

Laboratorio 3 - Segunda parte
Algoritmos de Enrutamiento

Stefano Alberto Aragoni Maldonado (20261)
Carol Andreé Arévalo Estrada (20461)
Fátima Priscilla González Sandoval (20689)

1. Introducción

En esta segunda parte de la práctica se implementaron diferentes algoritmos de enrutamiento en una red simulada sobre el protocolo XMPP. Más específicamente, se implementó *Dijkstra Link State*, *Flooding*, y *Distance Vector Routing* con el propósito de simular el proceso de envío de mensajes mediante routers de una red.

Una fase crucial de este laboratorio consistió en investigar detalladamente el funcionamiento de cada uno de los algoritmos. Este proceso permitió adquirir un conocimiento sólido sobre cómo cada algoritmo aborda el desafío de determinar las rutas más eficientes para dirigir los mensajes a sus destinos finales. Este conocimiento previo fue esencial para llevar a cabo la implementación de los algoritmos utilizando el lenguaje de programación Python. Para finalizar, se realizó una serie de pruebas con el propósito de evaluar el funcionamiento correcto de estos algoritmos.

En conclusión, esta práctica proporcionó una comprensión profunda de los algoritmos de enrutamiento y su aplicación en las redes de comunicación. Asimismo, nos permitió comprender la importancia de estas en nuestro día a día.

2. Objetivos

- a. Conocer los algoritmos de enrutamiento utilizados en las implementaciones actuales de Internet.
- b. Implementar los algoritmos de enrutamiento y probarlos.
- c. Analizar el funcionamiento de los algoritmos de enrutamiento.
- d. Implementar los algoritmos de enrutamiento en una red simulada sobre el protocolo XMPP.

3. Descripción

En esta fase del laboratorio, se utilizó el protocolo XMPP para simular una red con la cual se pudo evaluar los algoritmos de enrutamiento desarrollados en la primera fase del laboratorio. Esto se realizó con el propósito de poder analizar cómo estos algoritmos se comportan en un entorno simulado. Cabe destacar que la implementación de esta red permitió que los nodos de la red interactuaran entre sí de forma automática, lo que proporcionó una valiosa oportunidad para comprender cómo estos algoritmos funcionan en implementaciones actuales de Internet.

Los algoritmos de enrutamiento funcionan sobre nodos interconectados entre sí, donde cada nodo conoce únicamente cuáles son los vecinos que tiene. A través de *Dijkstra Link State* y *Distance Vector Routing*, cada nodo puede construir una tabla de enrutamiento que le permite determinar la mejor ruta para reenviar los paquetes hacia su destino. En el caso de Flooding, sin embargo, cada nodo envía los paquetes a sus vecinos (exceptuando al nodo vecino que le envió el mensaje o

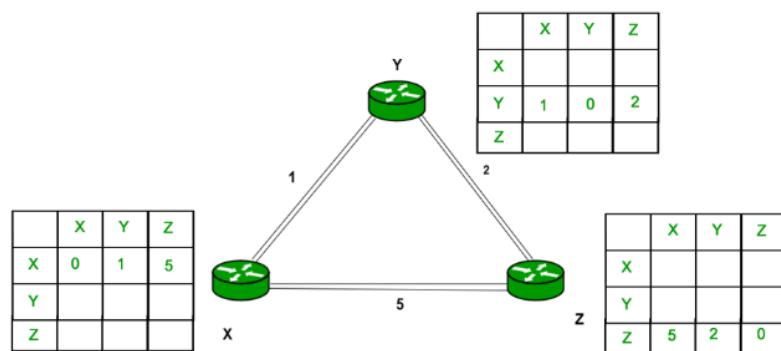
cualquier nodo que ya recibió el mensaje anteriormente). Esto permite que los mensajes lleguen –eventualmente– a su destino.

Estos algoritmos son fundamentales en las redes de comunicación, ya que permiten que un mensaje llegue de forma eficiente a su destino. A continuación se describe más a profundidad cada algoritmo implementado.

Distance Vector Routing

Es un algoritmo de enrutamiento para determinar las rutas óptimas para transmitir datos entre dispositivos en una red. Cada nodo en la red mantiene una tabla de enrutamiento que contiene información sobre el costo y la ruta más corta hacia cada posible destino. Este algoritmo opera a través de iteraciones, donde en cada iteración, los nodos actualizan sus tablas de enrutamiento basándose en la información obtenida de sus vecinos. Cabe destacar que dichas actualizaciones se realizan utilizando la ecuación de Bellman-Ford. El proceso continúa hasta que todas las tablas de enrutamiento convergen y se encuentran las rutas óptimas (GeeksForGeeks, 2017).

Figura 1. Tabla de enrutamiento inicial de Distance Vector Routing



Aunque este algoritmo es considerado un enfoque sencillo de implementar y puede ser adecuado para redes pequeñas y poco complejas, tiene ciertas limitaciones (GeeksForGeeks, 2017):

1. Counting to infinity: Este problema ocurre cuando hay un enlace roto en la red. Consiste en que un router hace broadcast una ruta a través de otro router que, a su vez, hace broadcast la misma ruta de regreso al router original. Esto crea un ciclo de broadcast donde la distancia a través de la ruta rota sigue incrementándose gradualmente hasta llegar a un valor muy grande (como el infinito).
2. Lenta convergencia en redes grandes: En redes grandes, la convergencia en el enrutamiento por vector de distancia puede ser lenta debido a la forma en que funciona el algoritmo.

Para el presente laboratorio, en base a la teoría anteriormente descrita, se logró implementar el algoritmo de *Distance Vector Routing* en Python de la siguiente manera.

1. Se inicia en una terminal una instancia del programa. El programa pregunta qué nodo representa dicha instancia. Cabe destacar que cada nodo es representado por una terminal diferente.
2. La terminal se conecta con el servidor “*alumchat.xyz*” e ingresa sesión con las credenciales correspondientes al nodo.
3. Todos los nodos comparten entre sí sus vectores de distancia a través de mensajes de XMPP. Al recibir algún vector de un vecino, si este genera alguna actualización se comparte con todos los vecinos el vector actualizado. Si no genera actualización, no se hace nada.
4. Se repite el paso 3 hasta que ya no haya más cambios y los vectores se estabilicen. A partir de aquí se puede empezar el proceso de enviar mensajes.
5. Si desean enviar un mensaje, primero preguntan a qué nodo se dirige el mensaje y cuál es el mensaje.
 - a. Utilizando los vectores y los saltos de enlace, se puede predecir a qué nodo se debe ir después para llegar al destino.
6. Cuando un nodo recibe un paquete, este revisa si es o no el destinatario final.
 - a. En caso de no ser el destinatario, determina a cuál nodo se debe enviar el paquete para que llegue de la forma más rápida al destinatario final.
 - b. De ser el destinatario final, muestra en la terminal el mensaje correspondiente.

Dijkstra Link State

Al igual que *Distance Vector Routing*, en *Dijkstra Link State* se busca determinar las rutas óptimas para transmitir mensajes y paquetes en una red de nodos interconectados. Inicialmente, cada nodo cuenta con una tabla que únicamente contiene información sobre el costo de los enlaces a los nodos vecinos. De forma iterativa, a través de un proceso de broadcast (*flooding*), el algoritmo *Dijkstra Link State* busca que los nodos conozcan todos los costos de los enlaces de la red. Para esto, los nodos comparten su información con sus vecinos. Cuando un nodo recibe la tabla de un vecino, actualiza su propia tabla con la nueva información y vuelve a compartir su tabla con sus vecinos. Este proceso de intercambio y actualización de tablas continúa hasta que todos los nodos en la red tengan conocimiento completo de los costos de enlace hacia todos los demás nodos (Javapoint, 2021).

Cuando todos los nodos conocen todos los costos de enlace de la red, cada uno puede calcular la ruta más corta hacia cualquier otro nodo en la red utilizando el algoritmo de Dijkstra (Javapoint, 2021).

Figura 2. Algoritmo de Dijkstra para encontrar camino más corto

```
Initialization
N = {A}      // A is a root node.
for all nodes v
if v adjacent to A
then D(v) = c(A,v)
else D(v) = infinity
loop
find w not in N such that D(w) is a minimum.
Add w to N
Update D(v) for all v adjacent to w and not in N:
D(v) = min(D(v) , D(w) + c(w,v))
Until all nodes in N
```

Cabe destacar que *Dijkstra Link State* es un algoritmo eficiente para determinar rutas óptimas. Sin embargo, debido a su funcionamiento tiene varias limitaciones y desventajas (Haider, 2021):

1. Recursos: Dado a que cada nodo tiene una tabla de enrutamiento conformada por los costos de los enlaces de toda la red, cada uno requiere más memoria. Asimismo, ya que cada nodo es responsable de calcular el camino más corto a cada nodo, también es necesario que estos tengan suficiente capacidad de procesamiento para realizar estos cálculos.
2. Ineficiencia: Dado a que se comparte la información de costos de enlace repetidamente (*broadcast*), se puede generar una gran cantidad de tráfico en la red.

Para el presente laboratorio, en base a la teoría anteriormente descrita, se logró implementar el algoritmo de *Dijkstra Link State* en Python de la siguiente manera.

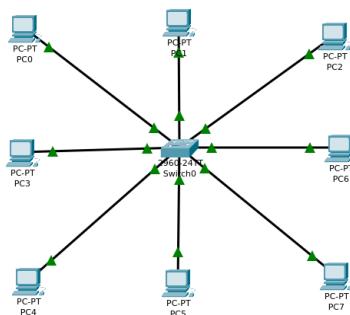
1. Se inicia en una terminal una instancia del programa. El programa pregunta qué nodo representa dicha instancia. Cabe destacar que cada nodo es representado por una terminal diferente.
2. La terminal se conecta con el servidor “*alumchat.xyz*” e ingresa sesión con las credenciales correspondientes al nodo.
3. Cada nodo hace un “echo” a los nodos vecinos a través de un mensaje de XMPP. Los nodos vecinos reciben el “echo” y le responden
4. Cada nodo posteriormente le comparte a sus nodos vecinos (que hayan respondido el “echo”) su tabla con costos de enlace. Esta tabla está vacía, exceptuando la fila que corresponde al nodo. Esta indica los costos de enlace de este nodo a sus vecinos.
5. Posteriormente, los vecinos envían su tabla actualizada a todos sus demás vecinos.

6. Se repite el paso 4 y 5 hasta que todos los nodos tengan su tabla llena con los costos de enlace de toda la red. Cuando la tabla está llena, con el algoritmo de Dijkstra se determinan las rutas más cortas a todos los nodos.
7. Finalmente, con esto listo, los nodos son capaces de enviar mensajes por XMPP. Si desean enviar un mensaje, primero preguntan a qué nodo se dirige el mensaje y cuál es el mensaje.
 - a. Utilizando la tabla de enrutamiento generada con el algoritmo de Dijkstra, los nodos determinan cuál es la ruta más corta para transmitir el mensaje.
8. Cuando un nodo recibe un paquete, este revisa si es o no el destinatario final.
 - a. En caso de no ser el destinatario, utiliza el resultado del algoritmo de Dijkstra para determinar a qué nodo enviar el paquete para que llegue de la forma más rápida al destinatario final.
 - b. De ser el destinatario final, muestra en la terminal el mensaje correspondiente.

Flooding

Este es un algoritmo de enrutamiento que consiste en reenviar todos los paquetes entrantes a todos los vecinos; exceptuando al vecino del cual se recibió el paquete, y cualquier otro vecino que haya recibido el mensaje anteriormente. De esta manera, se asegura que el paquete llegue a su destino eventualmente, ya que se está enviando a todas las direcciones posibles (Hanna, 2021).

Figura 3. Representación visual de Flooding



Aunque el enrutamiento por inundación garantiza la entrega de paquetes, tiene varias limitaciones y desventajas. Debido a estas limitaciones, el enrutamiento por inundación se utiliza raramente en redes modernas y eficientes (Tutorialspoint, s.f.). A continuación se presentan algunas desventajas:

1. Ineficiencia: Dado que los paquetes se envían a todos los vecinos (exceptuando al vecino que envió el mensaje, y otros que ya hayan recibido el mismo), esto puede generar una gran cantidad de tráfico en la red.
2. Duplicación de paquetes: Debido a que se envían copias del paquete por todos los nodos, puede haber duplicación de paquetes en la red.

3. Falta de una topología: Este enfoque no tiene en cuenta la topología de la red ni la disponibilidad de rutas más eficientes.
4. Seguridad y privacidad: El enrutamiento por inundación puede exponer información a dispositivos no deseados.

Para el presente laboratorio, en base a la teoría anteriormente descrita, se logró implementar el algoritmo de *Flooding* en Python de la siguiente manera.

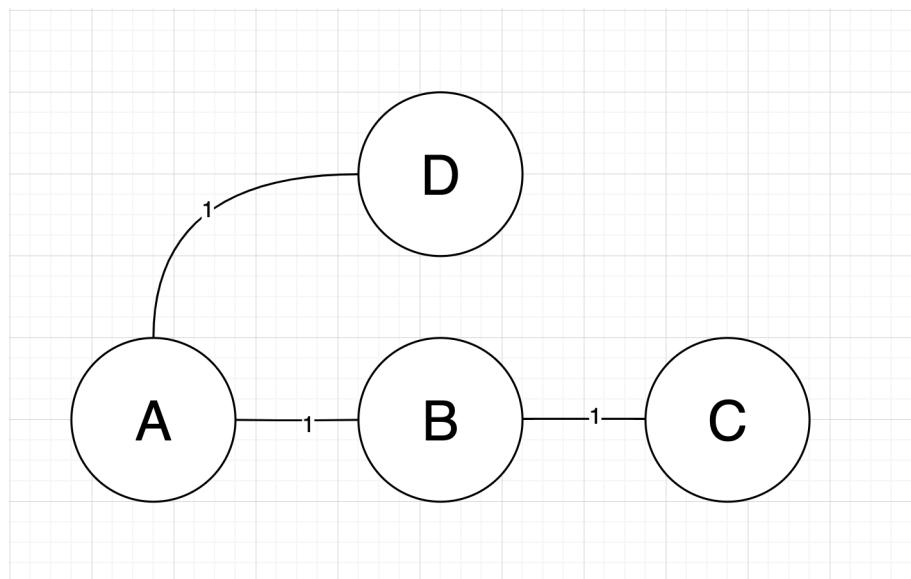
1. Se inicia en una terminal una instancia del programa. El programa pregunta qué nodo representa dicha instancia. Cabe destacar que cada nodo es representado por una terminal diferente.
2. La terminal se conecta con el servidor “*alumchat.xyz*” e ingresa sesión con las credenciales correspondientes al nodo.
3. Si el nodo quiere enviar un mensaje, se le pregunta el mensaje y el destinatario.
 - a. Posteriormente, se crea un paquete de tipo “message” al que se le agrega como *visitado* el nodo origen.
4. El programa posteriormente envía el paquete generado –a través de XMPP– a sus nodos vecinos (representadas por instancias del programa) que no han sido anteriormente visitados.
5. Cuando los nodos reciben el paquete, revisan si son o no el destinatario final. Asimismo, se revisa si el mensaje ya había sido enviado o recibido por el nodo; en caso que sí, se descarta.
 - a. En caso de no ser el destinatario final, se agrega a la lista de *visitado*. Posteriormente, se envía el paquete a los nodos vecinos que no han sido visitados anteriormente mediante el servidor XMPP.
 - b. De ser el destinatario final, muestra en la terminal el mensaje correspondiente.

4. Resultados / Pruebas

En esta segunda parte del laboratorio, como se mencionó anteriormente, se implementaron los algoritmos de enrutamiento en una red simulada sobre el protocolo XMPP. En la primera fase del laboratorio, se tenía que copiar y pegar los outputs de las terminales en otras terminales con el propósito de poder compartir los diferentes paquetes (mensajes, información y “echo”). Sin embargo, en esta segunda parte se usó XMPP para que los diferentes nodos se pudieran comunicar de manera más eficiente y automatizada. De tal manera permitiendo una interacción más fluida y dinámica entre los nodos de la red, similar a como funciona en la vida real.

Cabe destacar que se realizó una serie de pruebas con el propósito de verificar la integración correcta de los algoritmos de enrutamiento (Dijkstra Link State, Flooding y Distance Vector Routing) con el servidor “alumchat.xyz”. Más específicamente, se quiso evaluar que los diferentes nodos se pudieran comunicar de forma correcta a través de la red simulada. Durante estas pruebas, se revisó que no hubiera fallas en la transmisión de datos, que los mensajes llegaran a su destino, y que no se presentaran ciclos infinitos. Para la realización de estas pruebas, se utilizó la siguiente red compuesta de cuatro nodos (*figura 4*). A continuación se presentan los resultados.

Figura 4. Diseño de red a utilizar en pruebas



Distance Vector Routing

Para el algoritmo de Distance Vector Routing, se inicializaron cuatro terminales representando cada uno de los nodos que se requerían para la topología. Cada terminal se conecta con el servidor “alumchat.xyz” con las credenciales correspondientes.

Figura 5. Inicialización de nodos (Distance Vector)

```

lab3_redes - Python dv.py - 39x24
stefanoaragoni@stefanos-MacBook-Pro lab 3_redes % python3 dv.py
Using slower stringprep, consider compiling the faster cython/libidn one.

---DISTANCE VECTOR---
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 1

lab3_redes - Python dv.py - 39x24
stefanoaragoni@stefanos-MacBook-Pro lab 3_redes % python3 dv.py
Using slower stringprep, consider compiling the faster cython/libidn one.

---DISTANCE VECTOR---
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 2

lab3_redes - Python dv.py - 39x24
stefanoaragoni@stefanos-MacBook-Pro lab 3_redes % python3 dv.py
Using slower stringprep, consider compiling the faster cython/libidn one.

---DISTANCE VECTOR---
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 3

lab3_redes - Python dv.py - 38x24
stefanoaragoni@stefanos-MacBook-Pro lab 3_redes % python3 dv.py
Using slower stringprep, consider compiling the faster cython/libidn one.

---DISTANCE VECTOR---
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 4

```

Inicialmente, cada nodo tiene su vector de distancia inicial. Cada uno le comparte a sus vecinos –de forma repetitiva– sus vectores de distancia. Cuando un nodo recibe un vector de distancia, utiliza la ecuación Bellman-Ford para actualizar su vector. En caso haya una actualización, el nodo le comparte a sus vecinos su tabla actualizada. Este proceso se repite hasta que ya no haya actualizaciones.

Figura 6. Transmisión de Vectores de Distancia

```

lab3_redes - Python dv.py - 39x24
VECTOR DE DISTANCIA INICIAL: [0, 1, 9999, 1]
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a B...
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a A...
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a C...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de D...
--> No se han realizado cambios en la tabla de enrutamiento. Envio tabla de regreso.
--> Envio tabla a D...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de C...
--> No se han realizado cambios en la tabla de enrutamiento. Envio tabla de regreso.
--> Envio tabla a C...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de B...
--> Se han realizado cambios en la tabla de enrutamiento.
--> Envio tabla a B...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de A...
--> Se han realizado cambios en la tabla de enrutamiento.
--> Envio tabla a A...

lab3_redes - Python dv.py - 39x24
VECTOR DE DISTANCIA INICIAL: [1, 0, 1, 9999]
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a A...
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a C...
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a B...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de C...
--> No se han realizado cambios en la tabla de enrutamiento. Envio tabla de regreso.
--> Envio tabla a C...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de B...
--> Se han realizado cambios en la tabla de enrutamiento.
--> Envio tabla a B...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de A...
--> Se han realizado cambios en la tabla de enrutamiento.
--> Envio tabla a A...

lab3_redes - Python dv.py - 39x24
VECTOR DE DISTANCIA INICIAL: [9999, 1, 0, 9999]
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a B...
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a A...
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a C...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de B...
--> Se han realizado cambios en la tabla de enrutamiento.
--> Envio tabla a B...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de A...
--> Se han realizado cambios en la tabla de enrutamiento.
--> Envio tabla a A...

lab3_redes - Python dv.py - 38x24
VECTOR DE DISTANCIA INICIAL: [1, 9999, 9999, 0]
NOTIFICACION: COMPARTIENDO VECTOR DISTANCIA ----> Envio tabla a A...
MENSAJES / NOTIFICACIONES --> Vector de distancia recibido de A...
--> Se han realizado cambios en la tabla de enrutamiento.
--> Envio tabla a A...

```

Después de haber convergido, se puede observar el vector final así como los enlaces para llegar a los otros nodos con la menor cantidad de pasos. Con esto listo, se puede empezar a transmitir mensajes.

Figura 7. Vectores de Distancia Finales

```

lab3_redes - Python dv.py - 39x24
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingresar el número de la opción deseada: 1
--> Entrada inválida. Por favor, ingrese un número entero.
Ingresar el número de la opción deseada: 1
----- VECTOR DISTANCIA -----
[0, 1, 2, 1]
----- ENLACES -----
[' ', 'B', ' ', 'D']

lab3_redes - Python dv.py - 39x24
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingresar el número de la opción deseada: 1
--> Entrada inválida. Por favor, ingrese un número entero.
Ingresar el número de la opción deseada: 1
----- VECTOR DISTANCIA -----
[1, 0, 1, 2]
----- ENLACES -----
['A', ' ', 'C', 'A']

lab3_redes - Python dv.py - 39x24
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingresar el número de la opción deseada: 1
--> Entrada inválida. Por favor, ingrese un número entero.
Ingresar el número de la opción deseada: 1
----- VECTOR DISTANCIA -----
[2, 1, 0, 3]
----- ENLACES -----
['B', ' ', ' ', 'B']

lab3_redes - Python dv.py - 38x24
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingresar el número de la opción deseada: 1
----- VECTOR DISTANCIA -----
[1, 2, 3, 0]
----- ENLACES -----
['A', ' ', 'A', ' ']

```

Al querer enviar un mensaje, se solicita el destinatario así como el mensaje a enviar. A partir de esto, el nodo detecta a qué nodo se debe enviar el paquete para que llegue a su destino en la menor cantidad de pasos. Como se puede observar en la figura 8, enviar un mensaje del nodo A al nodo D se hace de forma directa ya que son nodos vecinos.

Figura 8. Transmisión de mensaje directo (Distance Vector)

Sin embargo, para enviar un mensaje del nodo D al nodo C, se debe de pasar por A y B primero. Como se puede observar en la *figura 9*, el algoritmo de *Distance Vector* fue capaz de primero reenviar el mensaje de D a A, de A a B, y de B a C. De esta manera, C logró recibir correctamente el mensaje destinado a él.

Figura 9. Re-transmisión de mensajes (Distance Vector)

```
● ● ● lab3_redes - Python dv.py - 39x24
--> Camino más corto a través de: D
--- Mensaje enviado a D, con destino a D.
-----
----- MENÚ DE COMUNICACIÓN -----
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingrese el número de la opción deseada: 1

----- MENSAGES / NOTIFICACIONES --
----- MENSAJE -----
--> Camino más corto a través de: B
--> Mensaje enviado a A, con destino a C.

----- MENÚ DE COMUNICACIÓN -----
[1, 0, 1, 2]
----- ENLACES -----
['A', ' ', 'C', 'A']

----- MENÚ DE COMUNICACIÓN -----
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingrese el número de la opción deseada: 1

----- MENSAGES / NOTIFICACIONES --
----- MENSAJE -----
--> Camino más corto a través de: C
--> Mensaje enviado a A, con destino a C.

----- MENSAJE -----
--> delta@lumchat.xyz ha enviado un mensaje para retransmitir a B, con destino a C.

----- MENÚ DE COMUNICACIÓN -----
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingrese el número de la opción deseada: 1

● ● ● lab3_redes - Python dv.py - 39x24
----- VECTOR DISTANCIA -----
[2, 1, 0, 3]
----- ENLACES -----
['B', 'B', ' ', 'B']

----- MENÚ DE COMUNICACIÓN -----
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingrese el número de la opción deseada: 1

----- MENSAGES / NOTIFICACIONES --
----- MENSAJE -----
--> delta@lumchat.xyz ha enviado un mensaje: Hola Charlie! Del Nodo D

----- MENSAJE -----
--> delta@lumchat.xyz ha enviado un mensaje: Hola Charlie! Del Nodo D

----- MENÚ DE COMUNICACIÓN -----
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingrese el número de la opción deseada: 1

● ● ● lab3_redes - Python dv.py - 38x24
nsaje: Hola! Nodo D
-----
2
----- ENVIRAR MENSAJE A USUARIO -----
Selecciónne un nodo de la lista:
1. alfa@lumchat.xyz
2. betta@lumchat.xyz
3. charlie@lumchat.xyz
4. delta@lumchat.xyz
Ingresé el número del nodo: 3
Mensaje: Hola Charlie! Del Nodo D
--> Camino más corto a través de: A
--> Mensaje enviado a A, con destino a C.
-----
----- MENÚ DE COMUNICACIÓN -----
1) Revisar vector distancia
2) Enviar mensaje
3) Salir
Ingrese el número de la opción deseada: 1
```

Dijkstra Link State

En el caso de *Dijkstra Link State*, como primer paso se inicializaron las terminales representativas de los nodos. Cada una se conecta con el servidor “alumchat.xyz” con sus credenciales correspondientes. Cabe destacar que cada nodo conoce a sus vecinos así como los costos de enlace hacia los mismos.

Figura 10. Inicialización de nodos (Dijkstra Link State)

```
● ● ● lab3_redes - Python dijkstra.py - 39x24 ● ● ● lab3_redes - Python dijkstra.py - 39x24 ● ● ● lab3_redes - Python dijkstra.py - 39x24 ● ● ● lab3_redes - Python dijkstra.py - 39x24
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 dijkstra.py
Using slower stringprep, consider compiling the faster cython/libidn one.
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 dijkstra.py
Using slower stringprep, consider compiling the faster cython/libidn one.
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 dijkstra.py
Using slower stringprep, consider compiling the faster cython/libidn one.
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 dijkstra.py
Using slower stringprep, consider compiling the faster cython/libidn one.

---DIJKSTRA---
Selezionne un nodo de la lista:
1. alfa@lumchat.xyz
2. betta@lumchat.xyz
3. charlie@lumchat.xyz
4. delta@lumchat.xyz

Ingrese el número del nodo: 1

---DIJKSTRA---
Selezionne un nodo de la lista:
1. alfa@lumchat.xyz
2. betta@lumchat.xyz
3. charlie@lumchat.xyz
4. delta@lumchat.xyz

Ingrese el número del nodo: 2

---DIJKSTRA---
Selezionne un nodo de la lista:
1. alfa@lumchat.xyz
2. betta@lumchat.xyz
3. charlie@lumchat.xyz
4. delta@lumchat.xyz

Ingrese el número del nodo: 3

---DIJKSTRA---
Selezionne un nodo de la lista:
1. alfa@lumchat.xyz
2. betta@lumchat.xyz
3. charlie@lumchat.xyz
4. delta@lumchat.xyz

Ingrese el número del nodo: 4
```

Posteriormente, cada nodo se encarga de mandarle un paquete tipo “echo” a sus vecinos. Esto con el propósito de verificar que estén activos y presentes. Posteriormente, cada nodo le comparte su tabla a los nodos que hayan hecho “echo” de regreso. Es importante mencionar que esta tabla está inicialmente vacía.

exceptuando la fila que corresponde al nodo. Esta indica los costos de enlace de este nodo a sus vecinos. Cuando los otros nodos reciben una tabla, estos actualizan su propia tabla y comparten la versión actualizada con sus vecinos activos (que hayan hecho “echo”). En caso de no haber una actualización, no comparte la tabla nuevamente. Esto se repite hasta que todos los nodos conozcan los costos de enlace de toda la red.

Figura 11. Echo y broadcast de información

Cuando ya se conocen todos los costos de enlace de la red, se debe utilizar el algoritmo de Dijkstra para calcular las rutas más cortas utilizando la información disponible. Esto actualiza la tabla de enrutamiento de cada nodo. Con esto listo, se puede empezar a enviar y recibir mensajes.

Figura 12. Cálculo de camino más corto (Dijkstra Link State)

Al querer enviar un mensaje, se solicita el destinatario así como el mensaje a enviar. A partir de esto, se determina cuál es el camino más corto a través del algoritmo de Dijkstra. Por ejemplo, como se puede observar en la *figura 13*, enviar un mensaje del nodo A al nodo D se hace de forma directa ya que son nodos vecinos.

Figura 13. Transmisión de mensaje directo (Dijkstra Link State)

```
lab3_redes - Python dijkstra.py - 39x24 lab3_redes - Python dijkstra.py - 39x24 lab3_redes - Python dijkstra.py - 39x24 lab3_redes - Python dijkstra.py - 38...
Ingrese el número de la opción deseada: [0, 1, 9999, 1], [1, 0, 1, 9999], [999 9, 1, 0, 9999], [1, 9999, 9999, 0]] [[0, 1, 9999, 1], [1, 0, 1, 9999], [999 9, 1, 0, 9999], [1, 9999, 9999, 0]] 4) Salir
Ingrese el número de la opción deseada : 2
----- DIJKSTRA -----
TABLA ACTUAL: [[0, 1, 9999, 1], [1, 0, 1, 9999], [999 9, 1, 0, 9999], [1, 9999, 9999, 0]]
----- MENÚ DE COMUNICACIÓN -----
1) Revisar tabla enrutamiento
2) Calcular rutas más cortas (Dijkstra)
3) Enviar mensaje
4) Salir
Ingresel número de la opción deseada: 2
----- DIJKSTRA -----
TABLA ACTUAL: [[0, 1, 9999, 1], [1, 0, 1, 9999], [999 9, 1, 0, 9999], [1, 9999, 9999, 0]] 4) Salir
Ingresel número de la opción deseada : 2
----- DIJKSTRA -----
TABLA ACTUAL: [[0, 1, 9999, 1], [1, 0, 1, 9999], [999 9, 1, 0, 9999], [1, 9999, 9999, 0]] 4) Salir
Ingresel número de la opción deseada : 2
----- MENÚ DE COMUNICACIÓN -----
1) Revisar tabla enrutamiento
2) Calcular rutas más cortas (Dijkstra)
3) Enviar mensaje
4) Salir
Ingresel número de la opción deseada : 2
----- MENÚ DE COMUNICACIÓN -----
1) Revisar tabla enrutamiento
2) Calcular rutas más cortas (Dijkstra)
3) Enviar mensaje
4) Salir
Ingresel número de la opción deseada : 2
----- MENAJE -----
--> alfa@lumchat.xyz ha enviado un me
nsaje:Hola Nodo D, de A
```

Sin embargo, para enviar un mensaje del nodo C al nodo D, se debe de pasar por B y A primero. Como se puede observar en la *figura 14*, el algoritmo de *Dijkstra Link State* fue capaz de calcular desde el nodo C la ruta completa a tomar para que el mensaje llegue a su destino. Posteriormente, se reenvió el mensaje del nodo C a B, de B a A, y de A a D. De esta manera, D logró recibir correctamente el mensaje destinado a él.

Figura 14. Re-transmisión de mensajes (Dijkstra Link State)

```
lab3_redes - Python dijkstra.py - 39x24 lab3_redes - Python dijkstra.py - 39x24 lab3_redes - Python dijkstra.py - 39x24
D.
9999, 0]]
----- MENÚ DE COMUNICACIÓN -----
1) Revisar tabla enrutamiento
2) Calcular rutas más cortas (Dijkstra)
3) Enviar mensaje
4) Salir
Ingrese el número de la opción deseada: 3

----- MENÚ DE COMUNICACIÓN -----
1) Revisar tabla enrutamiento
2) Calcular rutas más cortas (Dijkstra)
3) Enviar mensaje
4) Salir
Ingrese el número de la opción deseada: 3

----- ENVIAR MENSAJE A USUARIO -----
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo de: 4
Mensaje: Hola Charlie, de D
--> Mensaje enviado a C, con destino a C.
----- MENÚ DE COMUNICACIÓN -----
1) Revisar tabla enrutamiento
2) Calcular rutas más cortas (Dijkstra)
3) Enviar mensaje
4) Salir
Ingrese el número de la opción deseada: 4

----- MENSAJE -----
--> delta@alumchat.xyz ha enviado un mensaje para retransmitir a B, con destino a C.

----- MENSAJE -----
--> alfa@alumchat.xyz ha enviado un mensaje para retransmitir a C, con destino a C.

----- MENSAJE -----
--> betta@alumchat.xyz ha enviado un mensaje para retransmitir a D, con destino a D.

----- MENSAJE -----
--> charlie@alumchat.xyz ha enviado un mensaje para retransmitir a A, con destino a D.

----- MENÚ DE COMUNICACIÓN -----
1) Revisar tabla enrutamiento
2) Calcular rutas más cortas (Dijkstra)
3) Enviar mensaje
4) Salir
Ingrese el número de la opción deseada: 4
----- MENSAJE -----
--> charlie@alumchat.xyz ha enviado un mensaje: Hola Delta, de regreso
--> Camino más corto: ['D', 'A', 'B', 'C']
--> Mensaje enviado a A, con destino a C.
----- MENÚ DE COMUNICACIÓN -----
1) Revisar tabla enrutamiento
2) Calcular rutas más cortas (Dijkstra)
3) Enviar mensaje
4) Salir
Ingrese el número de la opción deseada: 4
----- MENSAJE -----
--> charlie@alumchat.xyz ha enviado un mensaje: Hola Delta, de regreso
```

Flooding

A diferencia de los algoritmos anteriormente descritos, *Flooding* se caracteriza por no calcular el camino más corto de un nodo a otro. Sino, en este algoritmo los nodos envían cualquier paquete entrante a todos los nodos vecinos directos; exceptuando a cualquier vecino que haya recibido el mensaje anteriormente. De esta manera, se asegura que el paquete llegue a su destino eventualmente, ya que se está enviando a todas las direcciones posibles.

Por tal razón, como primer paso, se inicializaron las terminales representativas de los nodos. En este caso, nuevamente, cada nodo conoce a sus vecinos y se conecta con el servidor “alumchat.xyz” con sus credenciales correspondientes.

Figura 15. Inicialización de nodos (Flooding)

```

lab3_redes - Python flooding.py - 39x...
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
Using slower stringprep, consider compiling the faster cython/libidn one.

---FLOODING---
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 1

----- MENSAJES / NOTIFICACIONES -----
----- MENÚ DE COMUNICACIÓN -----
1) Enviar mensaje
2) Salir
Ingrese el número de la opción deseada: 1

lab3_redes - Python flooding.py - 39x...
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
Using slower stringprep, consider compiling the faster cython/libidn one.

---FLOODING---
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 2

----- MENSAJES / NOTIFICACIONES -----
----- MENÚ DE COMUNICACIÓN -----
1) Enviar mensaje
2) Salir
Ingrese el número de la opción deseada: 1

lab3_redes - Python flooding.py - 39x...
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
Using slower stringprep, consider compiling the faster cython/libidn one.

---FLOODING---
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 3

----- MENSAJES / NOTIFICACIONES -----
----- MENÚ DE COMUNICACIÓN -----
1) Enviar mensaje
2) Salir
Ingrese el número de la opción deseada: 1

lab3_redes - Python flooding.py - 38x...
stefanoaragoni@Stefanos-MacBook-Pro lab3_redes % python3 flooding.py
Using slower stringprep, consider compiling the faster cython/libidn one.

---FLOODING---
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 4

----- MENSAJES / NOTIFICACIONES -----
----- MENÚ DE COMUNICACIÓN -----
1) Enviar mensaje
2) Salir
Ingrese el número de la opción deseada: 1

```

Debido al funcionamiento del algoritmo, este no requiere que se comparten las tablas de enrutamiento entre los nodos. Como resultado, directamente permite enviar mensajes a cualquier nodo, así como recibir mensajes. En la siguiente figura se muestra cómo enviar un mensaje del nodo A al nodo C.

Como primer paso, el nodo A pregunta a quién se le desea enviar el mensaje, así como el mensaje. El nodo A posteriormente le envía el paquete con el mensaje a sus vecinos (nodo B y D). El nodo D recibe el paquete y detecta que no puede reenviar el paquete a nadie más. Por otro lado, el nodo B envía el paquete recibido a C (su vecino directo). El nodo C recibe el paquete y detecta que el mensaje era destinado para él, por lo cual lo muestra en pantalla.

Figura 16. Transmisión de mensaje (Flooding)

```

lab3_redes - Python flooding.py - 39x...
ling the faster cython/libidn one.

----- MENÚ DE COMUNICACIÓN -----
1) Enviar mensaje
2) Salir
Ingrese el número de la opción deseada: 1

----- ENVIAR MENSAJE A USUARIO -----
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 2

----- MENSAJES / NOTIFICACIONES -----
----- MENÚ DE COMUNICACIÓN -----
1) Enviar mensaje
2) Salir
Ingrese el número de la opción deseada: 1
Mensaje: Hola NODO C

lab3_redes - Python flooding.py - 39x...
ling the faster cython/libidn one.

----- FLOODING ---
Seleccione un nodo de la lista:
1. alfa@alumchat.xyz
2. betta@alumchat.xyz
3. charlie@alumchat.xyz
4. delta@alumchat.xyz
Ingrese el número del nodo: 3

----- MENSAJES / NOTIFICACIONES -----
----- MENÚ DE COMUNICACIÓN -----
1) Enviar mensaje
2) Salir
Ingrese el número de la opción deseada: 1
--> Mensaje enviado a betta@alumchat.xyz.

lab3_redes - Python flooding.py - 39x...
ling the faster cython/libidn one.

----- RETRANSMITIENDO MENSAJE -----
-->> Retransmitiendo mensaje a charlie@alumchat.xyz.
-->> Mensaje enviado a delta@alumchat.xyz.

lab3_redes - Python flooding.py - 38x...
-->> A ha enviado un mensaje: Hola NODO C
----- MENSAJES / NOTIFICACIONES -----
----- MENÚ DE COMUNICACIÓN -----
1) Enviar mensaje
2) Salir
Ingrese el número de la opción deseada: 1
----- RETRANSMITIENDO MENSAJE -----
-->> No hay nodos a los que retransmitir el mensaje.

```

Cabe destacar que al retransmitir mensajes, únicamente se reenvía a aquellos nodos que no habían sido visitados anteriormente. Por ejemplo, el nodo B solo le envía el paquete al nodo C. No se le envía al nodo A (a pesar de que es un vecino directo) ya que el paquete ya había pasado por dicho nodo; así evitando ciclos infinitos.

Discusión

En el presente laboratorio se utilizó el servidor “alumchat.xyz” para simular una red sobre el protocolo XMPP. Cada usuario conectado en este servidor representaba un nodo de la red. A través de los tres algoritmos de enrutamiento desarrollados (*Dijkstra Link State*, *Flooding*, y *Distance Vector Routing*) se logró encontrar caminos óptimos entre los nodos para poder transmitir mensajes. Cabe destacar que los objetivos planteados se alcanzaron con éxito, ya que se logró comprender el funcionamiento de los distintos algoritmos de enrutamiento utilizados en las implementaciones actuales de internet.

Cabe mencionar que los algoritmos de enrutamiento se habían desarrollado en la primera fase del laboratorio. Asimismo, se había verificado el correcto funcionamiento de los mismos anteriormente. Sin embargo, en este laboratorio se implementó la red simulada sobre el protocolo XMPP. Por tal motivo, fue necesario hacer una serie de ajustes y cambios a la lógica original para poder utilizar dicho servidor con los algoritmos implementados. De igual manera, se tuvo que realizar pruebas extensas para verificar que los algoritmos implementados aún funcionaran correctamente, que los mensajes enviados por la red simulada llegaran a su destino, que no hubieran ciclos infinitos, entre otros aspectos.

En este caso, pudo observar cómo el algoritmo de *Flooding* destacó por su sencillez a comparación de los otros algoritmos implementados. En este algoritmo los nodos no compartieron su información (tablas de enrutamiento o vectores de distancia), ni calcularon las rutas más cortas. Sino, en este caso cada nodo envía los paquetes con mensajes a todos sus vecinos (exceptuando a aquellos nodos que ya procesaron el paquete con anterioridad). Esto asegura que los paquetes lleguen –eventualmente– al destinatario. Cabe destacar que no se presentó ningún problema al integrar XMPP. Esto debido a que se implementó programación defensiva para evitar que un paquete pasara por un nodo más de una vez. De tal manera, se envían los paquetes a través del servidor de forma ordenada.

Por ejemplo, en las pruebas realizadas, se pudo observar cómo un nodo le envió un paquete a un nodo vecino. Este nodo receptor no era el destinatario, por lo cual debía de reenviar el paquete a sus vecinos. Sin embargo, debido a la programación realizada, el nodo únicamente le envió el paquete a todos los nodos vecinos excepto el nodo que le envió el paquete originalmente. Esto permitió que el programa no se quedara en un ciclo infinito y sobrecargara el servidor “alumchat.xyz”. Asimismo, cabe destacar que también se implementó un registro con los mensajes recibidos. De tal manera, si existe una duplicación de paquetes y un mensaje llega a su destino varias veces, únicamente se muestra el mensaje una única vez. Esta duplicación de mensajes surge porque múltiples copias del mismo paquete pueden circular por la red simultáneamente. Esto puede ocasionar problemas de congestión en la red, ya que innecesariamente se están transmitiendo paquetes que posiblemente ya llegaron a su destino.

En el caso de *Dijkstra Link State*, se pudo observar como cada nodo inicialmente compartía grandes cantidades de información con sus vecinos a través

de mensajes enviados por el servidor. Más específicamente, se transmitían y recibían “echos”, así como las tablas con costos de enlace varias veces. Cabe destacar que esta transmisión masiva de datos causó ciertos errores inicialmente. Ya que se recibían varios mensajes simultáneamente en cada nodo, a veces no se procesaban correctamente los mensajes o simplemente no se recibía más de alguno. Para solucionar este problema, se incluyó diferentes métodos de sincronización para garantizar que solamente se recibiera un mensaje a la vez y en orden.

A través de la solución anteriormente descrita, el algoritmo de *Dijkstra Link State* fue capaz de compartir información necesaria entre nodos sin errores. Gracias al correcto funcionamiento de este proceso, cada nodo tenía un mapa completo de la topología de la red. Por tal razón, a diferencia del algoritmo de *Flooding*, este algoritmo de enrutamiento era capaz de enviar un mensaje por un camino específico. De tal manera evitando la duplicación de paquetes que anteriormente se explicó. Asimismo, resguardaba la privacidad del paquete ya que solo pasaba por donde era necesario.

Finalmente, cabe destacar que con el algoritmo de *Distance Vector Routing* se tuvo una situación similar a la de *Dijkstra Link State*. Con este algoritmo nuevamente se estaba compartiendo –de forma repetitiva– información entre los nodos (vectores de distancia). Cabe destacar que en este caso se estaba utilizando la ecuación de Bellman Ford en cada intercambio para poder actualizar las rutas de los caminos más cortos. Sin embargo, nuevamente se tuvo que implementar métodos de sincronización para garantizar que todos los mensajes recibidos con información fueran procesados de forma correcta. De esta manera, se logró que dicho procedimiento de intercambio de información se repitiera –sin errores– hasta que ya no había más cambios en los vectores de distancia. Cabe destacar que el algoritmo de *Distance Vector Routing*, similar a *Dijkstra Link State*, siempre logró detectar el camino más corto entre los nodos.

En conclusión, en esta práctica se comprendió a profundidad los algoritmos de enrutamiento utilizados en las implementaciones actuales de internet. Asimismo, se logró identificar las restricciones, limitaciones y ventajas específicas de cada algoritmo. Esto siendo esencial para poder identificar qué algoritmo utilizar a la hora de diseñar una red de comunicaciones. Finalmente, cabe destacar que la utilización del servidor “alumchat.xyz” en este laboratorio fue de mucha utilidad, ya que permitió analizar a más profundidad el funcionamiento automático de los algoritmos de enrutamiento.

5. Comentario

Esta práctica ha demostrado ser un valioso refuerzo para nuestra comprensión de los algoritmos de enrutamiento en redes de comunicación. A través de la implementación de los algoritmos *Distance Vector Routing*, *Dijkstra Link State* y *Flooding*, hemos adquirido un conocimiento profundo sobre cómo estos enfoques abordan el desafío de transmitir datos en una red. La simulación de los procesos de

envío de mensajes y la evaluación de sus resultados nos ha permitido apreciar las ventajas y limitaciones de cada algoritmo.

Más específicamente, hemos experimentado cómo el algoritmo de *Flooding*, a pesar de su simplicidad, puede generar ineficiencias y redundancias en la transmisión de paquetes. Por otro lado, *Dijkstra Link State* ha demostrado su capacidad para calcular rutas óptimas. Sin embargo, debido a la gran cantidad de información transmitida inicialmente, se tuvo que programar ciertos mecanismos para asegurar la integridad de la información recibida. Finalmente, *Distance Vector Routing*, también presentó buenos resultados para calcular rutas óptimas. Sin embargo, requirió de una comprensión profunda del algoritmo y de la ecuación de Bellman Ford para poder implementarlo.

En la primera fase del laboratorio, lo que más nos costó fue pensar cómo representar un nodo con cada instancia de los programas realizados. Sin embargo, ya que esta segunda fase utilizó XMPP, fue un poco más sencillo entender cómo lograr que los nodos se comunicaran entre sí mismos. Esto especialmente considerando la experiencia previa del proyecto, donde ya se había utilizando XMPP y el respectivo servidor.

Sin embargo, cabe destacar que sí tuvimos problemas con *Dijkstra Link State*, y *Distance Vector Routing* inicialmente debido a las grandes cantidades de información que los nodos se comparten para llenar las tablas de enrutamiento y los vectores de distancia. Sin embargo, después de una serie de pruebas y cambios lógicos, se logró solucionar dicho problema. De tal manera, se logró probar los algoritmos con topologías de hasta 9 nodos sin ningún problema.

En conclusión, este laboratorio nos ha permitido entender a más profundidad la teoría detrás de los algoritmos de enrutamiento utilizados hoy en día. Asimismo, nos ha dado un conocimiento práctico sobre cómo implementar y evaluar estos algoritmos en una red simulada utilizando el protocolo XMPP.

Conclusiones

1. El objetivo principal de esta práctica fue logrado pues se logró comprender a detalle el funcionamiento de los algoritmos de enrutamiento de *Distance Vector Routing*, *Dijkstra Link State* y *Flooding*.
2. El algoritmo de *Flooding*, a pesar de su sencillez, puede resultar ineficiente debido a la duplicación de paquetes y a la posible congestión de red. Sin embargo, se pueden implementar mecanismos para evitar que un mensaje se imprima más de una vez (como se realizó en el presente laboratorio).
3. Los algoritmos de *Distance Vector Routing* y *Dijkstra Link State* presentaron resultados óptimos debido a que podían calcular rápidamente la ruta óptima para que un mensaje llegara a su destino. Sin embargo, el proceso de llenar las tablas de enrutamiento de forma repetitiva fue complejo. Cabe destacar que de incrementar el tamaño de la topología, se podría tardar mucho este

- proceso. Asimismo, de no manejar correctamente los mensajes entrantes del servidor XMPP, se podrían presentar ciertos errores.
4. El mayor reto del laboratorio fue cambiar la lógica original de los algoritmos para integrar la red simulada sobre el protocolo XMPP. Esto siendo la base para posteriormente poder encontrar las rutas más cortas entre los distintos nodos y así enviar mensajes.

6. Referencias

1. Hanna, K. T. (2021). Flooding (network). Networking.
<https://www.techtarget.com/searchnetworking/definition/flooding>
2. TutorialsPoint (s. f.). Flooding in computer network.
<https://www.tutorialspoint.com/flooding-in-computer-network>
3. GeeksForGeeks. (2017). Distance Vector Routing DVR Protocol. GeeksforGeeks:
<https://www.geeksforgeeks.org/distance-vector-routing-dvr-protocol/>
4. Javapoint. (2021). Computer Network | Link State Routing Algorithm - Javatpoint. <https://www.javatpoint.com/link-state-routing-algorithm>
5. Haider, A. (2021). Advantages and disadvantages of link-state routing. Retrieved August 28, 2023, from Blogspot.com website:
<http://basicitnetworking.blogspot.com/2012/11/advantages-and-disadvantages-of-link.html>