se utiliza ACO para seleccionar un nodo específico que ejecutará cada tarea. La optimización se basa en el estado de la carga actual de los nodos, utilizando feromonas y funciones de coste que tienen en cuenta la disponibilidad de recursos.

4.1.1. Algoritmo Planteado

1. **Representación del problema:** la red es representada como un grafo, donde los nodos son servidores o dispositivos de red y los enlaces tienen asociada una métrica de carga.
2. **Métrica de selección de enlaces:** la métrica más influyente de la red es la carga de los enlaces. La idea es que la concentración de feromonas en un enlace es inversamente proporcional a su tasa de utilización. Es decir, que cuanto menos carga tenga un enlace mayor concentración de feromonas contendrá y será más atractivo para las hormigas.
3. El objetivo del algoritmo es calcular el camino con una carga mínima entre el nodo de entrada y el nodo de salida dentro de la subred elegida

4.1.2. Comparación con el TFG

Ambos trabajos comparten el uso de ACO para la optimización del balanceo de carga en SDN, sin embargo hay unas diferencias clave:

- **Arquitectura:** el artículo se basa en una arquitectura jerárquica con una clasificación previa de trabajos, mientras que el presente TFG propone una solución integrada y dinámica que actúa en tiempo real sobre la red sin necesidad de clasificación previa ni subredes homogéneas.
- **Aplicación ACO:** en el TFG se aplica directamente sobre la topología de la red utilizando métricas como el retardo, la carga y la pérdida de paquetes, datos obtenidos mediante monitorización activa, permitiendo una adaptación más flexible en redes cambiantes
- **Objetivo de optimización:** el artículo se enfoca en la asignación de tareas computacionales, mientras que en el TFG se aplica ACO para la selección de rutas óptimas, maximizando el rendimiento general de la red.

4.2. Joint route-server load balancing in software defined networks using ACO

Este artículo [27] propone una solución para el balanceo de carga dinámico en Redes Definidas por Software, combinando la selección dinámica de rutas y servidores mediante el algoritmo *ACS* (Ant Colony System) una variante optimizada del clásico ACO.

Este trabajo tiene un objetivo doble: optimizar simultáneamente **la selección del servidor más adecuado** (el menos cargado) y **la mejor ruta del cliente al servidor**, aprovechando la visión global de la red que ofrece SDN.

Para ello hacen uso del controlador *OpenDaylight*, se encarga de monitorizar en tiempo real las métricas de la red y las estadísticas de carga de CPU de los servidores.

Con esta información:

- Selecciona el servidor menos cargado, basado en la información recogida por el controlador.
- Mediante el algoritmo *ACS* encuentra el camino más óptimo teniendo en cuenta el estado dinámico de la red y las feromonas depositadas en rutas previas.
- Cuanto menos carga tenga un enlace en un momento dado, mayor concentración de feromonas contendrá, siendo más atractivo para las hormigas seleccionar esos enlaces como parte de su camino.

4.2.1. Comparación con el TFG

Similitudes con el TFG:

- Ambos trabajos se centran en la optimización en el balanceo de carga en SDN
- Ambas implementaciones utilizan la variante ACS para la optimización de rutas.
- En ambos enfoques la carga es una métrica fundamental para la selección de enlaces por las hormigas.

Carencias y mejoras:

- **Poca generalización:** está diseñado específicamente para escenarios cliente-servidor, donde es posible monitorizar directamente la CPU de los servidores, lo que limita su aplicabilidad a otras configuraciones más generales. En cambio, en mi propuesta se aplica ACS para el balanceo de carga abordando un escenario más genérico de múltiples nodos y rutas, haciéndolo más adaptable a distintas topologías.
- **Métricas de la red:** el algoritmo ACS propuesto en el artículo, solo tiene en cuenta la métrica de la carga de los enlaces y los servidores. En mi implementación se tienen en cuenta aparte de la carga, la pérdida de paquetes y el delay, mejorando así la optimización.

4.3. Dynamic Load Balancing of Software-Defined Networking Based on Genetic-Ant Colony Optimization

Este artículo [6] presenta un enfoque híbrido para el balanceo de carga en redes SDN, basado en una combinación del Ant Colony Optimization (ACO) y Genetic Algorithm (Algoritmo Genético) (GA), denominado **G-ACO**. Esta fusión tiene como objetivo principal superar las limitaciones de cada estrategia por separado, aprovechando la capacidad de exploración adaptativa de ACO junto con la exploración de espacios de soluciones y diversidad genética que ofrecen los algoritmos evolutivos.

4.3.1. Algoritmo planteado

El algoritmo G-ACO está desarrollado en dos fases:

1. **Fase de inicialización (ACO):** Las hormigas recorren la topología de la red generando múltiples rutas de un nodo origen hacia un nodo destino, depositando feromonas en función de métricas de la red (carga y delay). Básicamente consiste en la aplicación clásica del algoritmo ACO
2. **Fase de optimización (GA):** las rutas generadas por ACO son codificadas como individuos (cromosomas) en un algoritmo genético, aplicando:
 - **Selección:** las rutas con menor coste son prioritarias.
 - **Crossover (cruce):** las rutas exitosas son combinadas para crear potenciales mejores soluciones.
 - **Mutación:** son introducidas variantes aleatorias para poder encontrar rutas inexploradas.

Este proceso iterativo se repite durante generaciones para quedar atrapado en óptimos locales.

4.3.2. Comparación con el TFG

Similitudes con el TFG:

- Ambos trabajos tienen como objetivo principal la optimización del balanceo de carga en redes SDN.
- Ambos enfoques utilizan ACO como núcleo del algoritmo, inspirado en el comportamiento de las colonias de hormigas.

- En ambos trabajos la carga del enlace es una métrica fundamental para guiar a las hormigas a seleccionar caminos.

Carencias y mejoras:

- **Complejidad computacional:** la combinación de ACO y GA incrementa considerablemente la complejidad espacial y temporal del algoritmo. Esto dificulta su implantación en controladores SDN en tiempo real o entornos de red limitados.
- **Optimización de un solo algoritmo:** el G-ACO, mediante sus combinación busca al potencia de ambos algoritmos, pero esto podría no explotar al máximo las capacidades de ACO. En mi solución al enfocarnos en la única variante de ACO (ACS), podemos demostrar que un algoritmo basado exclusivamente en ACS es suficiente para el problema del balanceo de carga, ofreciendo un buen equilibrio entre rendimiento, complejidad y eficiencia computacional.
- **Viabilidad y realismo:** en el artículo podemos observar que G-ACO mejora las métricas de la red, aunque solo evaluado en simulaciones. En mi solución, se implementa sobre Mininet y Ryu, acercándose a escenarios reales y no únicamente soluciones teóricas.

4.4. Conclusiones

Tras el análisis de los trabajos revisados, se pueden extraer varias conclusiones:

- Existe consenso en que ACO y sus variantes (ACS, G-ACO) son herramientas potentes para abordar la optimización del balanceo de carga en SDN.
- Sin embargo, en los trabajos analizados presentan ciertas limitaciones: como restricciones de arquitectura (enfoques jerárquicos o cliente-servidor), consideración parcial de métricas (no tiene en cuenta el delay o packet loss), alta complejidad computacional o validaciones únicamente teóricas.
- En el presente TFG se plantea una solución más general, modular y realista. Empleando ACS sobre una red SDN creada con *Mininet* y controlada por *Ryu*, teniendo en cuenta métricas claves en la red (carga, delay y packet loss), mediante la ejecución de pruebas repetidas y ofreciendo un adecuado equilibrio entre rendimiento y eficiencia.

Por lo tanto, este TFG no solo contribuye al estado del arte con un enfoque implementado y probado en un entorno realista, sino que también propone una solución eficiente y práctica que puede ser utilizada como una base para futuros desarrollos en la gestión dinámica de redes SDN.

Capítulo 5

Planificación

En este capítulo se describen las distintas fases en las que se ha dividido el proyecto. Asimismo, se presentará una estimación del coste total asociado al desarrollo del trabajo, diferenciando entre las horas atribuidas al estudiante (asumiendo el rol de ingeniero en formación) y las del tutor académico (considerado como ingeniero sénior o analista).

5.1. Fases de proyecto

Como se muestra en la figura 5.1 el desarrollo del proyecto se divide en las siguientes fases:

1. **Reuniones iniciales:** para comenzar se han tenido una serie de reuniones con el tutor, para establecer las bases del proyecto, entender las necesidades y objetivos del mismo.
2. **Investigación**
 - 2.1. **Búsqueda sobre SDN:** entender realmente qué son las Redes Definidas por Software y cómo están formadas.
 - 2.2. **Búsqueda sobre ACO:** comprender el funcionamiento del algoritmo de Colonia de Hormigas, analizar distintas implementaciones sobre diversos problemas.
 - 2.3. **Estudio del balanceo de carga en redes SDN:** investigar sobre el impacto del balanceo de carga en redes SDN, y los distintos algoritmos utilizados para su optimización.
3. **Implementación ACO para TSP:**
 - 3.1. **Investigación sobre ACO para TSP:** una vez entendido el algoritmo de Optimización De Hormigas, ver su uso en un problema clásico como TSP.

- 3.2. **Implementación ACO para TSP:** antes de realizar su implementación en el contexto de redes SDN, realizar su implementación para el problema TSP, con el objetivo de comprender con su funcionamiento.
4. **Contextualizar a redes SDN**
 - 4.1. **Definir topología SDN:** mediante el uso *Mininet* para la creación de las topologías de red que se emplearán en la ejecución y validación del algoritmo propuesto.
 - 4.2. **Controlador/monitor SDN:** implementación del controlador SDN *Ryu* encargado de gestionar el comportamiento de la red definida por software (SDN) y de recopilar métricas relevantes sobre su funcionamiento
 - 4.3. **ACO para SDN:** una vez implementado el algoritmo de optimización por colonias de hormigas (ACO) para el problema del viajante (TSP), se procederá a su adaptación al entorno de redes definidas por software, ajustándose a las necesidades de la topología y de enrutamiento específicos del entorno SDN.
5. **Visualización gráfica de la red:** Se implementará una herramienta web que permita a los usuarios interactuar y visualizar la red en tiempo real, con el objetivo de ver el correcto funcionamiento del algoritmo:
6. **Implementación ACS:** Se mejorará el algoritmo previamente desarrollado mediante la implementación de la variante Ant Colony System (ACS), con el objetivo de optimizar el rendimiento y la eficiencia en la resolución del problema de enrutamiento en redes definidas por software.
7. **Experimentación y pruebas:** Se ejecutará el algoritmo mediante diferentes configuraciones, variando los parámetros relevantes del mismo. Además se comparará el rendimiento de la implementación basada en Ant Colony System (ACS) con la versión inicial basada en ACO y con algoritmos clásicos como Dijkstra.
8. **Memoria y Documentación:** Finalmente, este documento ha sido elaborado a partir de anotaciones recogidas a lo largo de las fases anteriores.

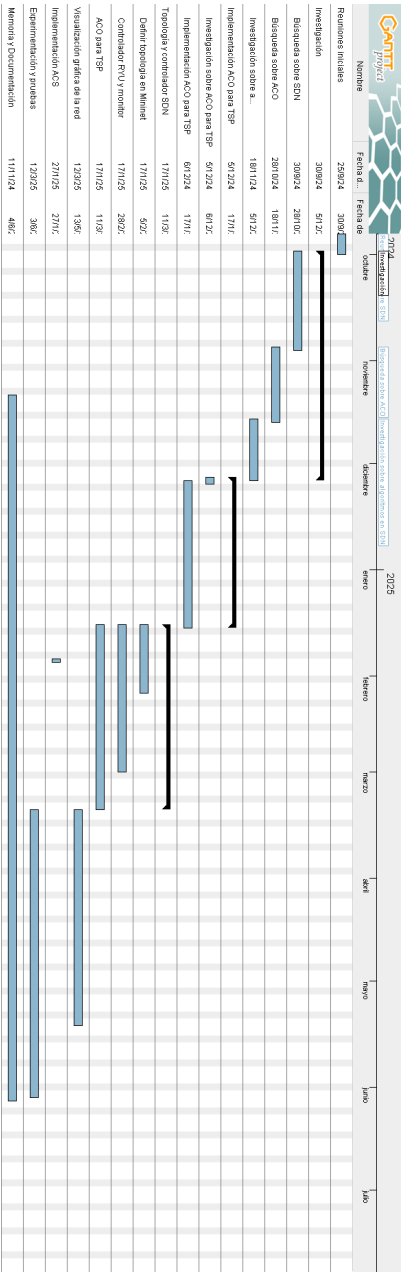


Figura 5.1: Diagrama de Gantt con la cronología de la realización de las diferentes tareas

5.2. Presupuesto estimado

A continuación se presenta una estimación del coste total del proyecto, estimado según las horas de trabajo del estudiante y del tutor, con tarifas horarias diferenciadas.

Fase	Horas estudiante	Horas tutor	Coste estimado(€)
Reuniones iniciales	4	2	$(4 \times 15) + (2 \times 40) = 140$
Investigación	30	8	$(30 \times 15) + (8 \times 40) = 710$
Implementación ACO para TSP	40	10	$(40 \times 15) + (10 \times 40) = 1000$
Contextualización SDN	65	12	$(65 \times 15) + (12 \times 40) = 1455$
Visualización gráfica	50	6	$(50 \times 15) + (6 \times 40) = 990$
Implementación ACS	25	8	$(25 \times 15) + (8 \times 40) = 595$
Experimentación y pruebas	40	8	$(40 \times 15) + (8 \times 40) = 920$
Memoria y documentación	50	12	$(50 \times 15) + (12 \times 40) = 1230$
Total	304	66	7040 €

Cuadro 5.1: Estimación de horas y coste del proyecto (estudiante: 15€/h, tutor: 40€/h)

Capítulo 6

HERRAMIENTAS Y DATOS

6.1. Mininet

Mininet es un emulador de red el cual crea una red virtual, con hosts switches, controladores y enlaces. Perfecto para desarrollar y experimentar con Software-Defined Networking [43]. Los hosts de Mininet se ejecutan en Linux ya que es una herramienta compatible exclusivamente con sistemas operativos Linux y los switches utilizan Openflow de acuerdo a SDN.

Una de las grandes ventajas de Mininet es que permite crear prototipos de redes complejas en un único portátil o ejecutar aplicaciones reales en los hosts emulados e interactuar con ellos, acelerando el ciclo de desarrollo y prueba [43].

6.1.1. ¿En qué consiste una red de Mininet?

Una red de Mininet [2] está compuesta de varios elementos que trabajan conjuntamente para simular un entorno de red:

- Hosts Virtuales: son procesos a nivel de usuarios que se ejecutan al espacio de la red de Linux. Esto les proporciona propiedad exclusiva de las interfaces, puertos y tablas de enrutamiento. Para las aplicaciones que se ejecutan en ellos, estos hosts tienen un comportamiento de máquinas reales.
- Enlaces emulados: para poder conectar los hosts y switches. La velocidad de datos, el retardo y la pérdida de paquetes en estos enlaces pueden configurarse por Linux Traffic Control (tc), permitiendo simular diferentes condiciones de la red
- Switches emulados: El Linux Bridge por defecto o el Open vSwitch, un switch virtual de código abierto compatible con Openflow, se utiliza para conmutar paquetes entre interfaces. Los conmutadores y enrutadores pueden ejecutarse en el kernel o en el espacio de usuario.

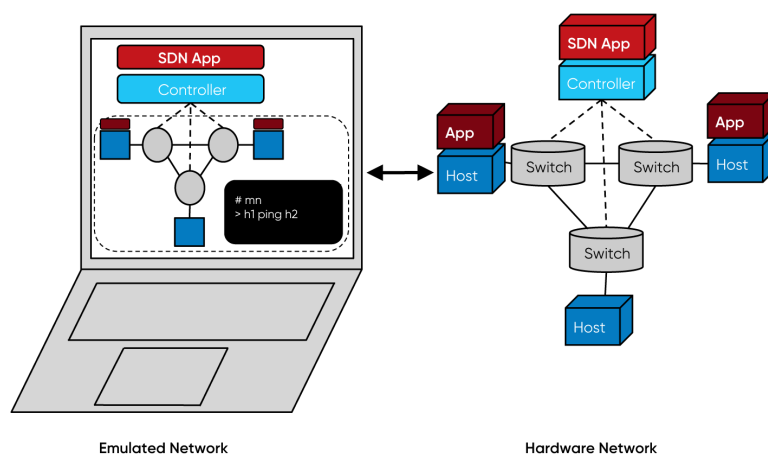


Figura 6.1: Estructura de una red Mininet. Fuente [2]

- Controlador (externo): Mininet no incluye un controlador SDN, sino que está diseñado para conectarse a un controlador SDN externo. El controlador es el “cerebro de la red”, tomando diferentes decisiones del flujo de la red.

6.2. Controlador SDN Ryu

Como hemos explicado anteriormente en el esquema de SDN, una parte fundamental es el controlador. En este proyecto se ha utilizado RYU, el cual es un framework de código abierto escrito en Python [44, 3]. El framework facilita el desarrollo proporcionando funcionalidades básicas para controlar la capa de datos y las funciones comunes en las aplicaciones SDN [44].

Arquitectura

Las interfaces southbound son los mecanismos que utiliza Ryu para comunicarse con los dispositivos de red físicos, como los switches SDN (el controlador piensa “arriba” y se comunica con los switches “abajo”), esa comunicación hacia abajo se realiza a través de estas interfaces [44].

Por ejemplo: Ryu usa Openflow para enviar instrucciones a los switches, “si un paquete viene del switch A y va para el switch B, envíalo por el puerto 3” y también para recibir información de ellos, “ha llegado un paquete nuevo que no se como gestionar”.

El “núcleo” (core) de Ryu es la parte central del framework, tiene algunas aplicaciones básicas:

- Topology discovery (descubrimiento de la topología): permite a Ryu entender cómo están conectados los switches en la red [44].

- Learning Switch (Switch de aprendizaje): imita el comportamiento de un Switch tradicional (de capa 2), aprendiendo las direcciones MAC de los dispositivos conectados a sus puertos [44].

Las aplicaciones externas (programadas por cualquier desarrollador), pueden ser desplegadas políticas de red, es decir, configurar reglas y comportamientos en los planos de datos (los switches que reenvían tráfico) [44, 3].

Northbound APIs: son las interfaces que exponen el controlador Ryu hacia arriba, para que las aplicaciones puedan interactuar con él y darle instrucciones. Si antes hablábamos de southbound para hablar con los switches, el northbound es para que otras aplicaciones hablen con Ryu. Es muy común utilizar la arquitectura REST para ello, para poder interactuar con el controlador a través de peticiones HTTP (Hypertext Transfer Protocol), de forma similar a como un navegador interactúa con un servidor web [44, 3].

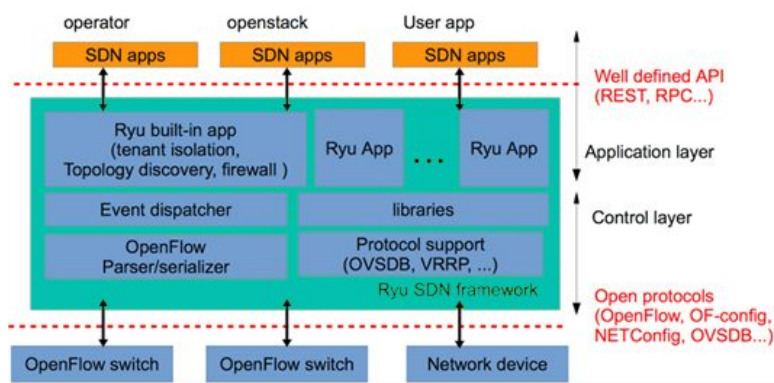


Figura 6.2: Arquitectura de Ryu. Fuente [3]

Modularidad y extensibilidad

Ryu es diferente a otros controladores SDN porque no proporciona una solución con múltiples funcionalidades predefinidas y una interfaz gráfica compleja. En lugar de eso, proporciona una infraestructura de soporte simple, un esqueleto básico con funciones simples [3].

Los usuarios tienen que escribir el código para utilizar la infraestructura y construir las funcionalidades necesarias [3].

En resumen Ryu proporciona los cimientos y las herramientas, pero tú construyes la casa (la aplicación SDN que necesitas) [3].

6.3. Flask

Flask es un microframework basado en Python que permite el desarrollo rápido de aplicaciones web ligeras y APIs RESTful con una mínima cantidad

de código [45]. A pesar de su simplicidad, Flask ofrece capacidades robustas para exponer servicios web, manejar peticiones HTTP y servir eficientemente contenido estático como JavaScript, CSS, plantillas HTML... [46]

6.3.1. Comunicación entre Flask y el controlador RYU

Como se muestra en la figura 6.3, el flujo de comunicación comienza con el usuario, que interactúa con la aplicación web a través de un navegador. Esta interacción incluye acciones como modificar parámetros, como definir el nodo origen o destino, modificar métricas de enlace o generar tráfico o visualizar el estado de la red.

Flask actúa como intermediario entre el usuario y el plano de control de la red. Flask recibe peticiones HTTP o SocketIO por parte del navegador y las traduce en peticiones HTTP POST que son enviadas al controlador RYU

En el lado de Ryu, un manejador WSGI (Web Server Gateway Interface) ('ControlHttpApp.py') actúa como servidor REST, procesando las solicitudes recibidas desde Flask. Este manejador traduce las peticiones en eventos que se colocan en una cola interna del sistema.

Finalmente el controlador, lee los eventos de la cola y ejecuta las acciones correspondientes, como la actualización de las métricas, el cambio de nodo origen o destino o la generación de tráfico.

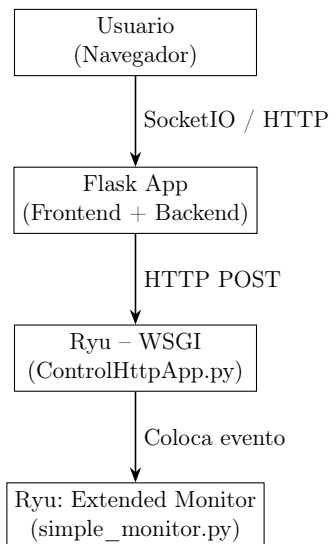


Figura 6.3: Flujo de comunicación entre el navegador, Flask y el controlador Ryu (elaboración propia)

6.3.2. Visualización de la red con D3

Data-Driven Documents JavaScript (D3.js) es una biblioteca de JavaScript diseñada específicamente para la creación de visualizaciones interactivas y altamente personalizables [47]. Ha sido utilizada en este proyecto para representar visualmente la topología de red en la interfaz web del usuario. Gracias a su capacidad de manipular el DOM basándose en los datos [48] y aplicar transformaciones dinámicas, pudiendo mostrar en tiempo real la estructura y el estado de la red SDN, incluyendo métricas clave como carga, delay o pérdida de paquetes.

Sinergia con Flask

Flask actúa como el servidor *backend* que proporciona la información estructurada de la red al *frontend*, donde D3 gestiona la visualización, leyendo los datos y generando una visualización interactiva de la red actualizada en tiempo real, como se aprecia en la figura 6.4. [48]

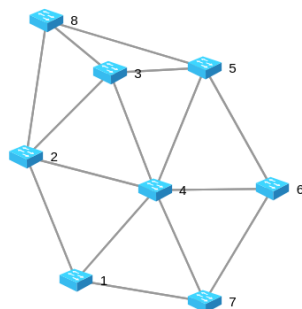


Figura 6.4: Visualización inicial de la topología de red con D3.js

Estas visualizaciones proporcionan al usuario una representación clara del estado actual de la red, permitiendo observar cómo cambian las condiciones de los enlaces y facilitando la interpretación de los resultados de los algoritmos ejecutados.

Capítulo 7

Utilidad de Visualización de la Red

En este capítulo se describe la utilidad/aplicación web desarrollada como complemento del trabajo, cuyo propósito es facilitar la interacción del usuario con la red definida por software. Además, permite la visualización en tiempo real de su topología, las métricas de cada enlace, la aplicación muestra cómo evoluciona el estado de la red a medida que el algoritmo de optimización propone nuevas soluciones.

De este modo, el usuario puede observar gráficamente la evolución progresiva de la carga, los costes y los caminos seleccionados por el algoritmo en distintos estados de la red.

7.1. Tecnologías utilizadas

La herramienta ha sido desarrollada como una aplicación web basada en:

- **Flask**: actúa como servidor web y backend de la aplicación.
- **Socket.io**: utilizada para establecer la comunicación bidireccional entre el navegador y el servidor (Flask).
- **HTTP/WSGI**: interfaz de comunicación entre Flask y el controlador Ryu para enviar comandos y recibir métricas.
- **D3.js**: para la visualización de la topología de la red.

La visualización de la red se actualiza automáticamente cada 5 segundos gracias a un mecanismo de *snapshots periódicos*. Estos snapshots son capturas del estado actual de la red, generadas por el monitor de Ryu, y contienen información relevante como la carga, el delay y la pérdida de paquetes en cada enlace. Posteriormente, estos datos se envían al servidor Flask, que los

transmite al navegador del usuario a través de Socket.IO, actualizando así la representación gráfica en tiempo real.

7.2. Acciones Disponibles para el Usuario

El usuario, a través de un navegador web, puede interactuar con la red mediante diferentes acciones, tal como se muestra en la Figura 7.1 en la parte superior se puede visualizar: el número de **instantánea** mostrada, la **ruta óptima** y su **coste**.

Visualización de la Red SDN

Instantánea: 4 Ruta óptima: Ruta óptima: 15 → 16 → 17 → 18 → 19 → 20 → 1 Coste: 0.0462

Pausa **Continuar** **Guardar Instantánea**

Seleccionar Origen y Destino

Origen (DPID): 15 Destino (DPID): 1 **Establecer**

Control de Parámetros de Enlace

Origen (DPID): 1 Destino (DPID): 10

Parámetro: Retardo (ms) 5 **Aplicar**

Generar Tráfico (iperf)

Host Origen (hX): h2 IP Destino (ej. 212.18.0.1 para h1): 212.18.0.1

Ancho de Banda (Mbps): 50 Duración (s): 60 **Iniciar Tráfico**

Figura 7.1: Visión general de la web

A continuación se describen las diferentes acciones que el usuario puede realizar.

Manejo de instantánea

El usuario puede realizar tres acciones sobre la instantánea: **Pausar** (para poder ver detenidamente la topología), **Continuar** (para que se sigan ejecutando instantáneas y **guardar instantánea** (para poder guardarla y ejecutar pruebas sobre ella, como se puede ver en la figura 7.2.

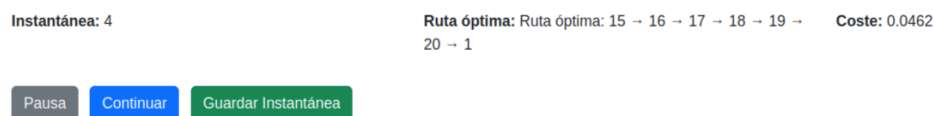


Figura 7.2: Botones de acción sobre la instantánea

Elegir el camino

Como el objetivo de nuestro algoritmo es encontrar la ruta óptima entre el **nodo origen** y **nodo destino**, es muy útil cambiar estos parámetros como se muestra en la figura 7.3.

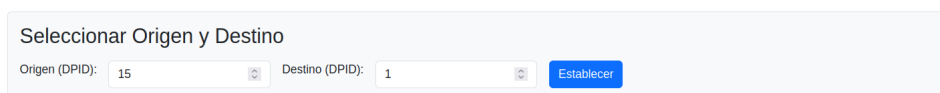


Figura 7.3: Cambiar nodo origen y nodo destino

Modificación de métricas

Esta parte incluye un mayor número de parámetros modificables como podemos apreciar en la figura 7.4. Primero tendremos que seleccionar el enlace a modificar (tiene que existir en la red) seleccionando el nodo origen y destino.

Después seleccionar la métrica a modificar:

- **Delay** del enlace medido en *ms*, siendo 1000 el máximo.
- **Pérdida de paquetes** medida en % siendo 100 % lo máximo.
- **Ancho de banda**, limitado dicha métrica del enlace en el número que indicamos.

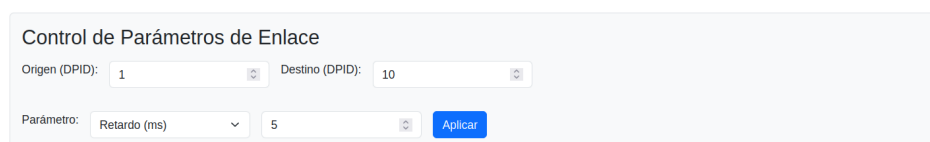


Figura 7.4: Modificación de delay del enlace a 10 ms

Generar carga

Para poder observar el correcto funcionamiento del algoritmo, es muy interesante la posibilidad de generar el tráfico que nos interesa. La figura 7.5 muestra cómo es posible generar tráfico desde un host origen a un host destino especificando el **Ancho de banda** y la duración de la herramienta iperf, una herramienta de medición activa del ancho de banda máximo alcanzable en redes IP [49].

Formulario de Generar Tráfico (iperf) con los siguientes campos:

- Host Origen (hX): h2
- IP Destino (ej. 212.18.0.1 para h1): 212.18.0.1
- Ancho de Banda (Mbps): 50
- Duración (s): 60
- Botón: Iniciar Tráfico

Figura 7.5: Generación de carga de host X a host Y

7.2.1. Visualización de la topología

Este apartado de la aplicación es posiblemente el más relevante, ya que es el responsable de que podamos ver la información de la red en tiempo real como apreciamos en la figura 7.6.

Mejor camino

Además de que en la parte superior nos mostrará los nodos que forman parte del mejor camino, en la visualización de la topología se puede ver resaltado en rojo el mejor camino de esa instantánea.

Información del enlace

En la topología mostrada también es posible mostrar la información de cada enlace, mediante un simple click en cualquier enlace, como se observa en la figura 7.7 la información del enlace que nos interese.

Se nos abrirá una tabla a la derecha de la topología, la cual muestra:

- Los nodos involucrados en el enlace
- El delay en ms
- La pérdida de paquetes en %
- La carga del enlace

También, como se observa en la figura 7.8 la topología destacará en azul el enlace clicado.

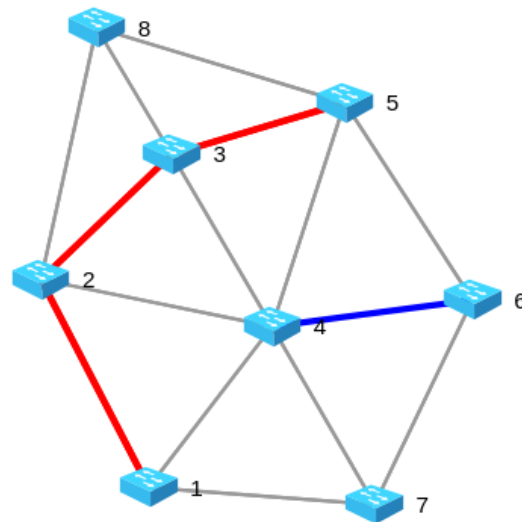


Figura 7.6: Topología de red

Información del Enlace Seleccionado

Enlace: 4 → 6

Carga: 0.0028800 %

Delay: 4.0689707 ms

Pérdida de Paquetes: 0.00 %

Figura 7.7: Tabla de métricas al clicar en el enlace

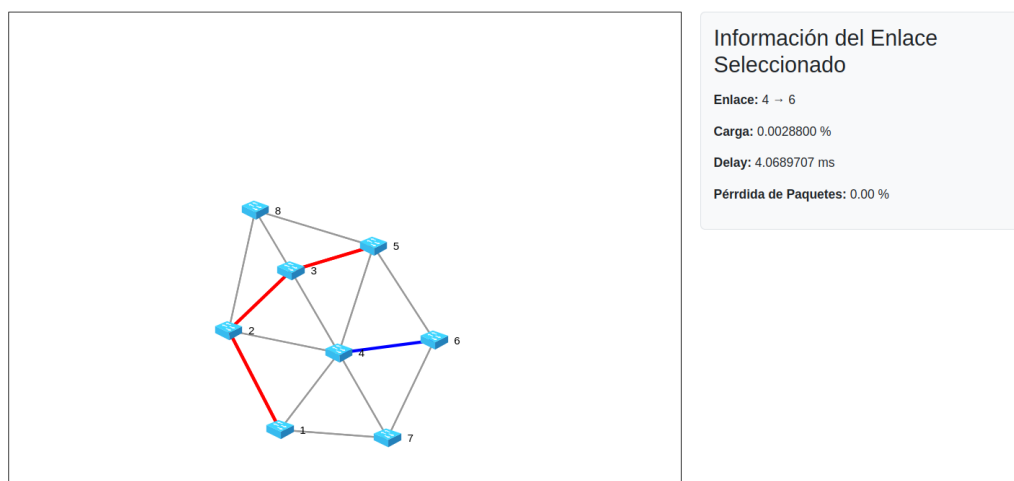


Figura 7.8: Topología y tabla

7.3. Potencial de Uso y Evaluación

Esta utilidad proporciona un entorno práctico y visual para evaluar y demostrar los beneficios del algoritmo propuesto en tiempo real, facilitando:

- La comprensión del comportamiento de los algoritmos bioinspirados.
- La observación del impacto de parámetros sobre la calidad del enrutamiento.
- El análisis en tiempo real del rendimiento y el equilibrio de carga en redes SDN.

Capítulo 8

Algoritmos Desarrollados

El algoritmo desarrollado en este trabajo toma como base la propuesta presentada en el artículo[4], donde se introduce LLBACO (Load and Link-based Ant Colony Optimization) como una estrategia de optimización bioinspirada para el balanceo de carga en redes definidas por software. En este TFG se ha implementado dicho algoritmo, pero además se ha mejorado integrando el enfoque del Ant Colony System (ACS), permitiendo introducir mecanismos adicionales como la actualización local de feromonas y decisiones más controladas mediante la estrategia de selección greedy. Estas modificaciones tienen como objetivo mejorar tanto la convergencia como la estabilidad del algoritmo bajo condiciones dinámicas de red.

Como paso previo a las mejoras planteadas, se implementó el algoritmo LLBACO tal y como fue descrito originalmente. A continuación, se detallan las principales fases clave de su funcionamiento.

8.1. Selección del siguiente nodo

En los algoritmos de colonia de hormigas, se utiliza el método de la “ruleta” para que las hormigas seleccionen el siguiente nodo. Como se puede ver en la figura 8.1 la probabilidad para que una hormiga en el nodo 1, seleccione el nodo 2 o el nodo 3, se representa como $P(1,2)$ y $P(1,3)$.

La ecuación para calcular la probabilidad $P(i,j)$, es decir la probabilidad de que una hormiga en el nodo ‘i’ seleccione el nodo ‘j’ viene dada por la siguiente fórmula:

$$P_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta} \cdot \mu_{ij}^{\gamma}}{\sum_{u \in \text{allowedSet}} \tau_{iu}^{\alpha} \cdot \eta_{iu}^{\beta} \cdot \mu_{iu}^{\gamma}} \quad \text{if } j \in \text{allowedSet} \quad (8.1)$$

Donde los parámetros α , β y γ representan la influencia relativa de cada factor, **allowedSet** son aquellos nodos que las hormigas pueden escoger.

- τ_{ij} representa la influencia de la intensidad de la feromona,

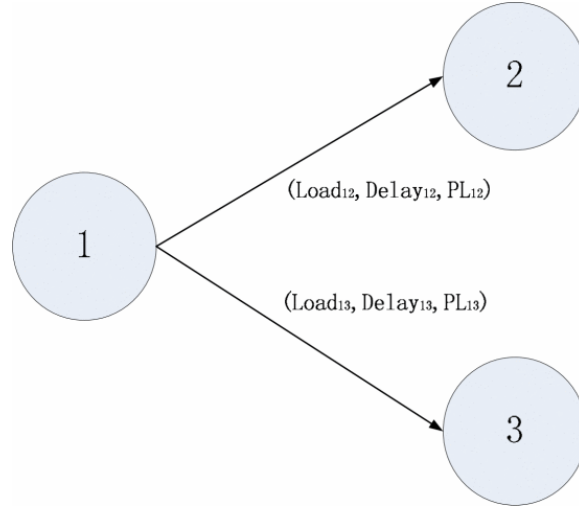


Figura 8.1: Selección de nodo. Fuente [4]

- $\eta_{ij} = \frac{1}{\text{Load}_{ij}}$ representa la influencia de la carga,
- $\mu_{ij} = \frac{1}{\text{Cost}_{ij}}$ representa la influencia del coste.

Todos son factores de impacto cuando las hormigas seleccionan el siguiente nodo.

8.2. Estrategia de actualización de feromonas

Una vez que la hormiga llega a su destino, deposita feromonas en cada enlace a lo largo de su camino. Sin embargo, estas feromonas se evaporan con el tiempo. Este proceso combinado de evaporación y depósito de feromonas es conocido como actualización de feromonas.

En LLBACO el método de actualización de feromonas es una actualización global. Esto significa que a diferencia de otros métodos donde cada hormiga deposita la feromona al completar su viaje, en este caso solo el mejor camino encontrado hasta el momento por todas las hormigas recibe un depósito de feromonas, esto ocurre después de que todas las hormigas han terminado su búsqueda en una iteración.

La actualización de feromonas se define a través de:

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \Delta\tau_{ij} \quad (8.2)$$

Donde:

$$\Delta\tau_{ij} = \begin{cases} \frac{Q}{L_{\text{best}}}, & \text{si } (i, j) \in \text{mejor camino} \\ 0, & \text{en otro caso} \end{cases} \quad (8.3)$$

Donde $\Delta\tau_{ij}$ es la cantidad de feromona depositada por una hormiga en el enlace entre el nodo i y j .

$\rho \in (0, 1)$ representa la velocidad de evaporación, Q es la cantidad total de feromonas y L la longitud del mejor camino.

La feromona es el medio por el cual las hormigas se comunican entre ellas, afectando directamente a la eficiencia de cada hormiga para buscar los caminos.

8.3. Cálculo de la carga y coste del camino

Como hemos hablado antes, $\eta_{ij} = \frac{1}{\text{Load}_{ij}}$ y $\mu_{ij} = \frac{1}{\text{Cost}_{ij}}$

Por lo tanto necesitamos saber cuál es la carga y el coste de cada camino individual encontrado por las hormigas.

- La carga del camino se define como la carga máxima de cualquiera de los enlaces que componen ese camino:

$$\text{Load}(\text{path}) = \max_{(i,j) \in \text{path}} \text{Load}_{ij} \quad (8.4)$$

- El coste del camino se define como la suma de los costes de todos los enlaces que componen ese camino

$$\text{Cost}(\text{path}) = \sum_{(i,j) \in \text{path}} \text{Cost}_{ij} \quad (8.5)$$

- El coste es calculado como:

$$\text{Cost}_{ij} = \text{Delay}_{ij} + \text{PacketLoss}_{ij} \quad (8.6)$$

Donde el Delay es el retardo del enlace y Packet Loss es la pérdida de paquetes del enlace.

8.4. Encontrar el mejor camino

LLBACO está diseñado para balancear la carga del enlace y asegurar la calidad de servicio (QoS) de la red. Así que, es seleccionado el mejor camino en función de la carga, delay y pérdida de paquetes. En una red compleja, un camino con poca carga es posible que tenga un alto delay y pérdida de paquetes. Para encontrar el mejor camino, el algoritmo establece un umbral dinámico que excluye esos caminos, en los que la carga excede dicho umbral. Entonces, el camino con coste mínimo es elegido de los caminos restantes.

8.4.1. Cálculo del umbral y actualización de la lista de caminos

El umbral consiste en la carga promedio de todos los caminos encontrados en esta iteración, Se calcula el umbral dinámico para esta iteración.

$$L_k = \frac{1}{|pathlist_k|} \sum_{path \in pathlist_k} Load(path) \quad (8.7)$$

Donde patchlist es el conjunto de caminos encontrados por las hormigas en la k-ésima iteración.

Una vez que hemos calculado el umbral (L_k), actualizamos la lista de caminos, eliminando de la lista original aquellos caminos cuya carga ($Load(path)$) sea mayor que el umbral.

Básicamente consiste en filtrar los caminos, que aunque pueden tener un coste total adecuado, que incluyen enlaces con una carga significativamente alta comparándola con el promedio de los caminos encontrados.

8.4.2. Encontrar el mejor camino

Finalmente, de la lista de caminos filtrados, se selecciona el mejor camino global para esta iteración.

El mejor camino es definido como el camino que tiene el mínimo coste entre todos los caminos que han sobrevivido al filtro de carga.

$$best_path = \arg \min_{path \in filteredPaths} Cost(path) \quad (8.8)$$

8.5. Descripción general del algoritmo

Algoritmo LLBACO (Load-aware ACS para Balanceo de Carga en SDN)

Entrada: Topología de red, métricas (coste, carga), nodo origen src , nodo destino dst

Salida: Ruta óptima P^* y su coste asociado C^*

Inicializar matriz de feromonas $\tau_{ij} \leftarrow \tau_0$

Calcular heurísticas: $\eta_{ij} \leftarrow 1/carga_{ij}$, $\mu_{ij} \leftarrow 1/coste_{ij}$

$P^* \leftarrow \emptyset$, $C^* \leftarrow \infty$

mientras iter < M **hacer**

ListaCaminos $\leftarrow \emptyset$

para cada hormiga $k \in \{1, \dots, K\}$ **hacer**

$P_k \leftarrow buscarCamino(src, dst, \tau, \eta, \mu)$

si P_k es válido **entonces**

Calcular C_k y L_k del camino P_k

Añadir (P_k, C_k, L_k) a ListaCaminos

```

    Calcular umbral de carga  $L^{(iter)} \leftarrow \text{media}([L_k])$ 
    Filtrar caminos con  $L_k \leq L^{(iter)}$  y seleccionar el de menor  $C_k$  como  $P^{(iter)}$ 
    si  $C^{(iter)} < C^*$  entonces
         $P^* \leftarrow P^{(iter)}$ 
         $C^* \leftarrow C^{(iter)}$ 
    Actualizar feromonas globales usando  $P^*$ 
retornar  $P^*, C^*$ 

```

8.6. Enfoque ACS

El algoritmo descrito en la sección anterior, es el algoritmo explicado en el artículo [4], el cual utiliza una implementación ACO simple. Sin embargo, existe una variante llamada Ant Colony System (ACS), la cual permite un mayor control sobre la exploración y explotación del espacio de soluciones.

8.6.1. Exploración frente a Explotación (q_0)

En cada paso, una hormiga decide si selecciona el siguiente nodo de forma determinista (enfoque greedy) o probabilística (“ruleta”), controlado por el parámetro q_0 . Para ello generamos un número aleatorio q entre 0 y 1:

- Si $q < q_0$, se aplica la estrategia de explotación (greedy), seleccionando el nodo que maximiza:

$$\operatorname{argmax}_{j \in \text{allowedSet}} \left(\tau_{ij}^\alpha \cdot \eta_{ij}^\beta \cdot \mu_{ij}^\gamma \right)$$

- En caso contrario $q > q_0$, será aplicada una estrategia de exploración basada en la probabilidad, donde el siguiente nodo es seleccionado utilizando el método de la ruleta, permitiendo encontrar nuevos caminos.

Esto permite que las hormigas prioricen caminos prometedores sin dejar de explorar otras posibilidades, evitando caer en óptimos locales,

8.6.2. Actualización Local de Feromonas (ϕ)

Cada vez que una hormiga recorre un enlace durante su exploración, va depositando feromonas sobre ese enlace, aplicando una *actualización local*. Reducen la atracción por otros caminos que ya han sido explorados, fomentando una mayor diversidad en la búsqueda.

$$\tau_{ij} \leftarrow (1 - \phi) \cdot \tau_{ij} + \phi \cdot \tau_0 \quad (8.9)$$

Donde:

- ϕ es la tasa de evaporación local ($\phi \in (0, 1)$).

- τ_0 es el valor inicial de la feromona.

Esta actualización se aplica en tiempo real a medida que la hormiga va recorriendo nodos, a diferencia de la actualización global, se ejecuta una vez finalizada la iteración y solamente al mejor camino.

8.6.3. Ventajas del enfoque ACS

El uso conjunto de actualización global y local de feromonas, junto con el control probabilístico de la exploración/explotación, ofrece un equilibrio entre:

- **Explotación:** concentrarse en las mejores soluciones actuales.
- **Exploración:** seguir generando diversidad y evitando estancarse.

Esto es crucialmente útil en entornos dinámicos, como es nuestro caso de las redes SDN, donde las condiciones cambian continuamente, siendo necesaria la adaptación constante del enrutamiento.

Capítulo 9

Experimentos

Una vez implementado nuestro algoritmo, se han realizado una serie de experimentos para validar su eficiencia. Se analizan diferentes configuraciones, modificando los parámetros del algoritmo para evaluar su impacto sobre el coste de las rutas, y la carga de la red y el delay.

También, se comparan los resultados obtenidos con otros enfoques clásicos de enrutamiento como Dijkstra y con la versión base de nuestro algoritmo (LLBACO), para determinar las mejoras introducidas por la variante LLBACS (que incorpora el enfoque ACS).

A continuación, se detallan los parámetros empleados y las configuraciones experimentales utilizadas para llevar a cabo dichas comparaciones.

9.1. Configuración

9.1.1. Parámetros

Tenemos como objetivo poder optimizar el balanceo de carga en redes SDN, ya que las redes hoy en día son altamente dinámicas y variables, será crítico tener en cuenta las métricas en tiempo real de la red. En concreto serán cruciales tres métricas de la red:

- **Load:** la carga de un enlace (Load_{ij}) se utiliza para calcular el factor heurístico η_{ij} , para la selección del siguiente nodo, donde $\eta_{ij} = \frac{1}{\text{Load}_{ij}}$. Básicamente significa que es más probable que las hormigas elijan enlaces con menor carga. La carga de un camino se define como la carga máxima de los enlaces entre todos los enlaces de ese camino.
- **Delay:** el retardo (delay_{ij}) es una componente del coste de transmisión de enlace (Cost_{ij}). El coste se calcula mediante la fórmula:

$$\text{Cost}_{ij} = \delta \cdot \text{delay}_{ij} + (1 - \delta) \cdot \text{packet_loss}_{ij}, \quad \text{donde } 0 < \delta < 1$$

Donde δ es un factor de ponderación. El coste de transmisión de un camino es la suma de todos los costes de los enlaces del camino.

- **Packet loss:** la pérdida de paquetes (pl_{ij}) en un enlace es el otro componente del coste de transmisión de un enlace ($Cost_{ij}$). Al igual que el delay contribuye al cálculo del costo total. La pérdida de paquetes es considerado como un factor de impacto en la selección de nodos para mejorar la calidad de servicio (QoS).

Para la realización del algoritmo también hay que tener en cuenta los siguientes parámetros clave:

Iteraciones : número total de ciclos o generaciones que ejecutará el ACO. En cada iteración, toda la colonia de hormigas busca caminos y las feromonas se actualizan en función del mejor camino encontrado. El algoritmo termina de ejecutar cuando llegue al número de iteraciones.

Colony size : número de hormigas que participan en el proceso de búsqueda en cada iteración. Cada hormiga busca independientemente un camino desde el origen hasta el destino.

α : representa la influencia de la intensidad de la feromona (τ_{ij}) cuando una hormiga selecciona el siguiente nodo. Un valor más alto de α significa que es más probable que las hormigas sigan caminos con una mayor concentración de feromonas (es decir, caminos que formaban parte de rutas anteriores exitosas o buenas).

β : representa la influencia de la carga ($\eta_{ij} = \frac{1}{Load_{ij}}$), cuando una hormiga selecciona el siguiente nodo. Como η_{ij} es inversamente proporcional a la carga, un valor más alto de β significa que las hormigas se sienten más atraídas por enlaces con una menor carga.

γ : representa la influencia del coste ($\mu_{ij} = \frac{1}{Cost_{ij}}$), donde

$$Cost_{ij} = \delta \cdot delay_{ij} + (1 - \delta) \cdot packet_loss_{ij}$$

Como μ_{ij} es inversamente proporcional al coste, un valor más alto de γ significa que las hormigas se sienten más atraídas por enlaces con menor coste.

ρ : este parámetro, con un valor entre 0 y 1, representa la velocidad de evaporación global de la feromona, es decir, la evaporación del mejor camino.

Q : representa la cantidad total de feromonas depositadas, En la estrategia global de actualización en LLBACO, la cantidad de feromona depositada en los enlaces del mejor camino centrado en una iteración se calcula como Q dividido por el coste de ese mejor camino.

q_0 : es el factor de exploración, el cual oscila entre 0 y 1, decidiendo cómo la hormiga selecciona su camino. Si un valor aleatorio $q \leq q_0$, la hormiga elige el mejor camino conocido (explotación). Si $q > q_0$, elige el camino de una forma probabilística. El objetivo de este parámetro es balancear la búsqueda de buenas soluciones con el descubrimiento de nuevas.

ϕ : este parámetro, con un valor entre 0 y 1, representa la velocidad de evaporación local de la feromona. Cada vez que una hormiga atraviesa un enlace, la feromona se reduce según este valor. Si es alto, dicha evaporación será mayor; si es más bajo, produce una evaporación más lenta. Este parámetro es muy importante para evitar la convergencia prematura a rutas subóptimas.

9.1.2. Configuraciones

En la sección anterior describimos cada parámetro necesario para nuestro algoritmo, a continuación se explican las distintas configuraciones usadas modificando dichos parámetros.

Para cada configuración se han realizado 30 ejecuciones sobre la misma instancia del problema, que consiste en una topología de red con 20 nodos y 27 enlaces como se muestra en la figura 9.1, con métricas de carga, delay y pérdida de paquetes. Esta instancia se ha seleccionado por tener la suficiente complejidad para demostrar que el algoritmo propuesto funciona correctamente, teniendo 20 nodos sin estar exactamente conectando unos con otros, podemos observar como en una situación de nodo X hacia nodo Y, salen diferentes caminos dependiendo de las métricas de la red.

Cada configuración ha sido evaluada realizando 30 ejecuciones independientes, siguiendo la regla ampliamente aceptada en estadística que establece que un tamaño de muestra de 30 es suficiente para obtener estimaciones fiables sobre la media y reducir la variabilidad en los resultados. [50].

En la tabla 9.1 se muestran los valores por defecto utilizados en la configuración base de nuestro algoritmo. Esta configuración servirá como referencia para evaluar el impacto en el rendimiento del algoritmo al modificarse los distintos parámetros.

A partir de la configuración base se han definido distintas variantes, donde en cada una se modifica un parámetro. La tabla 9.2 resume dichas configuraciones, indicando que parámetros hemos modificado y su nuevo valor.

Parámetro	Valor
Iteraciones	200
Tamaño de la colonia	20
α	1.0
β	1.0
γ	1.0
ρ	0.5
Q	1.0
high_cost (Coste alto)	1000
q_0	0.5
ϕ	0.5

Cuadro 9.1: Parámetros utilizados para la ejecución del algoritmo ACO

#	Configuración	Parámetro cambiado
1	Configuración base	—
2	Menos iteraciones	Iteraciones = 20
3	Más iteraciones	Iteraciones = 400
4	Colonia más pequeña	Tamaño colonia = 5
5	Colonia más grande	Tamaño colonia = 100
6	Alta influencia de feromona	$\alpha = 3.0$
7	Alta influencia de coste	$\beta = 3.0$
8	Alta influencia de la carga	$\gamma = 3.0$
9	Evaporación global mayor	$\rho = 0.95$
10	Evaporación global menor	$\rho = 0.05$
11	Menor exploración greedy	$q_0 = 0.9$
12	Mayor exploración greedy	$q_0 = 0.1$
13	Evaporación local mayor	$\phi = 0.95$
14	Evaporación local menor	$\phi = 0.05$

Cuadro 9.2: Resumen de configuraciones modificadas para evaluación

1. Configuración Base

El objetivo es establecer una línea base de rendimiento, servirá como punto de comparación para evaluar el impacto de las variaciones en otros parámetros.

2. Menos iteraciones

Iteraciones: 20

Tiene como objetivo evaluar la velocidad de convergencia del algoritmo y si un número reducido de iteraciones es suficiente para encontrar soluciones aceptables

3. Más iteraciones

Iteraciones: 400

Determinar si con un mayor número de iteraciones conduce a una mejora significativa de la solución, un coste menor del camino, o en la estabilidad, a costa de un mayor tiempo de cómputo.

4. Colonia más pequeña

Tamaño de la colonia: 5

Investigar el impacto de una menor exploración por iteración, esperando que una colonia menor reduzca el tiempo de cómputo significativamente en cada iteración, pero a su vez dificultando la búsqueda global o diversidad de caminos.

5. Colonia más grande

Tamaño de la colonia: 100

Al contrario que con la configuración anterior, queremos evaluar si una exploración mayor por iteración mejora la calidad de la solución o la velocidad de convergencia, y si el beneficio logrado compensa el mayor tiempo computacional por iteración.

6. Alta influencia de feromona

$\alpha = 3.0$

Entender cómo un mayor peso de feromona, es decir, una mayor tendencia a los caminos más visitados por las hormigas, afecta a la convergencia y la capacidad de exploración.

7. Alta influencia de coste

$\beta = 3.0$

Determinar el impacto provocado por dar mayor prioridad al coste (delay y pérdida de paquetes) en la decisión de la hormiga.

8. Alta influencia de la carga

$\gamma = 3.0$

Determinar el impacto provocado por dar mayor prioridad a la carga, para poder determinar si hay equilibrio entre una menor carga con enlaces con más delay o pérdida.

9. Evaporación global mayor

$$\rho = 0.95$$

Evaluar el efecto de una rápida “pérdida de memoria”, esperando que se fomente una mayor exploración de caminos y que no siempre predomine el camino con más feromonas concentradas.

10. Evaporación global menor

$$\rho = 0.05$$

Al contrario que al anterior configuración queremos comprobar si al permanecer más tiempo las feromonas en los enlaces conseguimos mejores resultados o nos quedamos atascados en las mismas soluciones.

11. Menor exploración greedy

$$q_0 = 0.9$$

Comprender cómo una mayor proporción de exploración aleatoria (menos decisiones greedy) afecta a la calidad de solución y velocidad de convergencia.

12. Mayor exploración greedy

$$q_0 = 0.1$$

Evaluar el impacto de una fuerte tendencia a la explotación (selección casi siempre greedy), esperando una convergencia más rápida pero con el riesgo de caer en óptimos locales.

13. Evaporación local mayor

$$\phi = 0.95$$

Entender cómo una evaporación local más agresiva impulsa la exploración previniendo estancarnos en los mismos caminos, evitando óptimos locales y produciendo una mayor diversidad de búsqueda.

14. Evaporación local menor

$$\phi = 0.05$$

Observar el efecto de “memoria” más larga de la feromona local, esperando que las hormigas persistan en caminos previamente visitados, reduciendo la búsqueda de nuevos caminos. El objetivo es si el algoritmo se estanque o es menos adaptable a los cambios de red.

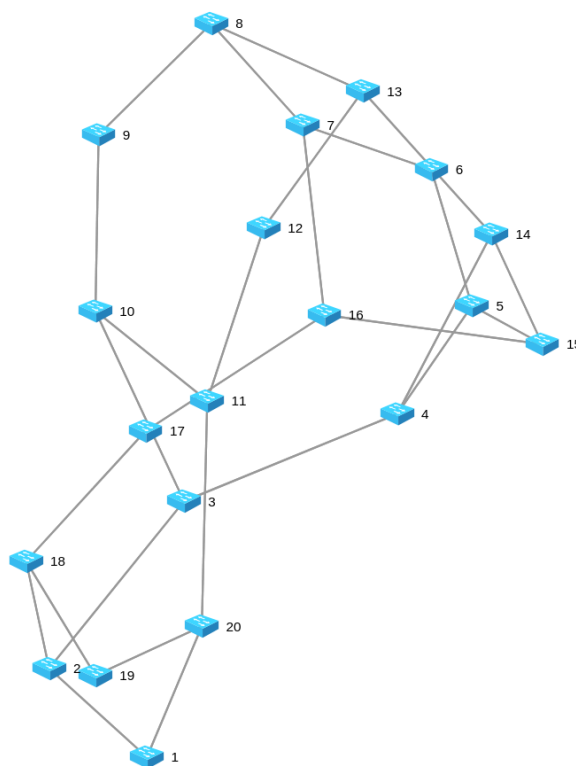


Figura 9.1: Visualización de la topología para las pruebas

9.1.3. Resultados de la comparativa de configuraciones

Como se ilustra en la gráfica 9.2 la configuración 10 es la que mejor coste promedio ha obtenido

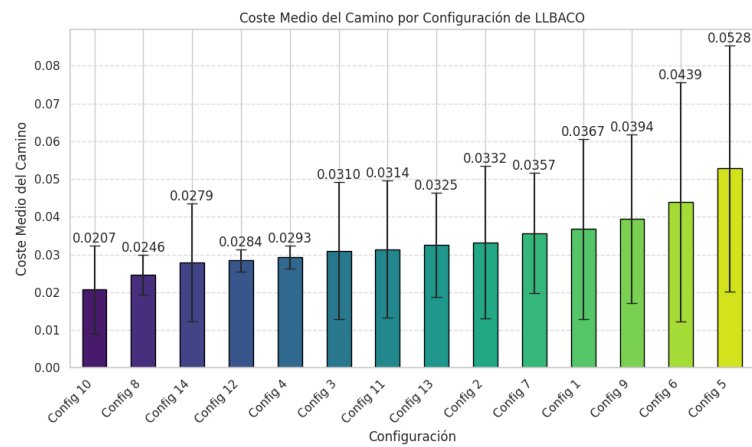


Figura 9.2: Costes promedio de cada configuración

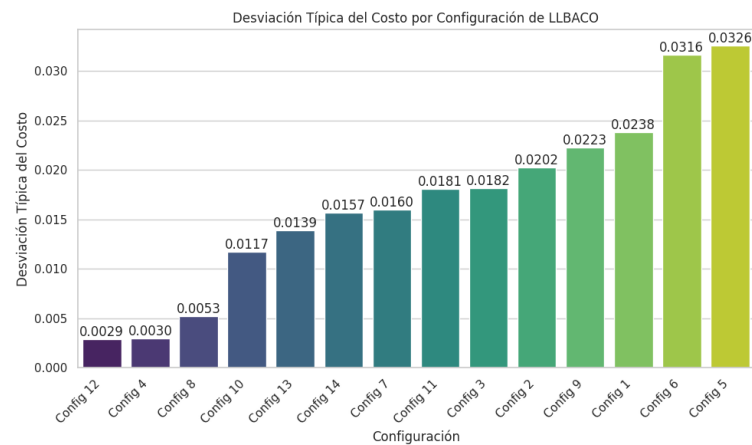


Figura 9.3: Desviación típica en los costes de cada configuración

En la figura 9.4 podemos observar el tiempo promedio de cada configuración siendo el más corto la configuración 2, la configuración con un **menor número de iteraciones**. Además, la configuración que más tiempo consume con mucha diferencia es la configuración 5, es la configuración con **mayor número de hormigas**.

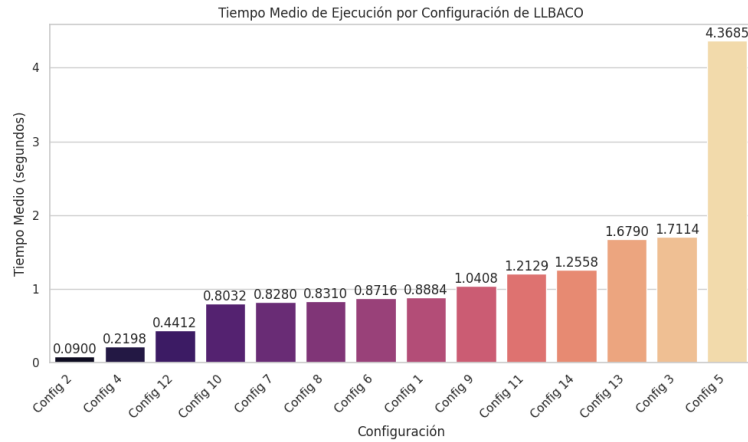


Figura 9.4: Tiempo promedio de cada configuración

9.2. Comparación de algoritmos

A lo largo de este tfg se ha hecho un especial énfasis en que los algoritmos bioinspirados, especialmente ACO, son mejores para el balanceo de carga en redes SDN que los algoritmos tradicionales como Dijkstra.

Por lo que vamos a comparar nuestra implementación con estos algoritmos tradicionales, para poder ver si realmente proporciona mejores resultados.

La topología de red utilizada para la comparación de diferentes algoritmos, es la misma que ha sido utilizada para la comparación de las configuraciones como se muestra en la figura 9.1. En la tabla 9.3 podemos observar los datos obtenidos de los diferentes algoritmos.

Cuadro 9.3: Comparación de Coste, Carga y Retardo Total por Algoritmo

Métrica	LLBACO	LLBACS	Dijkstra
Coste Total	12345.67	11340.89	13200.54
Carga Total	9876.54	9340.21	10100.77
Retardo Total	456.78	420.56	510.12

Como podemos observar en la gráfica 9.8, nuestra implementación LLBACS obtiene un menor coste que la implementación LLBACO, y que el algoritmo clásico Dijkstra. También podemos apreciar, mediante las líneas negras sobre cada barra, la **desviación estándar** del promedio de coste de ruta. Podemos concluir que Dijkstra no varía nada (ya que es un algoritmo determinista) lo cual resulta evidente en la gráfica 9.5, pero por otro lado tanto LLBACO como LLBACS son de naturaleza **estocástica**, lo que significa que incorporan elementos aleatorios en su funcionamiento.

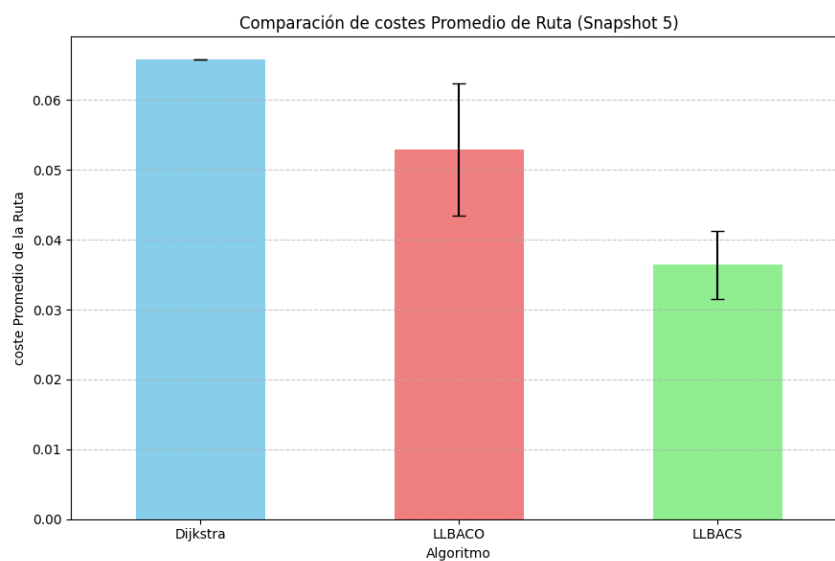


Figura 9.5: Coste promedio de cada algoritmo

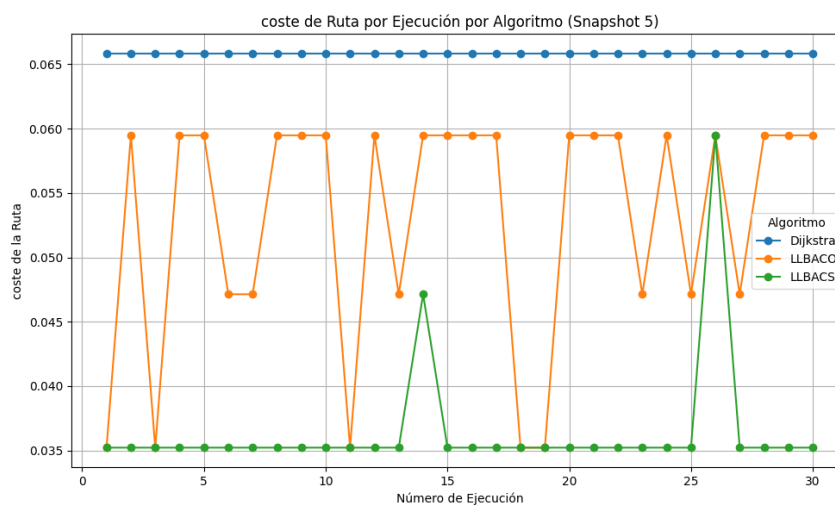


Figura 9.6: Coste de cada ejecución

Dado que, en una primera etapa, implementamos el algoritmo de Optimización de Colonia de hormigas **sin** la variante ACS, Como se aprecia en la gráfica de coste 9.5, la implementación con ACS muestra mejores resultados obteniendo menores costes. Además observando las gráficas 9.8 y 9.9, confirma que la variante ACS ofrece un mejor desempeño.

A la vista de los resultados obtenidos, podemos afirmar que la variante **LLBACS** es el algoritmo que mejor rendimiento ha demostrado en el con-

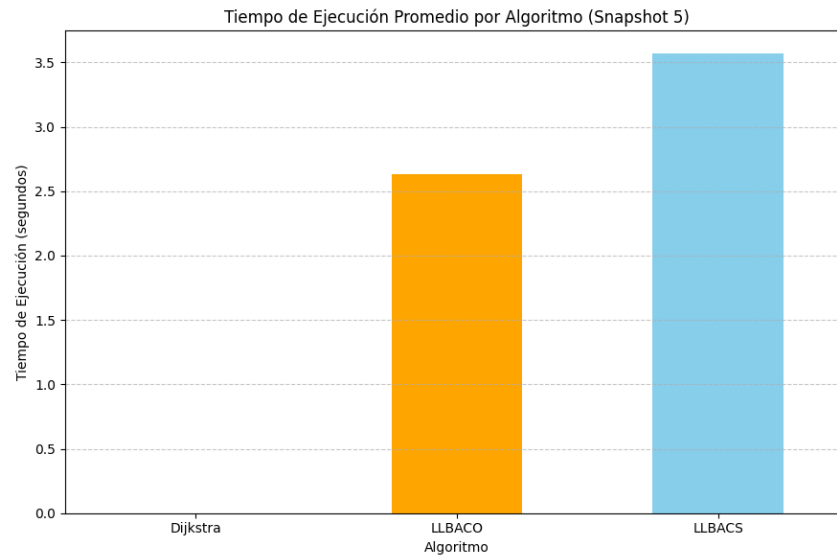


Figura 9.7: Tiempo promedio de cada algoritmo

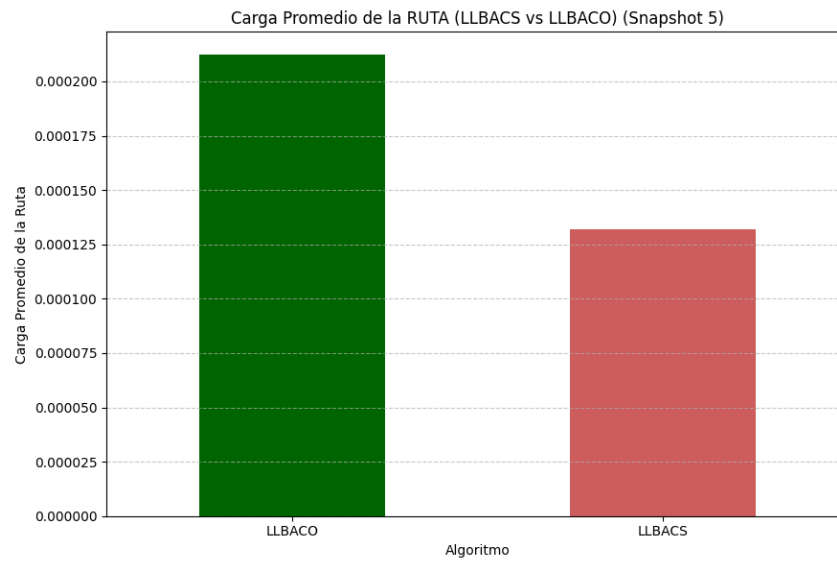


Figura 9.8: Carga promedio

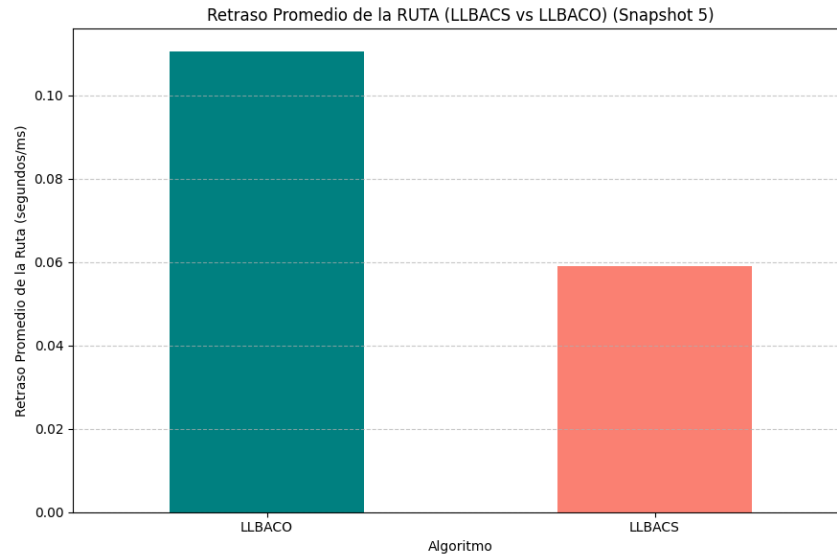


Figura 9.9: Delay promedio

junto de métricas analizadas. En comparación con el algoritmo **Dijkstra**, LLBACS logra seducir significativamente el coste promedio de las rutas como se muestra en la gráfica 9.5. Por su parte Dijkstra ofrece una solución más determinista, en la gráfica 9.5 se puede observar la nula variabilidad de resultados, lo cual es interesante en entornos donde se prioriza la rapidez sobre la adaptabilidad como observamos en la figura 9.7. Sin embargo, dicha ventaja la convierte al mismo tiempo inadecuada para redes dinámicas y con exigencias de calidad de servicio, como ocurre en entornos SDN.

Respecto a la versión base **LLBACO**, tras la incorporación de las técnicas del enfoque ACS, como la evaporación local y la estrategia de selección basada en el parámetro q_0 , permite una exploración más equilibrada, ofreciendo mejores resultados tanto en el coste promedio de las rutas, que es equivalente a lograr una carga promedio menor, pérdida de paquetes y delay de los enlaces, como se muestra en las gráficas 9.5, 9.8 y 9.9.

Finalmente, en base a los resultados obtenidos, **LLBACS** se posiciona como la mejor opción para el enrutamiento en redes SDN dentro del contexto experimental evaluado, al ofrecer un buen equilibrio entre eficiencia, adaptabilidad y calidad del servicio.

Capítulo 10

Conclusiones

Resumen del Trabajo Realizado

Este Trabajo Fin de grado ha abordado el problema del balanceo de carga en Redes Definidas por Software (SDN) mediante el enfoque del algoritmo Ant Colony Optimization, en concreto, su variante Ant Colony System (ACS), mediante el desarrollo de un sistema completo, que incluye:

- La implementación del algoritmo LLBACO adaptado al contexto de las redes, teniendo en cuenta múltiples métricas en tiempo real como la carga, el delay y la pérdida de paquetes.
- La integración del sistema en un entorno de red virtualizado mediante el uso de Mininet y el controlador Ryu.
- El desarrollo de una interfaz web interactiva con Flask y D3 que permita la visualización de la topología, modificar y observar parámetros en tiempo real.
- La comparación del rendimiento del algoritmo propuesto frente a algoritmos clásicos como Dijkstra, o la primera implementación de nuestro algoritmo ACO (sin la variante ACS) analizando métricas de eficiencia de red.

Objetivos Alcanzados

Los objetivos definidos al inicio del trabajo han sido cumplidos satisfactoriamente:

- Se ha implementado un algoritmo de optimización de colonia de hormigas, en concreto con su mejora ACS, adaptado a redes SDN.
- Se ha logrado diseñar el controlador en Ryu, que recoge métricas en tiempo real para la toma de decisión en el enrutamiento.

- Se ha demostrado, a través de experimentos, que el algoritmo propuesto mejora el balanceo de carga y reduce el coste respecto a soluciones tradicionales.
- Se ha conseguido desarrollar una herramienta visual que permite a los usuarios interactuar con la red y observar los resultados de forma intuitiva.

Líneas de Trabajo Futuro

Como se ha podido observar, los objetivos planteados para este proyecto se han cumplido plenamente, incluso con la implementación de una aplicación web ideal para que los usuarios puedan interactuar con la red. Sin embargo, esto no implica que la investigación sobre el problema del balanceo de carga en Redes Software esté terminada, ya que con este trabajo se abre la puerta a múltiples mejoras y ampliaciones:

- **Aprendizaje Automático:** Integrar técnicas de Machine Learning que permitan al sistema adaptarse mejor a patrones de tráfico históricos.
- **Topologías reales:** Validar el funcionamiento del sistema en redes físicas o escenarios más complejos y realistas.
- **Soporte multi-flujo:** Ampliar el algoritmo para manejar múltiples flujos concurrentes, analizando cómo se afectan mutuamente.
- **Interfaz avanzada:** Mejorar la interfaz de usuario para ofrecer dashboards más completos y opciones de configuración avanzada.

Reflexión Final

Este TFG ha permitido no solo aplicar conocimientos teóricos en el campo de las redes y los algoritmos bioinspirados, sino también desarrollar una solución práctica, extensible y fácilmente integrable. La experiencia obtenida durante su desarrollo pone de manifiesto el potencial de la combinación entre técnicas de optimización y tecnologías de red programables para resolver problemas complejos de gestión de tráfico en tiempo real.

Bibliografía

- [1] Huawei User. ¿qué es openflow? - conceptos básicos | programabilidad. <https://forum.huawei.com/enterprise/intl/es/thread/%C2%BFQu%C3%A9-es-OpenFlow-Conceptos-B%C3%A1sicos/748398411205459968?blogId=748398411205459968>, 2024.
- [2] Open Networking Foundation. Mininet - network emulator. <https://opennetworking.org/mininet/>, n.d.
- [3] Vishnu N. Comparison of software defined networking (sdn) controllers – part 5: Ryu. <https://aptira.com/comparison-of-software-defined-networking-sdn-controllers-part-5-ryu/>, July 2015.
- [4] Chunzhi Wang, Gang Zhang, Hui Xu, and Hongwei Chen. An aco-based link load-balancing algorithm in sdn. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. IEEE, 2016.
- [5] Ali Akbar Neghab, Nima Jafari Navimipour, Mehdi Hosseinzadeh, and Ali Rezaee. Nature-inspired meta-heuristic algorithms for solving the load balancing problem in the software-defined network. *International Journal of Communication Systems*, 31(18), 2018.
- [6] Hai Xue, Kyung Tae Kim, and Hee Yong Youn. Dynamic load balancing of software-defined networking based on genetic-ant colony optimization. *Sensors*, 19(2):311, 2019.
- [7] VmWARE. What is software-defined networking (sdn)? <https://www.vmware.com/topics/software-defined-networking>, 2024.
- [8] Qin Du, Xin Cui, Haoyao Tang, and Xiangxiao Chen. Review of load balancing mechanisms in sdn-based data centers. <https://www.scirp.org/journal/paperinformation?paperid=130469>, 2024.
- [9] Abdelaziz A., Fong A. T., Gani A., et al. Distributed controller clustering in software defined networks. *PloS One*, 12(4):e0174715, 2017.

- [10] Rego A., Canovas A., Jiménez J. M., and Lloret J. An intelligent system for video surveillance in iot environments. *IEEE Access*, 6:31580–31598, 2018.
- [11] Pashupati Baniya, Parma Nand, and Bharat Bhushan. Dynamic load balancing schemes for software-defined networking (sdn). In *Proceedings of Second International Conference on Intelligent System*. Springer, 2024.
- [12] Shao-Lun Chen, Yu-Yi Chen, and Shu-Hao Kuo. CLB: A novel load balancing architecture and algorithm for cloud services. *Computers & Electrical Engineering*, 58:154–160, 2017.
- [13] Amir Sina Milani and Nima Jafari Navimipour. Load balancing mechanisms and techniques in the cloud environments: Systematic literature review and future trends. *Journal of Network and Computer Applications*, 71:86–98, 2016.
- [14] Bruno A. A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turetli. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(1), 2014.
- [15] Víctor Berrocal. Bio-inspired optimization algorithms to efficiently solve optimization problems in diverse application areas. <https://www.oga.ai/en/blog/bio-inspired-optimization-algorithms/>, 2022.
- [16] Weiwei Jiang, Haoyu Han, Yang Zhang, Ji'an Wang, Miao He, Weixi Gu, Jianbin Mu, and Xirong Cheng. Graph neural networks for routing optimization: Challenges and opportunities. *Sustainability*, 16(21), 2024.
- [17] Jehn-Ruey Jiang¹, Widhi Yahya^{1,2}, and Mahardeka Tri Ananta^{1,2}. Load balancing and multicasting using the extended dijkstra's algorithm in software defined networking. <https://staff.csie.ncu.edu.tw/jrjiang/publication/2014ICS-SDN-CemeraReady.pdf>, 2014.
- [18] Jehn-Ruey Jiang, Hsin-Wen Huang, Ji-Hau Liao, and Szu-Yuan Chen. Extending dijkstra's shortest path algorithm for software defined networking. <https://staff.csie.ncu.edu.tw/jrjiang/publication/Dijkstra-SDN%280526%29.pdf>, 2014. Department of Computer Science and Information Engineering, National Central University, Taiwan.
- [19] VmWARE. Round robin load balancing definition. <https://www.vmware.com/topics/round-robin-load-balancing>, 2024.
- [20] MW Team. Why are we still using round-robin load balancers? <https://middleware.io/blog/round-robin-load-balancers/>, 2023.

- [21] Sukhveer Kaur, Japinder Singh, Krishan Saluja, and Navtej Ghumman. Round-robin based load balancing in software defined networking. In *2nd International Conference on "Computing for Sustainable Global Development"*, 03 2015.
- [22] Mohammed Jaber Alam, Ritesh Chugh, Salahuddin Azad, and Md Rahat Hossain. Ant colony optimization-based solution to optimize load balancing and throughput for 5g and beyond heterogeneous networks. *EURASIP Journal on Wireless Communications and Networking*, 2024(44), 2024.
- [23] Jerry Sheehan. Top benefits of software-defined networking (sdn). <https://synchronet.net/benefits-of-software-defined-networking/>, 2024.
- [24] Jingbo Li, Li Ma, Yingxun Fu, Dongchao Ma, and Ailing Xiao. *Load Balancing in Heterogeneous Network with SDN: A Survey*, pages 250–261. Communications in Computer and Information Science, 01 2021.
- [25] Murat Karakus and Arjan Durresi. A survey: Control plane scalability issues and approaches in software-defined networking (sdn). *Computer Networks*, 112:279–293, 2017.
- [26] Murat Karakus and Arjan Durresi. Economic viability of software defined networking (sdn). *Computer Networks*, 133:100–109, 2018.
- [27] Sushma Sathyanarayana and Melody Moh. Joint route-server load balancing in software defined networks using ant colony optimization. *2017 International Conference on Computing, Networking and Communications (ICNC)*, pages 338–342, 2017.
- [28] IBM. ¿qué son las redes definidas por software (sdn)? <https://www.ibm.com/es-es/topics/sdn>, 2024.
- [29] Li-Der Chou, Yu-Ting Yang, Yu-Min Hong, Jui-Kang Hu, and Bor-Shenn Jean. A genetic-based load balancing algorithm in openflow network. In James J. Park, Hai Jin, and Hojjat Adeli Jeong, editors, *Advanced Technologies, Embedded and Multimedia for Human-centric Computing*, pages 411–417. Springer, 2014.
- [30] Open Networking Foundation. Sdn architecture. Technical report, June 2014.
- [31] Christian Blum. Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews*, 2(4):353–373, 2005.
- [32] Wikipedia contributors. Ant colony optimization algorithms — wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms, 2024.

- [33] Antonio Mora. Optimización basada en colonias de hormigas. conceptos principales. <https://es.slideshare.net/Slidemora/optimizacin-basada-en-colonias-de-hormigas>, November 2011.
- [34] Marco Dorigo. Ant colony optimization. *Scholarpedia*, 2(3):1461, 2007. Revision #90969. Curator: Marco Dorigo.
- [35] Andrew Compton and Ian Rogers. Ant colony optimization: The traveling salesman problem. In *Swarm Intelligence: From Natural to Artificial Systems*, chapter 2.3. Oxford University Press, 2006.
- [36] Thomas Stützle and Marco Dorigo. Aco algorithms for the traveling salesman problem. *IRIDIA, Université Libre de Bruxelles*, 2000.
- [37] Yeiler Alberto Quintero Barco. Sdn: El futuro de las comunicaciones. *Ingente Americana*, 2(2), 2022. Fundación Universitaria María Cano, Colombia.
- [38] LACNOG. Tutorial: Aplicando sdn, nfv, devops y cloud en latinoamérica. <https://www.lacnic.net/innovaportal/file/2639/1/tutorial-aplicando-sdn-sdn-devops-y-cloud-en-latinoamerica.pdf>, 2017. LACNOG 2017, Montevideo.
- [39] Nile Secure. What is network utilization & how to monitor it. <https://nilesecure.com/network-management/network-utilization>, 2024.
- [40] Amazon Web Services. ¿qué es la latencia de red? <https://aws.amazon.com/es/what-is/latency/>, 2024.
- [41] Proofpoint. ¿qué es packet loss o pérdida de paquetes? <https://www.proofpoint.com/es/threat-reference/packet-loss>, 2024.
- [42] WuCai Lin and LiChen Zhang. The load balancing research of sdn based on ant colony algorithm with job classification. *2016 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, pages 1–4, 2016.
- [43] Mininet. Mininet overview. <https://mininet.org/overview/>, 2022.
- [44] Ichiro Ide, Katsuyoshi Ofuji, Akira Kato, and Tatsuya Tanaka. Ryu: Component-based software defined networking framework. <https://www.ntt-review.jp/archive/ntttechnical.php?contents=ntr201408fa4.html>, August 2014.
- [45] LearnPython.dev. Basic flask. <https://www.learnpython.dev/03-intermediate-python/80-web-frameworks/basic-flask/>, 2024.
- [46] Flask Documentation. Api. <https://flask.palletsprojects.com/en/stable/api/>.

- [47] D3. The javascript library for bespoke data visualization. <https://d3js.org/>.
- [48] Caroline Scharf. Flask graph visualization. <https://blog.tomsawyer.com/flask-graph-visualization>, January 2025. Publicado en el blog de Tom Sawyer Software.
- [49] iPerf - the ultimate speed test tool for TCP, UDP and SCTP. <https://iperf.fr/>, 2017.
- [50] Leanscape. The importance of identifying the right sample size for business improvement. <https://leanscape.io/the-importance-of-identifying-the-right-sample-size-for-business-improvement>, 2023. Consultado en junio de 2025.
- [51] Murat Karakus and Arjan Durresi. Economic analysis of software defined networking (sdn) under various network failure scenarios. Technical report, Indiana University Purdue University Indianapolis, 2017. Technical Report, Department of Computer and Information Science.
- [52] Lin Li and Qiaozhi Xu. Load balancing strategies in software-defined networking: A survey. *2017 7th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC)*, pages 364–367, 2017.
- [53] Evans Tetteh Owusu, Kwame Agyemang-Prempeh Agyekum, Marinah Benneh, Pius Ayorna, Justice Owusu Agyemang, George Nii Martey Colley, and James Dzisi Gadze. A transformer-based deep q learning approach for dynamic load balancing in software-defined networks. *arXiv preprint arXiv:2501.12829*, 2025. Disponible en arXiv desde enero de 2025.
- [54] Ahmed Abdelaziz, Ang Tan Fong, and Abdullah Gani. Distributed controller clustering in software defined networks. *PLoS ONE*, 12(4):e0174715, 2017.
- [55] A. Rego, A. Canovas, J. M. Jiménez, and J. Lloret. An intelligent system for video surveillance in iot environments. *IEEE Access*, 6:31580–31598, 2018.

Apéndice A

Manual de usuario

A.1. Introducción

El proyecto creado para la optimización del balanceo de carga por medio de algoritmos bioinspirados, requiere una serie de pasos e instrucciones para poder lanzar correctamente la aplicación web y poder interactuar con la red SDN.

A fin de ejecutar correctamente la aplicación, es necesario seguir una serie de pasos y disponer de un entorno previamente configurado. Todos los archivos necesarios para su implementación y uso se encuentran disponibles en el siguiente repositorio de GitHub: <https://github.com/lmchaves/tfg>. En este repositorio se encuentra disponible todo el código fuente desarrollado.

A.2. Requisitos y herramientas necesarias

Antes de descargar los archivos necesarios, es imprescindible contar con Mininet y el controlador Ryu instalados en el sistema.

Existen diversas formas de realizar la instalación. En este caso, se ha seguido un procedimiento sencillo y práctico, explicado en el siguiente video: <https://www.youtube.com/watch?v=q4wsx4u5juU&list=LL&index=5>

A.3. Guía de uso de la aplicación

Clonar el repositorio

Abre una terminal y ejecuta los siguientes comandos:

- `git clone https://github.com/lmchaves/tfg`
- `cd tfg`

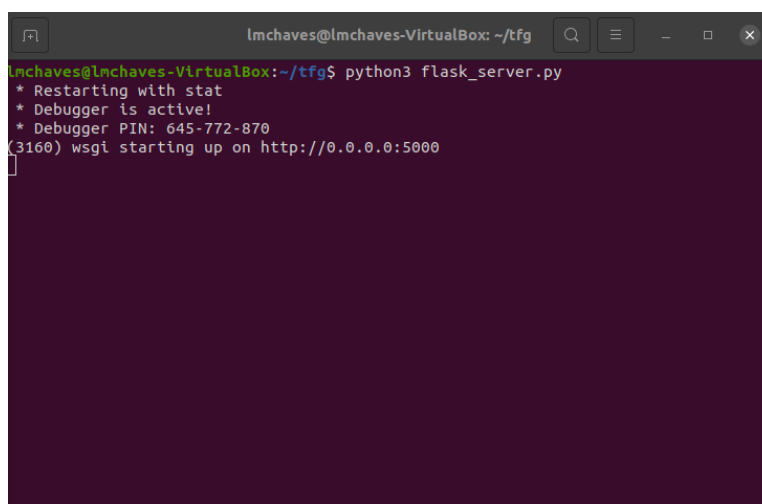
Una vez clonado el repositorio, nos situamos en el directorio y procedemos a abrir tres terminales independientes.

Ejecutar la aplicación web

En la primera terminal ejecutamos el siguiente comando:

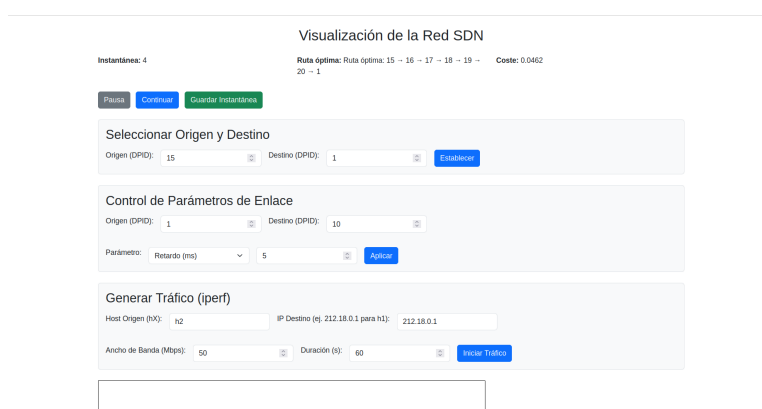
- `python3 flask_server.py`

Esto iniciará el servidor web y nos mostrará la dirección de la aplicación, como se observa en la Figura A.1. Basta con copiar dicha dirección en el navegador para visualizar la interfaz general de la aplicación (Figura A.2).



```
lmchaves@lmchaves-VirtualBox: ~/tfg$ python3 flask_server.py
* Restarting with stat
* Debugger is active!
* Debugger PIN: 645-772-870
(3160) wsgi starting up on http://0.0.0.0:5000
```

Figura A.1: Lanzamiento de la aplicación



Visualización de la Red SDN

Instantáneas: 4 Ruta óptima: Ruta óptima: 15 - 16 - 17 - 18 - 19 - 20 - 1 Coste: 0.0462

[Pausa](#) [Continuar](#) [Gestionar Instantáneas](#)

Seleccionar Origen y Destino

Origen (DPID): 15 Destino (DPID): 1 [Establecer](#)

Control de Parámetros de Enlace

Origen (DPID): 1 Destino (DPID): 10

Parámetro: Retardo (ms) 5 [Aplicar](#)

Generar Tráfico (iperf)

Host Origen (hX): h2 IP Destino (ej. 212.18.0.1 para h1): 212.18.0.1

Ancho de Banda (Mbps): 50 Duración (s): 60 [Iniciar Tráfico](#)

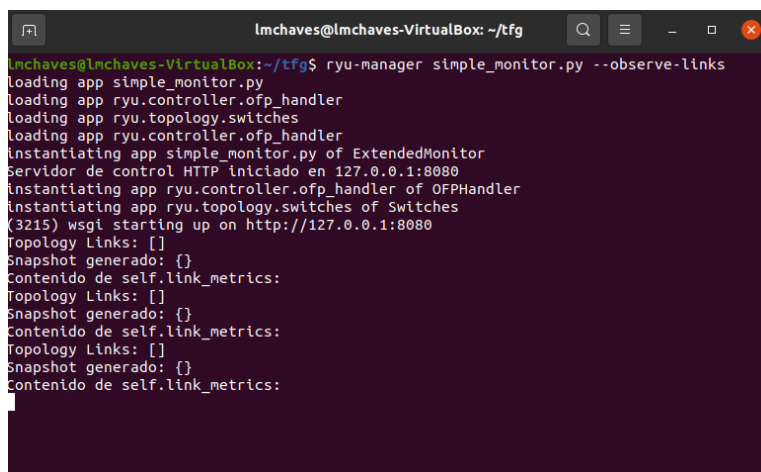
Figura A.2: Visión de la aplicación

Iniciar el controlador Ryu

En la segunda terminal, iniciamos el controlador de Ryu mediante el siguiente comando:

- `ryu_manager simple_monitor.py --observe-links`

Esto activará el controlador como se muestra en la figura A.3.



```
lmchaves@lmchaves-VirtualBox: ~/tfg
lmchaves@lmchaves-VirtualBox:~/tfg$ ryu_manager simple_monitor.py --observe-links
loading app simple_monitor.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
Instantiating app simple_monitor.py of ExtendedMonitor
Servidor de control HTTP iniciado en 127.0.0.1:8080
Instantiating app ryu.controller.ofp_handler of OFPHandler
Instantiating app ryu.topology.switches of Switches
(3215) wsgi starting up on http://127.0.0.1:8080
Topology Links: []
Snapshot generado: {}
Contenido de self.link_metrics:
Topology Links: []
Snapshot generado: {}
Contenido de self.link_metrics:
Topology Links: []
Snapshot generado: {}
Contenido de self.link_metrics:
```

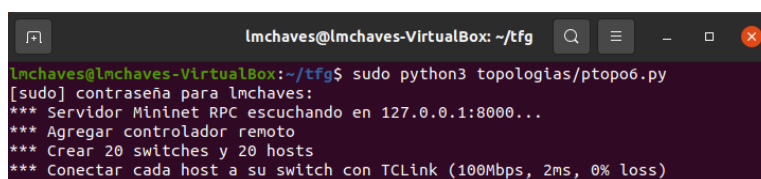
Figura A.3: Inicio del controlador

Lanzar la topología en Mininet

Finalmente en la tercera terminal, lanzamos la topología utilizando permisos de superusuario:

- `sudo python3 topologias/ptopo6.py`

Si el lanzamiento es correcto, se mostrará como en la Figura A.4.



```
lmchaves@lmchaves-VirtualBox: ~/tfg
lmchaves@lmchaves-VirtualBox:~/tfg$ sudo python3 topologias/ptopo6.py
[sudo] contraseña para lmchaves:
*** Servidor Mininet RPC escuchando en 127.0.0.1:8000...
*** Agregar controlador remoto
*** Crear 20 switches y 20 hosts
*** Conectar cada host a su switch con TCLink (100Mbps, 2ms, 0% loss)
```

Figura A.4: Ejecución de la topología

A.4. Modificar parámetros del algoritmo

Si se desea modificar los parámetros del algoritmo `n`, como el número de iteraciones, el tamaño de la colonia o los pesos de cada heurística, es necesario editar el archivo `simple_monitor.py`.

Dentro del método:

```
def run_llbaco(self, snapshot):
    ...
    best_path, best_cost = llbaco_aux.run_aco_llbaco(
        nodes, cost_matrix, load_matrix,
        self.src_node_dpid, self.dst_node_dpid,
        iterations=200,
        colony_size=100,
        alpha=1.0,
        beta=1.0,
        gamma=1.0,
        rho=0.5,
        Q=1.0,
        high_cost=1000,
        q0=0.5,
        phi=0.1
    )
```

Listing A.1: Llamada al algoritmo LLBACO con configuración personalizada

Cada uno de estos parámetros controla una parte del comportamiento del algoritmo:

- **iterations**: Número de iteraciones del algoritmo.
- **colony_size**: Número de hormigas por iteración.
- **alpha**, **beta**, **gamma**: Pesos de las heurísticas (feromona, carga, coste). Normalmente 1, si quieres ver el efecto de cada uno aumenta hasta 3.
- **rho**: Velocidad de evaporación global. Siempre tiene que tener un valor entre 0 y 1.
- **Q**: Cantidad total de feromonas depositadas.
- **high_cost**: Coste penalizador para caminos no válidos.
- **q0**: Factor de explotación frente a exploración. El valor tiene que converger siempre entre 0 y 1.
- **phi**: Evaporación local (tras cada paso de la hormiga). Al igual que la evaporación global el valor siempre tiene que estar entre 0 y 1.

