.UBAfiuba
FACULTAD DE INGENIERÍA

Estrategias de Actualización de Firmware en una Red Inalámbrica Mallada

Tomás Danko (Padrón 107431)

Ingeniería en Informática

Directores:
Dr. José Ignacio Alvarez-Hamelin y Dr. Georgios Papadopoulos

Jurados:
Dr. Pablo Roca, Dr. Federico Zacchigna, Ing. Ricardo Veiga

Facultad de Ingeniería, Universidad de Buenos Aires. 05/06/2026.

Índice

Capítulo 1
Introducción

En este capítulo se introduce el problema abordado en esta tesis, dado en el contexto de redes del Internet de las Cosas (IoT) organizadas en topologías malladas. En particular, se analiza el desafío que representan las actualizaciones de firmware en este tipo de redes, debido a las restricciones propias de los dispositivos.

A continuación, se presentan brevemente los conceptos necesarios para comprender este escenario, para luego formular el problema que motiva la escritura de esta tesis.

1.1. Contexto: redes malladas y IoT

1.1.1. Redes malladas inalámbricas

Las redes malladas inalámbricas (del inglés, *Wireless Mesh Networks*, WMN) surgen en un contexto en el que la cobertura y la escalabilidad se vuelven limitaciones en las redes inalámbricas. En este tipo de topología, cada nodo cumple un doble rol: actúa como dispositivo final y, simultáneamente, como router capaz de reenviar paquetes hacia otros nodos (es decir, funcionan como encaminadores intermedios) [2].

Las WMN tienen ventajas fundamentales en comparación al resto de redes inalámbricas. En principio, la robustez y confiabilidad de la red se incrementan significativamente, pues la existencia de múltiples caminos entre cualquier par de nodos permite que la red tolere fallos en nodos o enlaces específicos sin perder conectividad. Por otro lado, este tipo de redes es altamente escalable; la adición de nuevos dispositivos se ve facilitada por la colaboración entre los nodos, que tienen permiso para extender la cobertura. Por último, las WMN y su funcionamiento distribuido se vuelven muy útiles en entornos de difícil acceso, donde la instalación de cableado es compleja.

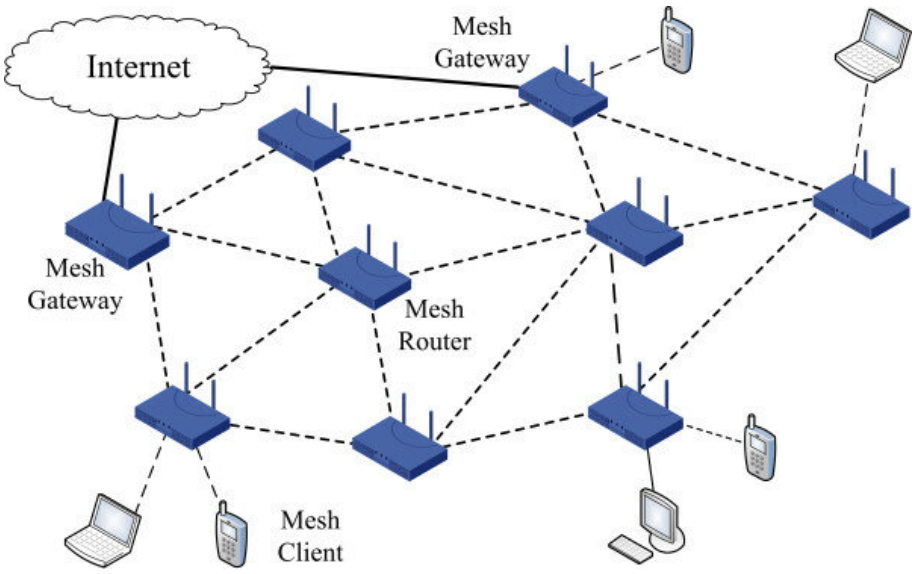

Figura 1.1: Ejemplo de red mallada inalámbrica (extraído de [3]).

Para poner en práctica este tipo de topologías, fue necesario diseñar nuevas tecnologías. Entre ellas, se destaca la familia de estándares 802.15 para la capa de enlace, gracias a la cual se permite la técnica de multisalto entre *hosts*. En particular, estos estándares se destacan por su diseño enfocado en el bajo consumo de energía y la baja transferencia de datos, lo que los vuelve ideales para dispositivos restringidos por el hardware.

1.1.2. El Internet de las Cosas

En el contexto de la creciente digitalización de entornos industriales, impulsada por el paradigma de la Industria 4.0, surge el *Internet of Things* (IoT), que propone la interconexión de dispositivos físicos mediante redes de comunicación, permitiendo su cooperación para cumplir objetivos comunes [4]. Estos dispositivos, tales como sensores y actuadores, cuentan con identificación única y son capaces de intercambiar información de manera autónoma, generando y procesando datos de forma continua. En este sentido, una característica fundamental de la arquitectura de los sitemas IoT es que suele organizarse en capas de hardware, middleware y presentación [5], que estructuran las tareas de captura, procesamiento y visualización de la información.

El uso de este tipo de tecnologías se extiende a múltiples dominios. En el ámbito industrial y de servicios, por ejemplo, es común el uso de sensores por parte de empresas de suministro eléctrico para medir el consumo en los hogares, permitiendo una gestión más eficiente de la red.

No obstante, el surgimiento de estas tecnologías viene acompañado de una serie de desafíos. Por un lado, los dispositivos involucrados suelen contar con recursos

limitados en términos de capacidad de cómputo, memoria y energía. Por otro, la integración de múltiples tecnologías y la necesidad de mantener comunicaciones confiables en entornos distribuidos incrementan la complejidad del sistema. Estas características tienen un impacto directo en tareas de mantenimiento de la red, como en la distribución de actualizaciones de firmware.

1.2. Problema: actualizaciones de firmware en redes inalámbricas malladas

Dadas las complicaciones de este novedoso paradigma, varios estándares han sido propuestos a lo largo del tiempo. En 2005, se formó el primer grupo de trabajo sobre IoT, *IPv6 over Low power Wireless Personal Area Networks* (6LoWPAN), dentro del seno del *Internet Engineering Task Force* (IETF), una organización internacional que busca contribuir a la ingeniería del internet. Este grupo dio lugar al estándar homónimo, 6LoWPAN, diseñado con el objetivo de lograr que dispositivos limitados participen de las redes IoT, lo que requirió de una integración del protocolo IPv6 (que, recordemos, era necesario dadas las limitaciones en cuanto a direccionamiento de IPv4).

En este contexto, **Wi-SUN FAN (*Field Area Network*)** [6] representa una tecnología avanzada de red inalámbrica diseñada específicamente para abordar aplicaciones de gran escala. Su capa de protocolos incluye estándares como 6LoWPAN y RPL, que será explicado más adelante. Su funcionamiento se basa en el uso de espectro de radio no licenciado, permitiendo así conexión a larga distancia, transmisión de radio de baja potencia, y una comunicación confiable entre dispositivos IoT distribuidos en áreas geográficas extensas. Esta tecnología se distingue de otros protocolos que atienden las mismas necesidades al estar intrínsecamente basada en IPv6, emplear bandas de subgigahercios y adoptar una topología en malla, que permite la conectividad directa entre los nodos. Además, se basa en una colección de estándares abiertos, lo que promueve la transparencia.

En el actual escenario de un mundo cada vez más interconectado, nuestras redes, tanto privadas como públicas, se enfrentan diariamente a crecientes amenazas cibernéticas. En estas circunstancias, la implementación de mecanismos de actualización de firmware para dispositivos se vuelve crucial, exigiendo soluciones robustas, eficientes y seguras que operen de manera inalámbrica. Es esencial que este proceso de actualización sea lo más rápido posible, minimizando su visibilidad para los usuarios de la red y garantizando que no afecte negativamente a los servicios en uso. Además, en el contexto de la posible propagación de brechas de seguridad en la red, es crucial que estos mecanismos se desenvuelvan de la forma más eficaz posible, con

tal de no obstruir la prevención de tales inconvenientes.

Es, por todo lo mencionado, que el propósito de esta tesis será desarrollar una estrategia de actualización de firmware para dispositivos IoT usando Wi-SUN que pondere:

- La disponibilidad de los servicios provistos mediante la red
- La congestión generada en la red, en función de la cantidad de paquetes enviados y la cantidad de reintentos
- El tiempo que se toma en aplicar las actualizaciones a un porcentaje de los nodos y a la totalidad de la red

Para ello, se abordará el marco teórico detallado, incluyendo desarrollos de la literatura actual. Posteriormente, se presentará el diseño de la estrategia propuesta, se detallará la metodología de simulación adoptada, y se analizarán los resultados obtenidos para validar el cumplimiento de estos objetivos.

Capítulo 2
Estado del arte

En este capítulo, se presentarán una serie de protocolos pertinentes a los dispositivos IoT, seguida diversas mejoras propuestas por diversos autores, que se consideran relevantes para el aumento de la eficiencia en las actualizaciones de firmware. Asimismo, se desarrollará una serie de algoritmos propios de la teoría de grafos, que sientan las bases para la propia solución que fue desarrollada a los efectos de esta tesis.

2.1. Wi-SUN

Wi-SUN (*Wireless Smart Ubiquitous Network*) es un novedoso protocolo de redes de malla inalámbricas diseñado para aplicaciones IoT a gran escala en exteriores [6]. Surge de la necesidad de soluciones de redes inalámbricas fiables para infraestructuras inteligentes, y permite ser utilizado en una amplia gama de aplicaciones. En base a esta tecnología, se han desarrollado varias especificaciones técnicas para dar soporte en consideración de diversos objetivos. En particular, nos interesa **Wi-SUN FAN**, frecuentemente utilizada para dispositivos como sensores o monitores hacia la nube.

Wi-SUN FAN hace uso de una serie de estándares IEEE para redes de baja potencia y alta pérdida, que se muestra en la figura 2.1.

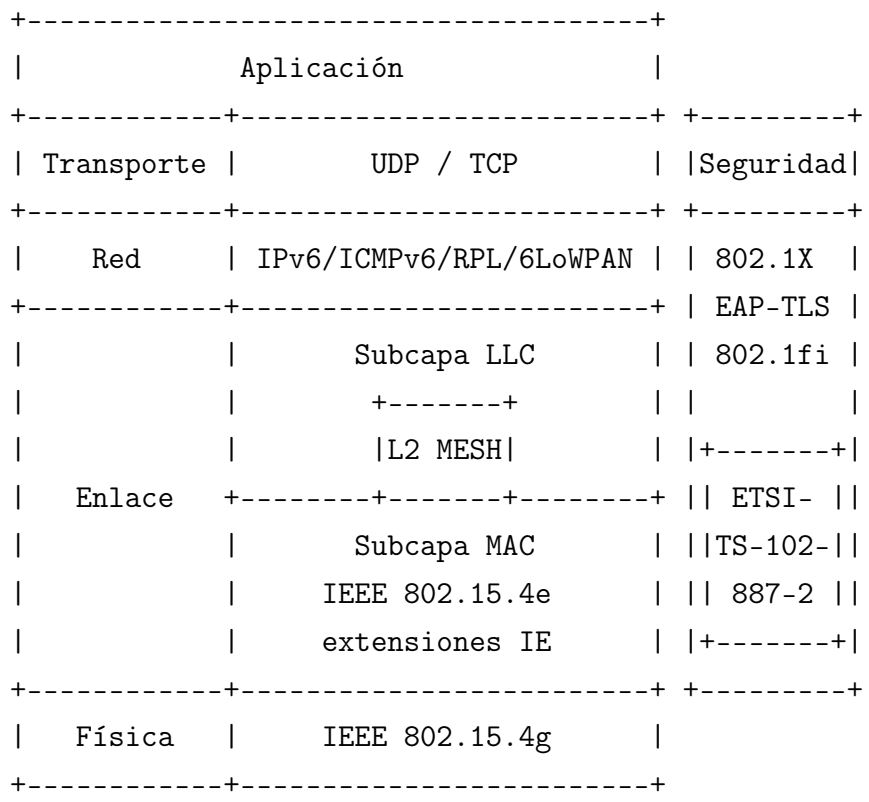

Figura 2.1: Descripción general del stack de Wi-SUN FAN.

En la capa física y la subcapa de control de acceso al medio, se utiliza el protocolo IEEE 802.15.4, que provee comunicación bidireccional con una alta tasa de datos, baja latencia y bajo consumo de energía, lo que lo vuelve ideal para redes IoT. Esto es gracias al uso de varias técnicas como la de FHSS (*Frequency Hopping Spread Spectrum*), que es un método de modulación de señal en el que se emite sobre una serie de radiofrecuencias aleatorias, reduciendo al mínimo la interferencia y aumentando la seguridad de los datos. En lo que respecta a la capa de red, se hace uso del estándar 6LoWPAN [7], que es el que permite que dispositivos de baja potencia logren participar de redes inalámbricas a través de IPv6. El enrutamiento de mensajes es llevado a cabo a través del protocolo para redes de bajo consumo RPL, que en Wi-SUN se suele utilizar en modo de no almacenamiento (*non-storing*). Finalmente, la capa de transporte provee servicios tanto UDP como TCP, siendo UDP predominante debido a sus costos reducidos de transmisión.

Wi-SUN FAN representa el marco tecnológico general de esta tesis, cuyos protocolos subyacentes requieren implementaciones de algoritmos y mensajería que permitan llevar a cabo actualizaciones de firmware de la forma más eficiente posible. En este sentido, el análisis se centrará en el protocolo RPL y en su arquitectura interna, dado que sus mecanismos, así como sus posibles extensiones y modificaciones, resultan fundamentales para abordar el problema planteado en esta tesis. El presente trabajo no propone una modificación del estándar Wi-SUN en su totalidad, sino

una extensión del protocolo RPL, orientada a mejorar la eficiencia del proceso de actualización de firmware dentro de redes que utilizan dicho marco.

2.2. RPL

2.2.1. Definición del protocolo

El protocolo de ruteo IPv6 para Redes de Baja potencia y Alta Pérdida (*Low-power and Lossy Networks*, LLNs), normalmente referido como **RPL**, es un protocolo de capa de red utilizado ampliamente en redes IoT y adoptado por tecnologías como Wi-SUN. Dicho protocolo surge dadas las limitaciones de los dispositivos que integran este tipo de redes: bajo poder de procesamiento, baja memoria y restriciones energéticas. Los mismos suelen estar interconectados por medio de enlaces con pérdida, que normalmente soportan una baja proporción de datos, y en las que suelen darse patrones de tráfico punto a punto, punto a multipunto y multipunto a punto.

Para que el protocolo sea útil en un gran rango de aplicaciones con distintos propósitos, RPL separa el procesamiento y la propagación de paquetes del **objetivo de optimización** del enrutamiento (que depende del uso, como minimizar el uso de energía o satisfacer ciertas limitaciones). Las operaciones en RPL requieren de conexiones bidireccionales y de una articulación para obtener y transportar información de control, con el objetivo de evitar la formación de bucles. De esta forma, provee un mecanismo para diseminar información alrededor de la topología de red formada de manera dinámica. Esto es así pues las LLNs no suelen tener típicamente topologías predefinidas: es trabajo de RPL descubrir posibles conexiones y seleccionar pares.

En lo estructural, RPL organiza la topología de red en la forma de un grafo acíclico direccionado (*Directed Acyclic Graph*, en adelante **DAG**), que es particionado en uno o varios *Destination-Oriented DAGs* o **DODAG**s. Los DODAGs son DAGs en los que todos los caminos llevan a un mismo destino, que se corresponde con el nodo designado como *raíz* de dicho grafo. Cada topología formada está contenida en una *instancia* de RPL, que puede contener uno o más DODAGs y, a su vez, cada red puede tener diferentes instancias asignadas, las cuales pueden estar optimizadas para distintas aplicaciones. Cada paquete enviado en RPL se encuentra asociado a una instancia en particular, razón por la cual incluyen el ID de la misma en su *payload*.

En la figura 2.2 se muestra un ejemplo de instancia. La instancia 1 se compone de dos DODAG, liderados por las raíces 2 y 3, mientras que la instancia 2 se compone de un DODAG, dirigido hacia la raíz 1.

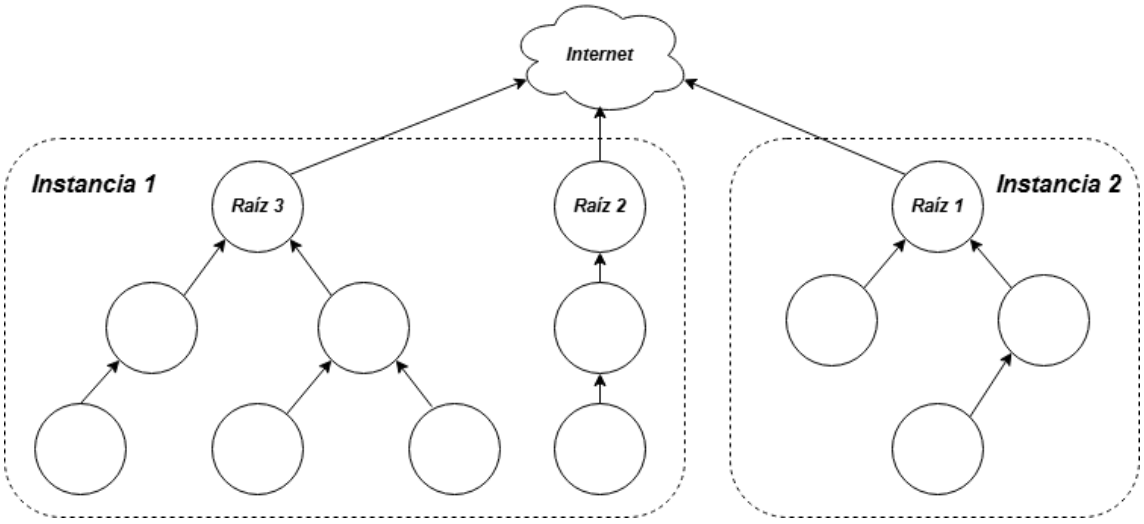

Figura 2.2: Ejemplo gráfico de instancias de RPL.

RPL forma las distintas rutas hacia la raíz haciendo uso de una **función objetivo** (*Objective Function*, OF), que define cómo los nodos seleccionan y optimizan rutas dentro de las instancias a las que pertenecen. Una OF especifica la forma en que los nodos traducen una o más métricas de enrutamiento en un valor llamado **rango** (*rank*), el cual determina la posición relativa de cada nodo dentro del DODAG y condiciona la elección de su nodo padre. Este valor escalar es utilizado por RPL como parte de sus mecanismos de validación para detectar la formación de ciclos durante el envío de mensajes, aunque dichos mecanismos no eliminan completamente la posibilidad de que estos ocurran. Cabe destacar que el rango no es necesariamente el costo del camino hacia la raíz, pero normalmente su valor se ve influenciado por el mismo. En este sentido, a partir del Rank puede interpretarse que el DODAG queda organizado en **capas** lógicas, donde cada capa agrupa nodos con valores de Rank similares y que se encuentran a la misma distancia lógica respecto de la raíz.

RPL implementa una serie de mensajes de control, que serán detallados a continuación:

- *DODAG Information Solicitation* (Solicitud de Información del DODAG o **DIS**): este mensaje es utilizado para solicitar el envío de un mensaje DIO por parte de un vecino RPL. Un nodo puede enviar este mensaje en modo *broadcast* con el objetivo de sondear a los nodos vecinos en busca de nuevos DODAGs.

- *DODAG Information Object* (Objeto de Información del DODAG o **DIO**): este mensaje contiene la información necesaria para permitir a un nodo descubrir una instancia RPL nueva, aprender sus parámetros de configuración, seleccionar un set de padres, y mantener el DODAG.

- *Destination Advertisement Object* (Objeto de anuncio de destino o **DAO**): este mensaje se utiliza para propagar la información del destino hacia arriba en el DODAG, permitiendo así la formación de rutas descendentes desde la raíz. En caso de ser explícitamente pedido, estos mensajes deben contestarse con un DAO-ACK, para confirmar su recepción y el correcto funcionamiento del camino formado.

Cada uno de estos mensajes puede, además, incorporar distintas **opciones** (*options*), que permiten detallar el contenido esperado en las respuestas o el modo en que los nodos deben comportarse al recibirlos. A los efectos de esta tesis, solo debemos tener en cuenta la opción de Información Solicitada (*Solicited Information*). La misma puede estar presente en los mensajes DIS recientemente introducidos, y es utilizada para que un nodo solicite mensajes DIO de un subconjunto de nodos vecinos. En dicha opción se especifica una serie de criterios que los nodos deben cumplir para ser considerados aptos para contestar, limitando así la congestión en la red.

El descubrimiento de rutas hacia arriba permite a los nodos unirse a un DODAG, lo que se realiza con un sondeo de los nodos que pertenecen al mismo. Dicho proceso tiene en cuenta dos conjuntos. Por un lado, el conjunto de vecinos candidatos se compone de aquellos nodos que pueden alcanzarse a través de mensajes directos por medio de un enlace. Por otro lado, el conjunto de padres es un subconjunto restringido del conjunto de vecinos candidatos. En base a ellos, se determina el **padre preferido**, que es el miembro del conjunto de padres que posee el menor rango de entre todos ellos. En este sentido, no debemos pasar por alto que el rango va a estar determinado así por la función objetivo y las métricas de ruteo correspondientes.

Los nodos pueden modificar su rango dentro del DODAG, siempre y cuando el mismo sea menor al rango de su padre preferido. En este sentido, los nodos pueden *envenenar* las rutas del DODAG anunciando un rango **infinito**. Esto implica que el nodo se encuentra desconectado de la red y que ya no es capaz de actuar como un padre, razón por la cual ningún nodo debe poseer vecinos con rango infinito en su conjunto de padres. En caso de recibir un DIO con rango infinito, los nodos deben permanecer en el DODAG, reconectándose a través de otro vecino.

Entonces, a medida que los nodos reciben mensajes DIO, sus vecinos pueden ser promovidos a padres del DODAG según el proceso descrito. Pero la propagación de estos mensajes no es trivial, y se rige en base al uso del *Trickle timer*, que determina la necesidad de enviar o no mensajes de control en base a inconsistencias en la información recibida. Por ejemplo, si un nodo recibe un DIO de su padre indicando un rango mayor al suyo, se considera una inconsistencia. Según el RFC6550, un nodo no debería reiniciar su Trickle timer en respuesta a un mensaje DIS unicast, pero si

al recibir un DIS multicast genérico.

Respecto al descubrimiento de rutas hacia abajo, el mismo se hace por medio de los mensajes DAO antes descriptos: su propósito es que los nodos informen a sus ancestros sobre las direcciones que pueden alcanzar 'hacia abajo'. El proceso va a depender de la configuración de la instancia RPL en cuestión. Por un lado, si la instancia funciona en *storing mode*, los nodos almacenan las tablas de enrutamiento para su sub-DODAG (entonces, en cada salto de un mensaje, se revisa la tabla correspondiente al nodo para determinar el próximo paso). Por otro lado, si la instancia funciona en 'non-storing mode', los nodos no almacenan dichas tablas, y es la raíz del DODAG la que debe encargarse de inyectar la ruta a cada paquete para viajar entre dos nodos. Teniendo esto en cuenta, la propagación del DAO funciona de la siguiente manera:

- En *storing mode*, los padres reciben mensajes DAO de sus hijos correspondientes, y almacenan la información de enrutamiento hacia sus descendientes, de forma que mantienen una tabla de enrutamiento actualizada para alcanzar cualquier nodo de su subárbol. De esta forma, los paquetes descendentes se reenvían salto por salto, utilizando las tablas locales para determinar el siguiente salto hacia destino, lo que reduce el tamaño de los encabezados de los paquetes a costa de un alto uso de memoria.
- En *non-storing mode*, cada padre simplemente propaga cada mensaje DAO recibido, que contiene ínicamente la información del nodo que envió dicho mensaje y su padre preferido correspondiente. De esta forma, dichos mensajes de propagan hasta la raíz, que mantiene una tabla de enrutamiento actualizada del DODAG. De esta forma, los paquetes descendentes requieren de un *Source Routing Header* (SRH), que incluye la lista completa de saltos para alcanzar a destino. Esto implica que, para comunicación peer-to-peer, cada mensaje debe llegar hasta la raíz para luego volver a ser propagado hacia abajo, lo que permite ahorrar en memoria pero es muy ineficiente.

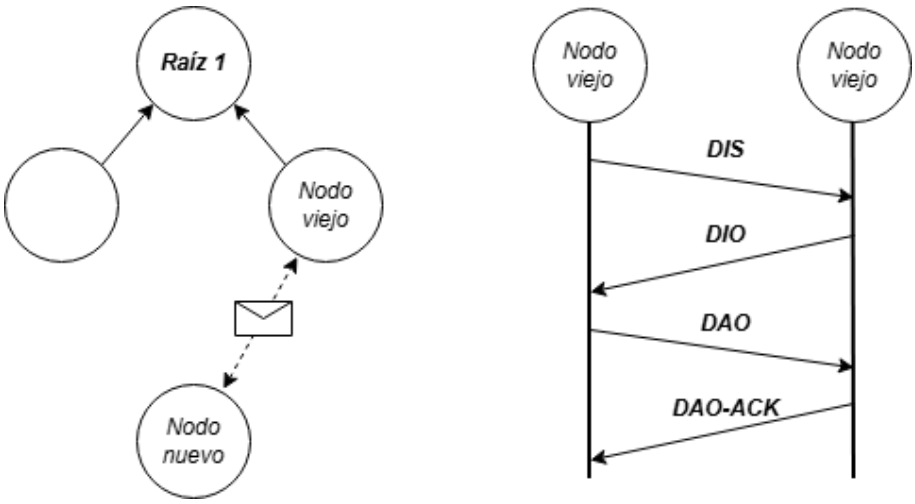

Figura 2.3: Gráfico del envío de mensajería RPL para la conexión de un nuevo nodo.

La Figura 2.3 ilustra el proceso mediante el cual un nodo recién incorporado descubre la red y se une a un DODAG existente. En primer lugar, el nodo emite un mensaje DIS en modo multicast con el fin de solicitar información de ruteo a los nodos vecinos. En respuesta, uno o más nodos envían un DIO, que contiene los parámetros necesarios para que el nuevo nodo participe de la instancia de RPL y seleccione un padre preferido. Una vez elegido dicho padre, el nodo transmite un mensaje DAO para anunciar su presencia y permitir la construcción de rutas descendentes. En ciertas configuraciones, este mensaje puede ser confirmado mediante un DAO-ACK, completando así el proceso de incorporación al DODAG.

Los DODAGs pueden encontrarse 'conectados a tierra' o 'flotar'. Un DODAG conectado a tierra permite conectividad a los nodos con hosts a través del internet, lo que contribuye a satisfacer el objetivo definido por la aplicación. Por otro lado, un DODAG flotante solo es capaz de proveer rutas entre los nodos dentros del DODAG, lo que puede ser muy útil para mantener la interconectividad durante etapas de reparación.

En lo que respecta a la seguridad en RPL, el protocolo ofrece tres modos de uso:

- Modo sin seguridad, en el que los mensajes de control de RPL se envían sin mecanismos adicionales de seguridad. El uso de este modo no significa que la red sea insegura, pues pueden estar usándose otras primitivas de seguridad (por ejemplo, en la capa de enlace) para cumplir con los estándares de la aplicación.
- Modo preinstalado, en el que los nodos que se unan a una instancia de RPL deben contener llaves preinstaladas que les permitan procesar y generar mensajes

RPL seguros.

- Modo autenticado, en el que los nodos también tienen llaves preinstaladas, pero que solo deben usarse al momento de unirse a la instancia de RPL.

Por último, en caso de fallas, tanto por una alta pérdida de paquetes como también por la detección de loops, la raíz del DODAG puede llevar a cabo una reparación 'global'. Para ello, todos los nodos almacenan un contador llamado 'número de versión del DODAG', que permite a los nodos en la nueva versión elegir una nueva posición cuyo rango no dependa del anterior rango que tuvieran definido en el DODAG anterior. Asimismo, RPL permite mecanismos de reparación 'local' dentro del DODAG, que pueden configurarse a partir de ciertos parámetros en los mensajes DIO. El RFC6550 no propone ningún método genérico de reparación local, pero más adelante se exponen algunas ideas de diversos autores.

2.2.2. Contiki OS

Contiki es un sistema operativo de código abierto, diseñado especialmente para dispositivos IoT inalámbricos de bajo consumo [26]. Fue producido por un equipo internacional de desarrolladores de múltiples agrupaciones, y gana su popularidad debido al uso de una pila TCP/IP integrada y su innovación al basarse en un kernel controlado por eventos. Esto se realiza por medio de la implementación de *protothreads*, que permiten explotar la concurrencia sin la necesidad de *stacks* por cada *thread*, lo que vuelve a Contiki muy útil para dispositivos con limitaciones de hardware. Aún con estas funcionalidades, este sistema operativo solo requiere de unos kilobytes de código y una RAM del orden de los bytes.

Dentro de los mecanismos de red que Contiki ofrece, se incluye la pila de redes IPv6. Si bien Wi-SUN FAN no utiliza la pila completa de Contiki, ambos comparten el uso de 6LoWPAN y RPL como núcleo del enrutamiento en redes LLN. Esto convierte al sistema operativo en un indispensable para llevar a cabo pruebas en un entorno lo más realista posible respecto de nuestro caso de uso. Cabe destacar, la implementación de RPL que se incluye provee una implementación básica de reparaciones locales, que supone envenenar al nodo en cuestión en caso de perder la conectividad.

En los últimos años, se desarrolló una nueva versión de Contiki, llamada **Contiki-NG (*Next Generation*)**. La misma continúa con las heurísticas del programa original, pero se enfoca en mejorar el rendimiento energético y la compatibilidad con los dispositivos IoT modernos, razón por la cual continúa utilizando 6LoWPAN y RPL.

En el marco de la necesidad por realizar simulaciones que den lugar al análisis

de las soluciones propuestas, Contiki incluye además un simulador de sensores llamado **Cooja** [27]. El mismo ofrece la posibilidad de construir topologías arbitrarias con una gran cantidad de nodos, permitiendo parametrizar diversas variables tales como la pérdida de paquetes o el rango de transmisión. Cooja se destaca por su personalización, pues permite que el usuario haga uso de su propia implementación de los nodos en función del objetivo que corresponda a la aplicación simulada, lo que significa que admite la comunicación entre dispositivos con diferentes versiones de Contiki. Además, ofrece un entorno de *scripting*, que permite manejar las simulaciones de forma automática, agregando y quitando nodos, enviando mensajes arbitrarios y/o incluyendo temporizadores, haciendo uso del lenguaje JavaScript. En la figura 2.4, se ofrece una captura de pantalla de la interfaz gráfica de COOJA.

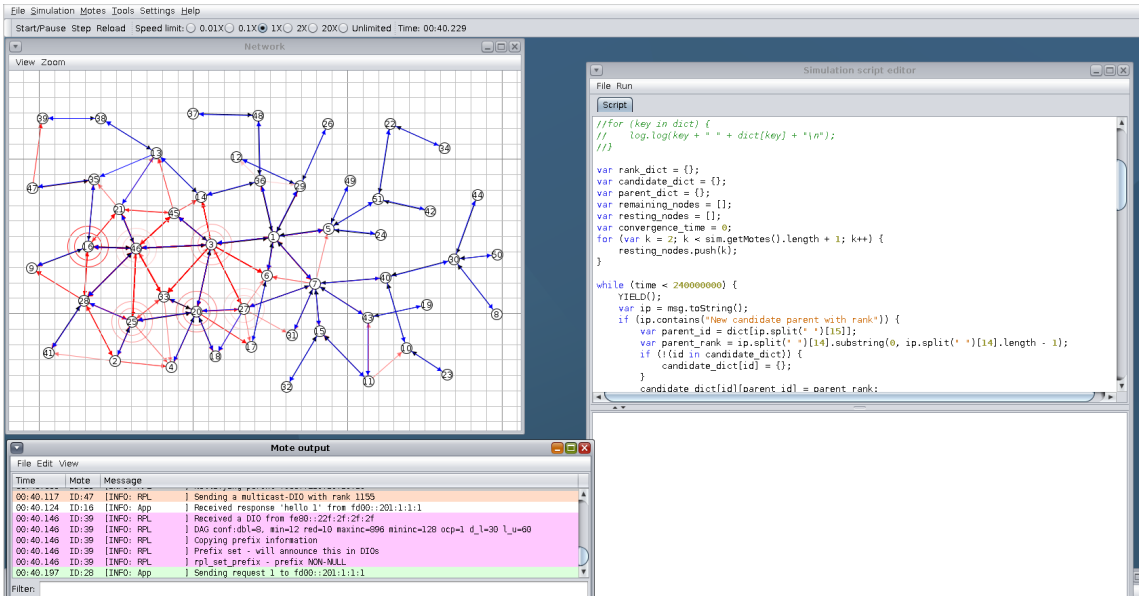

Figura 2.4: Interfaz gráfica de Cooja.

Para el desarrollo de esta tesis, Cooja fue la herramienta principal de simulación utilizada, pues permitió evaluar distintas estrategias de actualización de firmware en condiciones realistas.

2.3. Mejoras de RPL

A lo largo de los años, varios autores propusieron distintas mejoras al protocolo RPL antes expuesto, en consideración a las limitaciones derivadas del bajo nivel de procesamiento y memoria de los dispositivos participantes. A continuación, se repasa una serie de modificaciones de distintas autorías que son relevantes a la extensión de esta tesis.

2.3.1. Modificaciones del mensaje DIS

Recordemos: un nodo RPL puede mandar un mensaje de 'DODAG Information Solicitation', o DIS, para así recibir mensajes 'DODAG Information Object', o DIO, de otros enrutadores RPL vecinos. Un DIS puede llevar una opción de 'información solicitada' en la que se especifican las características de los DAGs en los que el nodo solicitante está interesado. Según el protocolo RPL original, los enrutadores consideran la recepción de un DIS multicast como una inconsistencia, a raíz de la cual resetean sus 'Trickle timers' para los DAGs correspondientes, lo cual deriva en el envío de los futuros mensajes DIO con una frecuencia mayor. Por otro lado, los mensajes DIS unicast se responden enviando un mensaje DIO por cada DAG correspondiente, sin reiniciar el Trickle timer. En caso de no enviarse ninguna opción de información solicitada, se reciben DIOs de todos los DAGs conocidos por los enrutadores solicitados.

Ahora, supongamos que un nodo hoja de RPL desea unirse a un cierto DAG. Este nodo puede esperar a que sus enrutadores vecinos transmitan mensajes DIO de forma voluntaria, o puede proactivamente solicitar DIOs utilizando un mensaje DIS. El problema es que el envío voluntario de un DIO puede pasar en un tiempo muy largo si la red es estable y los intervalos del Trickle timer alcanzaron valores altos, razón por la cual buscar DIOs de forma proactiva es la única opción razonable. Como el nodo no sabe qué vecinos pertenecen al DAG, se va a realizar el envío de un DIS multicast, que va a provocar que sus vecinos reinicien sus Trickle timers y empiecen a transmitir sus DIOs con mayor frecuencia. Esto tiene un gran inconveniente: los enrutadores van a seguir transmitiendo DIOs con mucha frecuencia por una duración considerable, hasta que los Trickle timers vuelvan a alcanzar intervalos largos, de forma completamente innecesaria y generando congestión en la red.

Además, ciertos escenarios pueden requerir que un enrutador RPL solicite un DIO de uno de sus padres utilizando un DIS unicast. El padre está obligado a incluir una opción de 'configuración' dentro del DIO unicast, y el enrutador original podría todavía tener dicha información contenida de manera local. Esto infla el DIO unicast de manera innecesaria por 16 bytes en cada pedido.

En este sentido, hay una necesidad en la industria por el uso de flags y opciones DIS que permitan a un nodo de RPL controlar la forma en que sus enrutadores vecinos responden a su solicitud por mensajes DIO, expresando:

- las características de ruteo que los enrutadores deben cumplir para estar habilitados para responder, así reduciendo el número de respondedores
- si los enrutadores correspondientes deberían reiniciar o no sus Trickle timers, limitando la cantidad de DIOs transmitidos

- si los enrutadores correspondientes deberían responder con un DIO unicast en vez de una multicast, reduciendo la congestión en la red
- si los enrutadores correspondientes deberían omitir las opciones DIO que no fueron explicitamente pedidas, reduciendo así el tráfico
- el intervalo temporal en el que los enrutadores correspondientes deberían programar sus transmisiones de mensajes DIO, evitando colisiones

Todas estas modificaciones suponen una mejora fundamental para el problema que estamos resolviendo, en el que los nodos de la red deben atravesar reinicios para luego reincorporarse a los DAGs correspondientes. Así, evitamos desestabilizar la red con el envío de mensajes innecesarios o de mayor tamaño del requerido.

Teniendo todo lo mencionado en cuenta, Gundogan y compañía [22] proponen una serie de modificaciones a los mensajes DIS. En principio, proponen la implementación de tres nuevos flags:

- El flag de 'No Inconsistencia' (N): al recibir un DIS multicast con el flag N prendido, un enrutador RPL no debe reiniciar sus Trickle timers para los DAGs correspondientes. Además, el enrutador debe responder explícitamente enviando un DIO, que incluya una opción de configuración. El intervalo de envío de este mensaje va a depender de si se incluye o no una opción de 'Propagación de la Respuesta', que mencionaremos más adelante.
- El flag de 'Tipo de DIO' (T): en caso de que el flag N esté prendido, el flag T especifica el tipo de DIO que se envía a modo de respuesta. Si T está prendido, debe enviarse un DIO unicast, y sino debe enviarse un DIO multicast.
- El flag de 'Pedido de opción de DIO' (R): al recibir un DIS con el flag R prendido, el receptor debe incluir todas las opciones que fueron pedidas por el DIS incluyendo las opciones de 'pedido de opción DIO' que mencionaremos más adelante. Un enrutador RPL no debe responder incluyendo otras opciones DIO que no fueron pedidas explícitamente. Notemos que este comportamiento se contradice del RFC-6550 original de RPL para el caso en que se incluye una opción de configuración en todos los DIOs pedidos por un DIS unicast.

Vale aclarar que, al enviar un mensaje DIS unicast, ambas flags N y T deben ser 0, que son los valores por defecto según el RFC-6550. A continuación, se muestra una figura con el objeto base del DIS modificado:

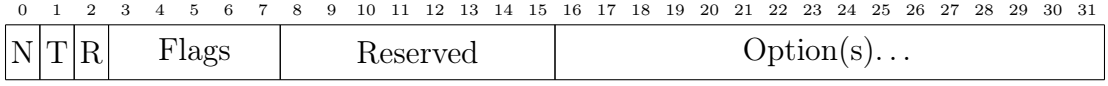

Figura 2.5: Modified DIS Base Object

Por otro lado, se proponen varias opciones DIS nuevas.

En primer lugar, se propone el 'Contenedor de Métricas', que permite incluir restricciones de ruteo a un DIS para reducir el número de enrutadores que deben responder al mismo (solo pueden responder aquellos enrutadores que satisfagan dichas características). Los contenedores de métricas fueron definidos en el RFC-6550, inicialmente solo permitidos en DIOs. Al recibir un DIS con esta opción, el enrutador RPL debe evaluar los objetos de restricciones y compararlos con el valor de la métrica de ruteo correspondiente que dicho enrutador mantiene para los DAGs en cuestión. Todas las métricas de ruteo deben satisfacer las restricciones mandatorias para que el enrutador esté habilitado para responder al DIS. Esta opción puede utilizarse tanto para DIS en unicast como en multicast.

Luego, se propone la opción de 'Propagador de la Respuesta'. Cuando un enrutador RPL recibe un DIS con el flag de no inconsistencia y con esta opción, debe esperar un tiempo elegido de manera uniforme en un intervalo de $[0, 2^{IntervaloPropagacin}]$ antes de intentar transmitir el DIO correspondiente como respuesta. Si el DIS no incluye esta opción, el nodo es libre de transmitir el DIO como lo haría normalmente. Esta opción es muy útil considerando que un DIS multicast puede generar que una gran cantidad de enrutadores RPL tomen acción inmediata y respondan con mensajes DIO, lo que puede generar congestión y pérdida de paquetes. Esta opción puede usarse con un DIS unicast, pero no tiene ningún beneficio.

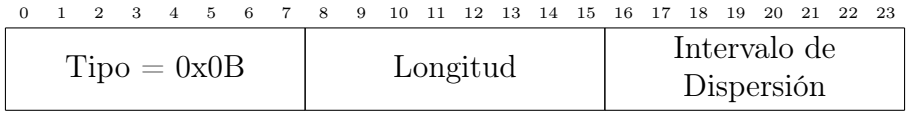

Figura 2.6: Opción de Dispersión de Respuesta

Por último, se propone la inclusión de la opción de 'Pedido de opción de DIO'. Si un DIS unicast se utiliza para pedir un DIO, según el RFC6550 se debe incluir una opción de configuración en este DIO. Dicha opción contiene generalmente información estática que no se modifica a lo largo del DAG. En escenarios en los que un nodo de RPL ya es parte del DAG y, por lo tanto, contiene dicha información, enviar esta opción es innecesario. Por ello, la implementación del pedido de opción de DIO permite un control total sobre las opciones del DIO solicitado. Al recibir un DIS unicast o multicast con el flag R prendido y con una o más de estas nuevas opciones, se debe responder con un DIO que incluya las opciones solicitadas.

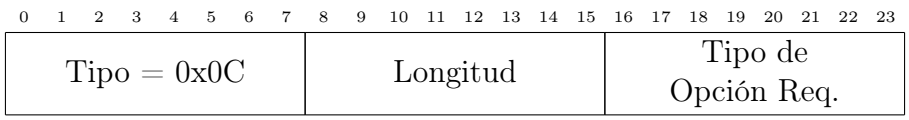

Figura 2.7: Opción de Pedido de Opción DIO

Académicos del IMT Atlantique llevaron a cabo simulaciones en las que se testeó la performance de estas modificaciones en RPL [23], en las que un nodo se

desconecta y reconecta a la red periódicamente, requiriendo del envío de mensajes DIS para ello. Los resultados fueron contundentes: la cantidad de DIOs enviados en simulaciones que hacían uso de los flags N y T se redujo drásticamente en comparación con la implementación estándar de RPL. Esto nos indica que las modificaciones propuestas en el paper suponen una mejora fundamental en lo que refiere a la congestión generada por la reconexión de un nodo a la red.

2.3.2. Algoritmos de reparación local

Un desafío en las redes de sensores inalámbricos es el de recuperar la conectividad de la red luego de que un nodo muere. Dicha desconexión puede afectar el throughtput de los datos y puede causar un particionamiento de la red. Generalmente, estos problemas se producen cuando nodos desconectados no tienen caminos alternativos con el mismo costo hacia su destino que el camino roto, razón por la cual se los conoce como nodos 'envenenados'. Para reconectar estos nodos, se puede usar una reparación local o global de la red, como vimos anteriormente. Sin embargo, para el caso de nuestro problema, una reparación local es más sensata, pues se realiza intercambiando información con nodos locales, a diferencia de la reparación global que genera un exceso de tráfico y retarda el proceso de recuperación. Recordemos, el RFC-6550 no propone ningún algoritmo de reparación local, y es por eso que varios autores trabajaron en desarrollar sus diferentes propuestas.

El autor J.Guo propone el procedimiento de los 'rangos fraccionales' [24], que se basa en un método de cálculo del rango que permite a los nodos modificar su rango sin cambiar su distancia relativa al sumidero. Para ello, los rangos se definen como fracciones, con un numerador entero m y un denominador entero n. Esto nos garantiza que siempre sería posible asignar un rango entre dos rangos consecutivos, permitiendo a nodos de alto nivel elegir a nodos con rangos decrecientes como padres sin generar ciclos en el grafo. Si quisiéramos calcular un rango nuevo entre dos rangos dados $R_1 = \frac{m}{n}$ y $R_2 = \frac{p}{q}$, usamos la siguiente fórmula: $sp(R_1, R_2) = \frac{m+p}{n+q}$. La misma nos garantiza que $R_1 < sp(R_1, R_2) < R_2$. De esta forma, la raíz se define con rango 0, y el rango infinito es 1. Los nodos van a elegir el rango mínimo entre sus padres candidatos, tal que $R_n \leq sp(R_M, \frac{1}{1})$.

De esta forma, el algoritmo de reparación local se basa en la transmisión de dos nuevos mensajes: DR-REQ (DODAG Repair Request) y DR-REP (DODAG Repair Reply). Un nodo envenenado disemina un mensaje de DR-REQ que contiene información como su identificador y su rango, que se va propagando hacia arriba hasta llegar a un nodo con un rango menor (no necesariamente el sumidero). Una vez llegado, dicho nodo responde con un mensaje DR-REP, que se envía en forma unicast al nodo que inició el proceso, y se actualiza el rango de cada uno de los nodos

N_i que propagan el mensaje hasta su destino, tal que $R(N_i) = sp(R(N_q), R(N_p))$, siendo N_q el nodo que inicia el proceso y N_p el padre de N_i. En la figura 2.8 se muestra este proceso.

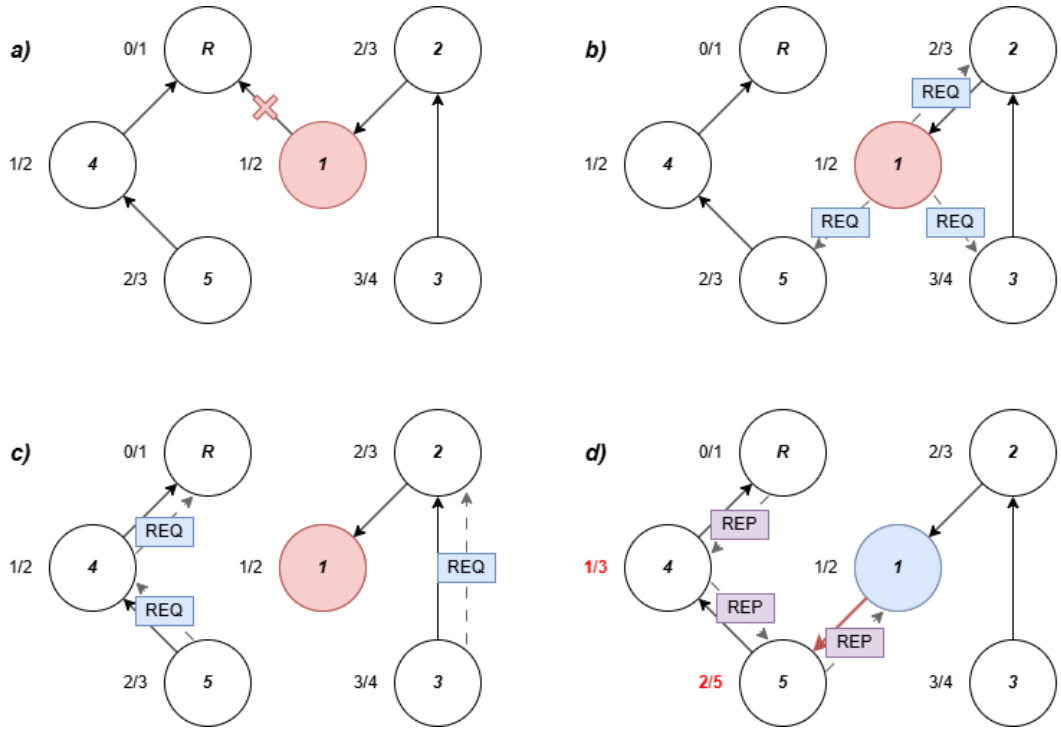

Figura 2.8: Ejemplo del algoritmo de rangos fraccionales en un grafo.

Como vemos el ejemplo, el nodo 1 pierde su conexión directa con la raíz, y envía un mensaje DR-REQ que se propaga a través de los nodos 5 y 4 hasta llegar otra vez a la raíz, que es el primer nodo en recibir el mensaje con un rango menor al del nodo solicitante. De esta forma, se propaga un mensaje DR-REP hacia el nodo 1 a modo de respuesta y se actualizan en este camino los rangos de los nodos por los que pasa dicho mensaje.

Este algoritmo produce muy poco overhead en lo que refiere a mensajería (pues, en el mejor de los casos, los mensajes enviados solo llegan a los vecinos), pero tiene varios inconvenientes. El primero es que si el único camino disponible es a través de uno o más nodos hijos, dicho camino nunca se encontrará, pues, como vemos en el ejemplo, los mensajes provenientes de nodos padres se descartan (como vemos que pasa con todos los mensajes que llegan al nodo 2 en el ejemplo). Por otro lado, si la muerte de un nodo produce que dos o más nodos hermanos pasen a estar envenenados, y la única forma de llegar a un nodo de rango mayor es a través de otro nodo hermano, es posible que dicho camino nunca se encuentre, produciéndose así una condición de carrera.

Por todo esto, los autores Kampen y Ovsthus proponen dos nuevos algoritmos de reparación que mejoran sobre la propuesta de Guo [25]. En primer lugar, se pro-

pone el método de 'número de secuencia'. Cada nodo en la red mantiene un número de secuencia local, para garantizar la selección de rutas libres de ciclos durante los procesos de reparación. Cuando un nodo pasa a estar envenenado, difunde una solicitud que incluye tanto su número de secuencia global (correspondiente a la versión del DODAG) como su número de secuencia local actual. Esta solicitud solo puede ser respondida por nodos que posean un número de secuencia global más reciente (mayor) que el del nodo solicitante, o bien por aquellos con el mismo número de secuencia global pero con un número de secuencia local más reciente. En caso de que un nodo no cumpla estas condiciones, simplemente retransmite la solicitud, utilizando una memoria caché para evitar la propagación de mensajes duplicados. Las respuestas generadas por los nodos aptos son enviadas por unicast hacia el nodo que originó la solicitud. Durante este proceso, cada nodo intermedio actualiza su número de secuencia local por el del nodo que respondió, lo cual le permite así también atender solicitudes de nodos que posean un número local más antiguo. Si la solicitud llega hasta el sumidero, este actualiza su número de secuencia local antes de enviar la respuesta correspondiente. Una vez que el nodo solicitante recibe una respuesta válida, puede incrementar su rango según sea necesario para unirse al nuevo camino propuesto. Al tener la respuesta un número de secuencia más reciente que el del nodo solicitante, se nos garantiza que hay efectivamente una ruta para llegar a la raíz que no involucra al nodo envenenado. En la siguiente figura, se muestra un ejemplo de su funcionamiento.

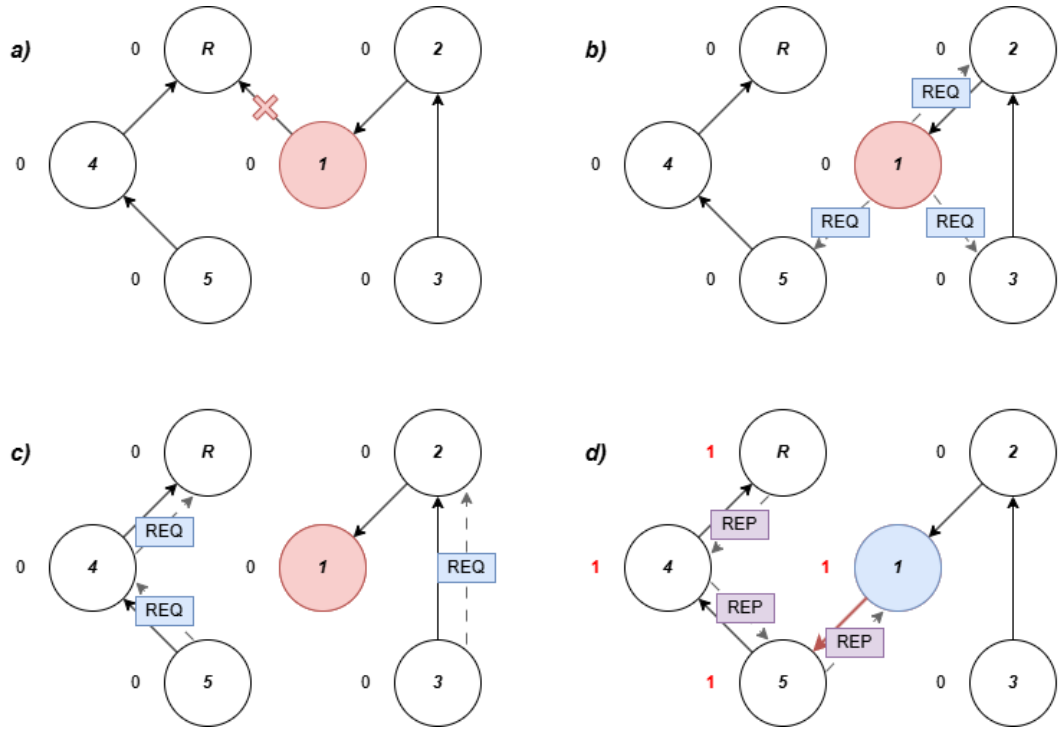

Figura 2.9: Ejemplo del algoritmo de números de secuencia en un grafo.

Como podemos ver, el nodo 1 pierde su conexión con la raíz, propaga un

mensaje de REQ. La propagación es muy similar a la del algoritmo anterior, solo que en este caso se actualiza solamente el número de secuencia local almacenado de los nodos que reciben el mensaje REP, para que así el nodo 1 actualice su rango correspondiente en función a su nuevo padre.

Entre las principales desventajas de este mecanismo se encuentra la posibilidad de pérdida de solicitudes o respuestas, al no tener ningún mecanismo de 'acknowledgement', lo cual puede afectar la rapidez de la reparación. Además, si ningún nodo intermedio ha realizado actualizaciones locales desde el último número de secuencia global, la solicitud será retransmitida hasta el sink, generando un mayor tráfico de control en la red.

Por lo expuesto anteriormente, los autores proponen otro método de reparación: el algoritmo ACK. Este método se basa en el "envenenamiento confiable" del sub-DAG para descubrir nuevas rutas libres de ciclos, para lo cual cada nodo necesita conocer quiénes son sus nodos hijos, información que se proporciona a través de los mensajes DIO. Cuando un nodo pasa a estar envenenado, transmite un mensaje 'DIO envenenado' a sus vecinos para informar que ha perdido la conexión con el nodo raíz. Todos los nodos hijos de un nodo envenenado confirman la recepción de este mensaje y buscan padres alternativos; en caso de no encontrar uno, ellos mismos pasan a estar envenenados antes de transmitir el mensaje de ACK. Una vez que todos los hijos de un nodo envenenado hacen esto, el nodo pasa a aumentar su rango y reconectarse al DODAG, con la seguridad de que no van a producirse ciclos. Para reducir la probabilidad de que se produzca 'oscilación entre rutas' y el overhead asociado a los mensajes enviados, se hace uso de temporizadores. Luego de recibir o transmitir un mensaje DIO envenenado, y durante un período de tiempo predefinido Δt, solo se procesan mensajes de tipo ACK o DIO envenenado. Una vez terminado dicho período, cada nodo elige al mejor padre candidato entre sus vecinos. En la siguiente figura, se muestra un ejemplo del funcionamiento de este algoritmo.

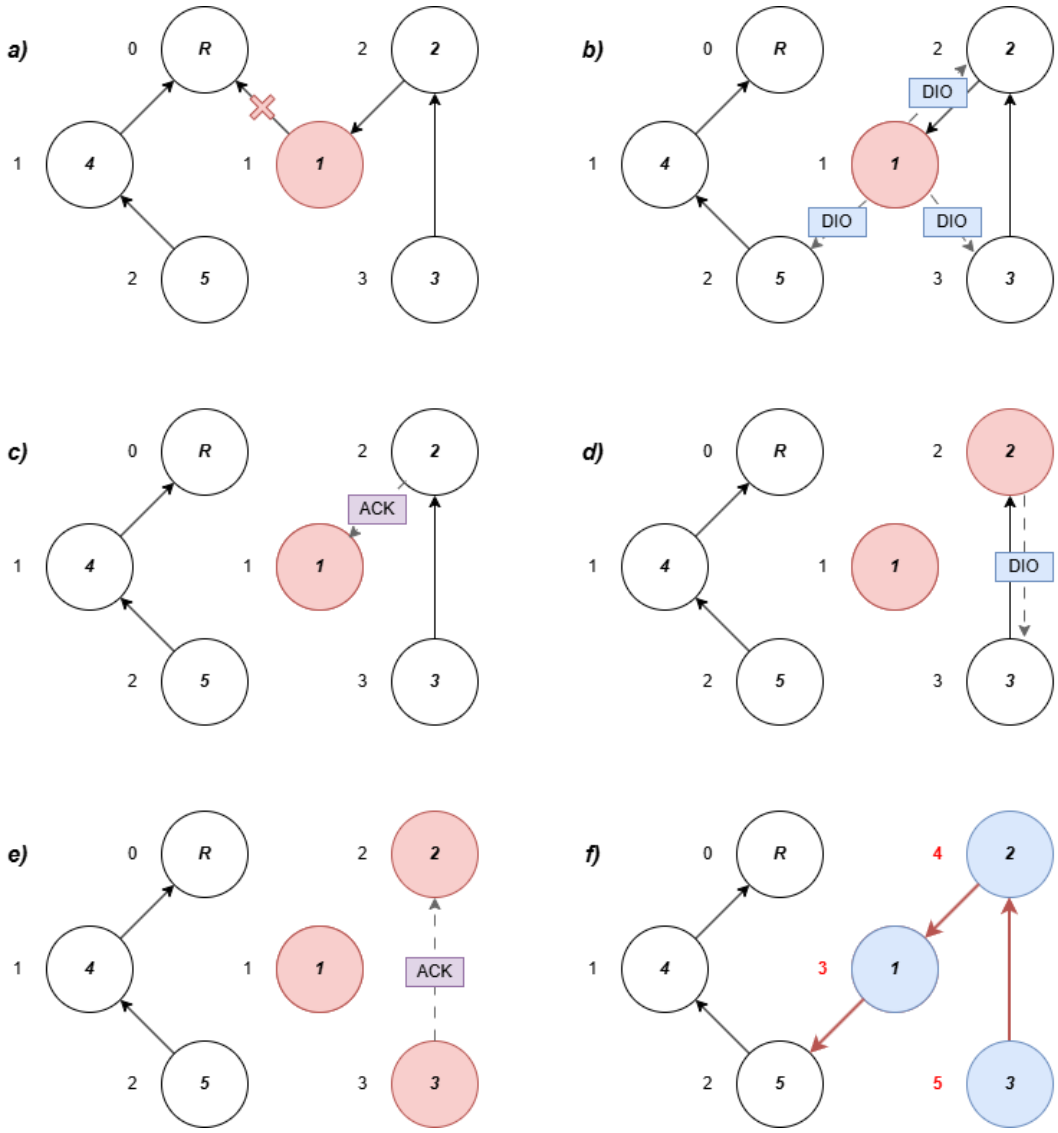

Figura 2.10: Ejemplo del algoritmo ACK en un grafo.

En el ejemplo, vemos que el nodo 1 pierde la conexión con la raíz, por lo cual disemina mensajes de DIO envenenado. A raíz de esto, el nodo 2 reconoce que su padre está envenenado, por lo cual pasa a estar también envenenado, y le responde con un mensaje de reconocimiento ACK. El mismo caso se produce con el nodo 3. Una vez pasados los timers correspondientes, cada nodo pasa a elegir al mejor padre candidato.

Como desventaja de este algoritmo, los mensajes ACK podrían no llegar al nodo envenenado, pero esto puede evitarse haciéndose un reenvío de los DIO envenenados en caso de no recibir el mensaje ACK esperado. Por otro lado, el tiempo de reparación se ve acotado inferiormente por el período correspondiente al temporizador, que es necesario para evitar congestión en la red.

En su paper, los autores Kampen y Ovsthus incluyen resultados de simulaciones realizadas sobre estos algoritmos. Por un lado, la propuesta de rangos fraccionales

alcanza un tiempo de convergencia menor en comparación a los otros dos algoritmos, pero el algoritmo de números de secuencia y el de ACK otorgan mayor fiabilidad. Además, el algoritmo de ACK produce el menor aumento en la cantidad de paquetes de mantenimiento de la red. Por todo lo expuesto, podría considerarse que este último es el algoritmo más robusto de cara a la realización de las reparaciones locales, ponderando entre fiabilidad, tiempo de convergencia y congestión.

2.4. Teoría de grafos

Para comprender la solución, es fundamental adentrarnos en conceptos relacionados a la teoría de grafos, que nos son particularmente útiles como abstracción para problemas en infinidad de áreas, y particularmente para el modelado de redes en general. En este sentido, se van a repasar en esta tesis varios problemas de grafos y los algoritmos con los que se solucionan hoy en día.

2.4.1. Puntos de articulación

En teoría de grafos, nos referimos a 'puntos de articulación' como aquellos vértices v de un grafo G no dirigido y conexo que, al ser eliminados, producen un incremento en el número de componentes conexas del mismo, desconectándolo y separándolo en diferentes partes. Definiéndolo más formalmente, dados tres vértices v, w y a, si el vértice a se encuentra en todos los caminos $p : v \Rightarrow w$ (y existe al menos uno de esos caminos), entonces a se considera un punto de articulación. En el análisis de redes, estos vértices corresponden a nodos fundamentales cuyas posibles fallas derivarían en problemas en la conectividad general de la red. Por esta razón, es vital identificarlos para evitar disrupciones en el tráfico.

Como se explica en [10], el algoritmo de búsqueda en profundidad, o DFS (*Depth-First Search*), permite detectar puntos de articulación en tiempo lineal. Imaginemos que G es un grafo que deseamos explorar. Inicialmente, todos los vértices de G se encuentran sin explorar, por lo que vamos a comenzar tomando alguno en particular, y seguir por alguna de sus aristas. Consecuentemente, vamos a llegar a un nuevo vértice, del cual vamos a atravesar nuevamente una de sus aristas, y así sucesivamente. La búsqueda DFS, en particular, consiste en tomar siempre aquellas aristas que emanen del vértice más recientemente explorado que contenga una arista sin explorar.

Supongamos ahora que G es un grafo no dirigido. Una búsqueda de G impone una dirección en cada una de sus aristas, dada por la dirección en que las mismas son recorridas, lo que nos da como resultado un grafo dirigido G'. Ahora, si nos

quedamos únicamente con el set de aristas que dan lugar a un vértice nuevo al ser recorridos durante la búsqueda, obtenemos un árbol de expansión (spanning tree) de G'. Un árbol de expansión de un grafo dirigido G corresponde a un árbol que es un subgrafo de G que contiene todos sus vértices; es decir, es G pero manteniendo solo alguna de sus aristas.

En la siguiente figura se muestra un ejemplo del árbol de expansión (a la derecha) obtenido a partir de un grafo (a la izquierda). Si recorremos en orden alfabético los nodos (con la regla de ir 'en profundidad'), le damos una orientación a los vértices dada por el árbol de la derecha. Una de las aristas del grafo original no forma parte del mismo, mostrándose en gris.

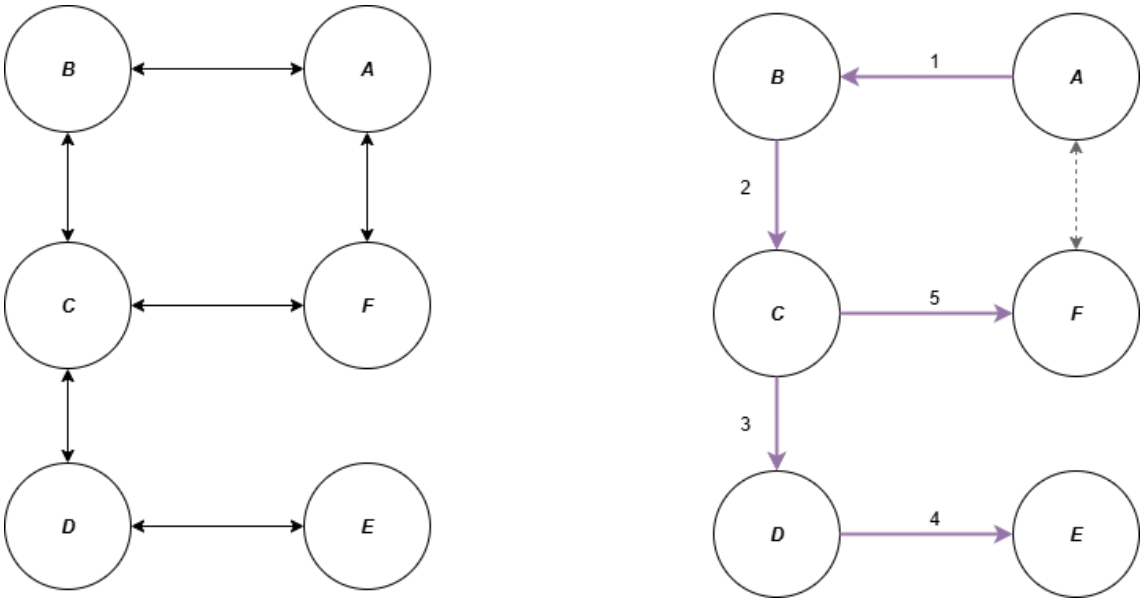

Figura 2.11: Ejemplo de recorrido DFS y árbol de expansión.

Tarjan propone en su paper un algoritmo para determinar las componentes biconexas de un grafo. Un grafo biconexo es un grafo 'inseparable', lo que quiere decir que, si uno de sus vértices es eliminado, no se produce un aumento de las componentes conexas. Este algoritmo se basa en la búsqueda de puntos de articulación. Hasta entonces, sólo existían algunos algoritmos como el de Shirey [18], que eliminaban de a uno los vértices del grafo y chequeaban la cantidad de componentes conexas resultantes para determinar que fueran puntos de articulación, lo que conducía a complejidades de $O(V * E)$. Tarjan se basa en la asignación de dos variables para cada uno de los vértices:

- **Disc**: es el orden en el que cada vértice fue descubierto por primera vez en la búsqueda DFS. Se asigna de manera incremental (por ejemplo, si empezamos por el vértice A, a este corresponde $disc[A] = 1$, y si seguimos por el B, corresponde $disc[B] = 2$).

- **Low**: representa el orden de descubrimiento más antiguo alcanzable desde el sub-árbol del vértice en cuestión, descendiendo por el árbol DFS hasta algún vértice en particular y atravesando una (y sólo una) arista de retroceso (que son aquellas que pertenecen al grafo original y no al árbol, y permiten ir desde un descendiente hasta alguno de sus ancestros). Formalmente, para cada vértice u descubierto, la formula con la que calculamos el low es la siguiente:

$$\text{low}[u] \;=\; \text{mín}\left(\text{disc}[u],\; \min_{(u,v)\,\in\, DFS\_Tree} \text{low}[v],\; \min_{(u,w)\,\in\, Back\_Edges} \text{disc}[w]\right)$$

Bajo estos dos conceptos, Tarjan define un lema. Supongamos que tenemos un grafo G no dirigido, y su árbol de expansión T correspondiente. Supongamos que además tenemos 3 vértices a, v y w tales que existe una arista entre a y v (es decir, $(a, v) \in T$), y w no es un descendiente de v en T. Si $low[v] \geq disc[a]$, esto significa que a es un punto de articulación de P, y su remoción provoca una desconexión entre v y w. Esto es así porque todos los caminos desde v que no atraviesan a se mantienen por dentro del sub-árbol dado por T_v, que no contiene el vértice w. El algoritmo propuesto busca calcular los valores low de todos los vértices del grafo en un único recorrido DFS.

De esta forma, se muestra un pseudocódigo para el algoritmo descrito:

Algorithm 1: Algoritmo de Tarjan para puntos de articulación

Input: Grafo conexo no dirigido $G = (V, E)$

Output: Conjunto de puntos de articulación

```
1  i ← 1;
2  PA ← {};
3  Función DFS(u, padre):
4      disc[u] ← i;
5      low[u] ← i;
6      i ← i + 1;
7      for v en la lista de adyacencia de u do
8          if v no fue visitado todavía then
9              DFS(v, u);
10             low[u] ← min(low[u], low[v]);
11             if low[v] ≥ disc[u] y disc[u] ≠ 1 then
12                 PA ← PA ∪ {u}
13         else if disc[v] < disc[u] y v ≠ padre then
14             low[u] ← min(low[u], disc[v])
15 return PA;
```

Como podemos ver en el pseudocódigo, el algoritmo se basa en la recursividad para hacer todos los cálculos correspondientes en una única pasada. En este sentido, los valores low correspondientes a cada vértice se determinan en 3 ocasiones: cuando se descubre el vértice, cuando se descubre una arista de retroceso, y cuando se vuelve hacia atrás en la recursividad. El valor que prevalece es el menor entre ellos.

Para entender bien este proceso, vamos a incluir un ejemplo. Para el mismo, vamos a hacer uso del siguiente grafo:

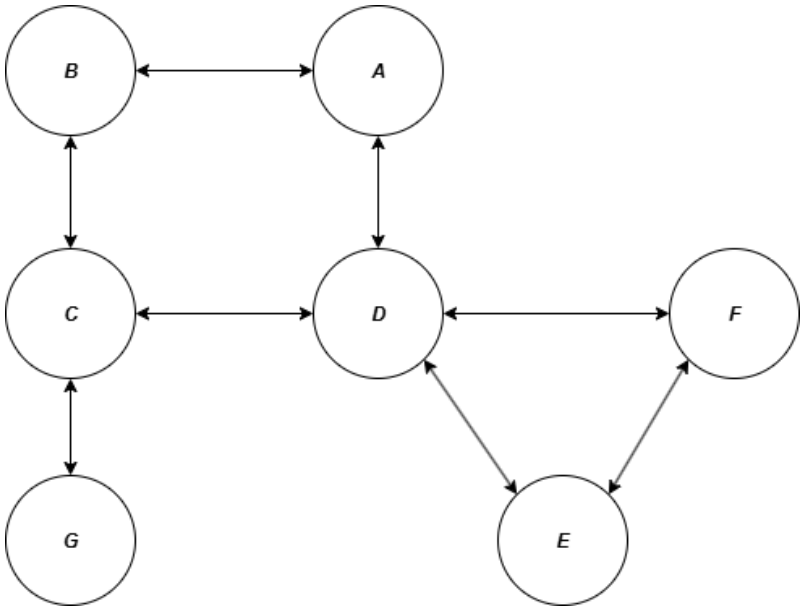

Figura 2.12: Grafo de ejemplo para ilustrar el algoritmo de Tarjan.

Ahora, vamos a realizar un recorrido DFS a lo largo de dicho grafo, en el que vamos a descubrir los vértices en orden alfabético. Dado dicho ordenamiento, los valores disc para cada vértice se muestran a la derecha de cada nodo.

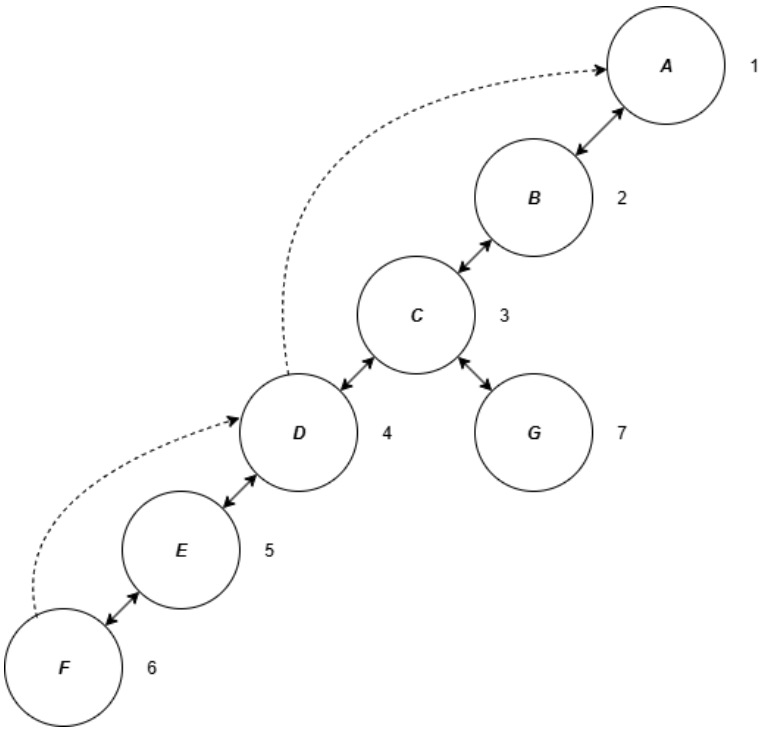

Figura 2.13: Grafo luego de descubrir todos los vértices.

Ahora, vamos a volver hacia atrás en la recursividad y calcular los valores low de cada nodo. Por ejemplo, enfoquémonos en F. Al descubrirlo, se setea low[F] a disc[F], es decir $low[F] = 6$. Luego, se explora su lista de adyacencia, y encontramos una arista de retroceso, que vamos a usar para recalcular low[F] como el mínimo entre sí mismo y disc[D] (que es a donde apunta la arista de retroceso). Así, $low[F] = min(disc[D], low[F]) = min(4, 6) = 4$. Como F no tiene descendientes en el árbol de expansión, no se vuelve a calcular low[F]. Repitiendo este proceso hacia atrás en la recursividad, vamos a obtener los valores low retratados en la figura (que se muestran al lado de los valores disc).

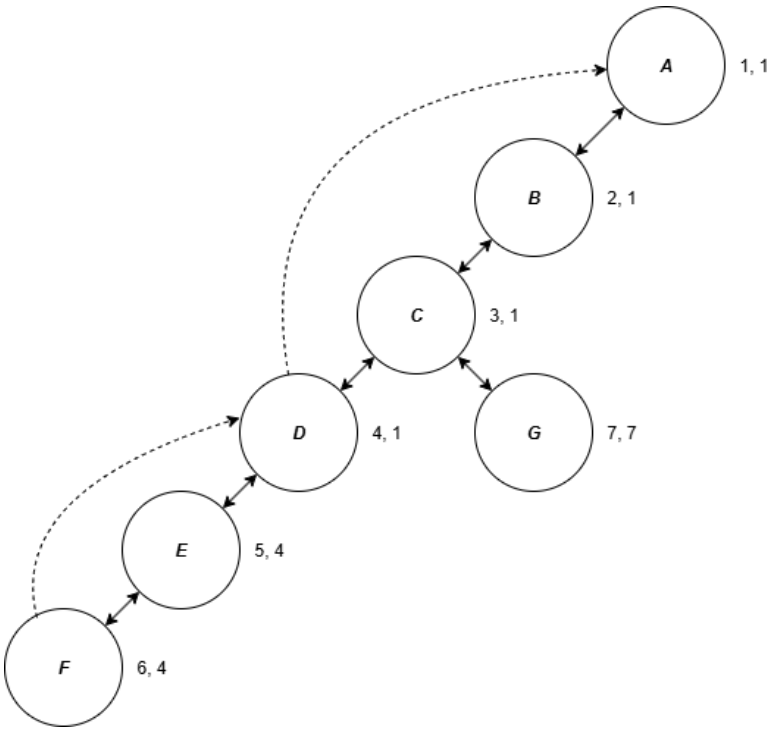

Figura 2.14: Grafo con los valores low calculados, luego de volver hacia atrás en la recursividad.

Con esto, ya tenemos calculados todos los valores necesarios. Y solo queda realizar las comparaciones correspondientes para cada arista. Por ejemplo, miremos la arista (C, G): $low[G] \geq disc[C]$, razón por la cual C es un punto de articulación en el grafo que causa la desconexión de G. El mismo caso se da para D, si repetimos el proceso con la arista (D, E).

Para terminar nuestro análisis, se procede a desarrollar la complejidad del algoritmo de Tarjan. El mismo solo requiere visitar cada vértice y cada arista una única vez gracias al uso de un recorrido DFS y su recursividad. Esto implica que el procedimiento en cuestión incurre en una complejidad lineal, es decir, $O(n + m)$.

2.4.2. Cortes mínimos de nodos

La conectividad de un grafo es una propiedad fundamental de los mismos, que suele ser muy aplicada en problemas relacionados a la fiabilidad de las redes (como ocurre para los puntos de articulación). Definimos que un grafo no dirigido está 'k-conectado' si y solo si entre cada par de vértices hay k caminos desarticulados (es decir, que no comparten vértices entre sí), lo que equivale a decir que el grafo permanece conectado aunque se remuevan $k-1$ vértices cualesquiera. Formalmente, y dado un grafo no dirigido k-conectado $G = (V, E)$, definimos que un subconjunto V' de k vértices de G es un k-conjunto separador si el subgrafo inducido por $V - V'$ no es conexo. En muchas ocasiones, se quiere buscar aquellos conjuntos de vértices

de tamaño mínimo que provocan la desconexión del grafo en cuestión, y es por eso que Kanevsky [11] presenta en su paper un algoritmo para encontrar dichos conjuntos para una conectividad k en general. Dicho algoritmo se basa en dos técnicas conocidas:

- La primera técnica es la reducción del problema de encontrar todos los conjuntos separadores de vértices mínimos de un grafo no dirigido al problema de encontrar todos los cortes de aristas de tamaño mínimo en un grafo dirigido, lo que se conoce como el algoritmo de Even-Tarjan [19].
- La segunda técnica es la de Provan-Shier para enumerar todos los cortes de aristas mínimos s-t [21].

Para comprender qué hace el algoritmo de Kanevsky, primero tenemos que dar algunas definiciones sobre grafos en general. Por un lado, vamos a definir que un grafo dirigido es fuertemente conexo si para cada par de vértices v y w hay al menos un camino dirigido de v a w y otro de w a v. Por otro lado, debemos entender qué significa el 'flujo' en los grafos. Dado un grafo no dirigido $G = (V, E)$ con dos vértices distinguidos, una fuente s y un sumidero t, y una capacidad $0 < cap(v, w) \leq \infty$ en cada arista (v, w), definimos el flujo en G como una función real no negativa entre pares de vértices que cumple tres propiedades fundamentales:

- El flujo nunca puede sobrepasar a la capacidad, es decir: $f(v, w) \leq cap(v, w)$. Si una arista (v, w) posee un flujo tal que $f(v, w) = cap(v, w)$, decimos que el flujo satura (v, w).
- Para cada vértice v distinto de s y t, el flujo que entra a v equivale al flujo que sale de v, es decir:

$$\sum_{w:(v,w)\in E} f(v, w) = \sum_{w:(w,v)\in E} f(w, v)$$

- El valor $|f|$ de un flujo f es el flujo neto que sale de la fuente, es decir:

$$\sum_{v:(s,v)\in E} f(s, v) - \sum_{v:(v,s)\in E} f(v, s)$$

  En este sentido, el problema del flujo máximo consiste en encontrar el flujo de máximo valor que es capaz de recorrer el grafo.

En base a un determinado flujo f en un cierto grafo, podemos definir a su grafo residual inverso N correspondiente, tal que:

$$N = (V, E^*), \quad \text{donde} \quad E^* = E_1 \cup E_2$$

$$E_1 = \{(i,j) \in E : f(i,j) > 0\}$$

$$E_2 = \{(j,i) : (i,j) \in E,\ f(i,j) < cap(i,j)\}$$

Esto quiere decir que el grafo N posee una arista (u,v) si y solo si hay flujo corriendo del vértice u al v, y posee una arista (v,u) si y solo si todavía queda capacidad disponible para que corra más flujo entre dichos vértices. Dentro de este grafo, definimos a los vértices incidentes como aquellos involucrados en el flujo positivo, es decir:

$$V(E_1) = \{v \in V : \exists (u,v) \in E_1 \text{ o } (v,u) \in E_1\}$$

Finalmente, definimos a un corte como un conjunto de aristas que cumplen:

$$(X, \bar{X}) = \{(u,v) \in E : u \in X,\ v \in \bar{X}\}$$

$$\bar{X} = V - X$$

Esto implica que dichas aristas separan al grafo en dos grupos disjuntos de vértices, razón por la cual se llama corte. Además, decimos que un corte es un corte $s-t$ cuando $s \in X, t \in \bar{X}$. Además, un corte $s-t$ es mínimo cuando su capacidad es la mínima posible.

Con todo esto definido, podemos proceder a exponer el algoritmo de Kanevsky, para un grafo no dirigido $G = (V,E)$. Como primer paso, vamos a determinar la conectividad k de G usando algoritmos de flujos. No entraremos en detalles respecto a dichos algoritmos, pero alcanza con mencionar que Kanevsky se basa en el algoritmo propuesto por Even y Tarjan, quienes introdujeron la reducción a flujos máximos, y en la optimización de Galil [20], que reduce el número de flujos calculados. Con esto, se logra una complejidad de $O(\text{máx}(k, \sqrt{n})\ k\ m\ \sqrt{n})$ (siendo n la cantidad de vértices y m la cantidad de aristas), que representa una mejora sustancial respecto a versiones anteriores. A día de hoy, existen otros algoritmos con mejor complejidad, pero para ser consistentes con los cálculos de complejidad realizados por Kanevsky vamos a tomar dicha definición como cota.

Con la conectividad calculada, se procede a tomar un subconjunto de vértices X de tamaño k, a partir del cual se comienza la búsqueda por la colección Γ_{st} de los conjuntos de tamaño mínimo de vértices separadores del grafo, entre pares de la forma (s,t) tales que $s \in X$ y $t \in V$. En este sentido, para este procedimiento, se va a iterar para todos los posibles s pertenecientes a X, y para cada s vamos a tomar un vértice t no adyacente a s. Por ello, para que este proceso sea más eficiente, vamos a conformar al conjunto X a partir de los vértices de G de mayor grado (es decir, con la mayor cantidad de vértices adyacentes). Antes de proceder a explicar que se hace con cada par (s,t), es importante notar lo siguiente: suponiendo

que ya hubiéramos encontrado todos los k-conjuntos de vértices que separan a s y t en G, podemos agregar una arista (s,t) a E sin modificar ninguno de los otros k-conjuntos de G. Esto es así porque, para cualquier otro conjunto separador Y que no aparte a s y t, ambos vértices van a permanecer en el mismo componente conexo del grafo 'separado', razón por la cual la adición de una arista directa entre los mismos no produce ninguna alteración. Por todo esto, para cada par (s,t), se agrega una arista al final de la iteración (lo que provoca que se agreguen, como máximo, kn aristas nuevas al grafo). Además, una vez finalizadas todas las iteraciones, se chequea también que el mismo conjunto X sea un conjunto separador.

Con todo esto en consideración, procedemos a detallar cómo se encuentra la colección Γ_{st} de conjuntos. Para la misma, vamos a hacer uso de la reducción de Even-Tarjan, que convierte este problema a la búsqueda de los cortes mínimos (s,t) de un grafo dirigido. Para ello, se construye un dígrafo $\bar{G} = (\bar{V}, \bar{E})$, con el siguiente procedimiento:

- Cada vértice $v \in V - \{s,t\}$ es reemplazado por dos vértices v' y v'' en $\bar{V}$, con una arista dirigida $(v', v'') \in \bar{E}$, con $c(v', v'') = 1$ (es decir, una capacidad de una unidad). Esto es así porque se busca que dichas aristas representen el 'costo' asociado a la eliminación de un vértice.
- Por cada arista $(u,v) \in E$, se agregan dos aristas (u'', v') y (u', v'') en $\bar{E}$, ambas con capacidad infinita.
- Por cada arista $(s,v) \in E$, se agregan dos aristas (s, v') y (v'', s) en $\bar{E}$; y por cada arista $(t,v) \in E$, se agregan dos aristas (v'', t) y (t, v') en $\bar{E}$, todas con capacidad infinita.

En la siguiente figura se muestra un ejemplo de grafo, y acto seguido su dígrafo correspondiente.

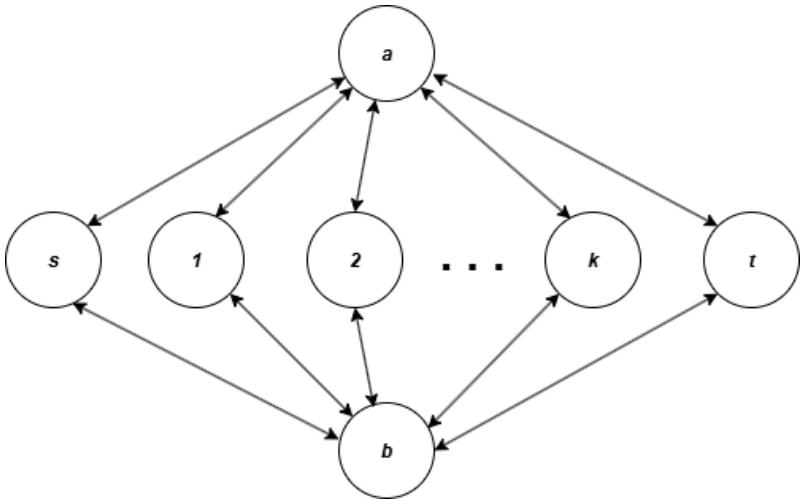

Figura 2.15: Grafo de ejemplo a utilizar.

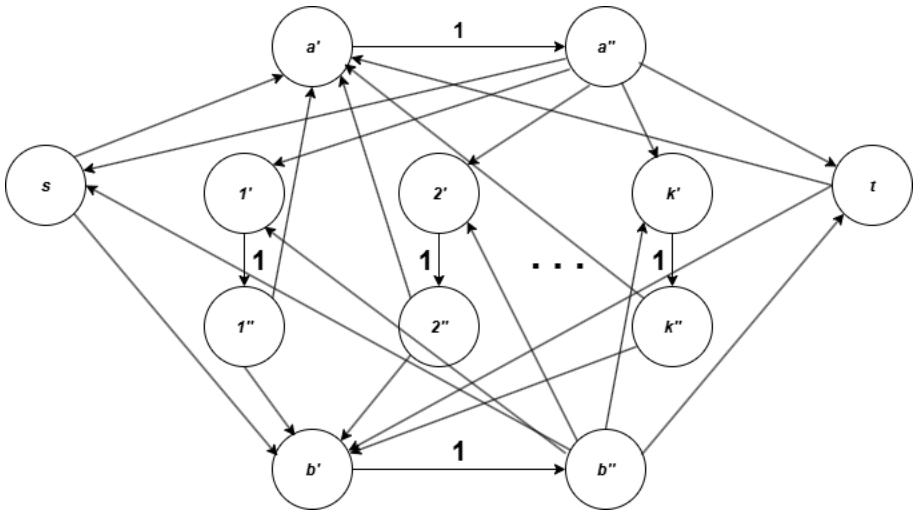

Figura 2.16: Dígrafo formado a partir del grafo de ejemplo.

En el dígrafo, todas las aristas en las que no se incluye su capacidad tienen capacidad infinita.

Con nuestro dígrafo $\bar{G}$ armado, el problema de encontrar Γ_{st} se corresponde a encontrar los cortes mínimos (s, t). Hay varios algoritmos que hacen esto, y que hacen uso de otro grafo auxiliar. En principio, debemos computar el grafo residual inverso de $\bar{G}$ con respecto a un flujo máximo desde s a t, al que llamaremos N_{st}. Acto seguido, vamos a comprimir las componentes fuertemente conexas de N_{st} en vértices únicos. Esto nos va a dar como resultado un grafo acíclico L_{st} al que llamaremos red auxiliar de G, cuyos vértices serán supervértices. Es importante notar que cada vértice de N_{st} se corresponde a un único supervértice de L_{st}.

En la siguiente figura, se muestra el grafo residual inverso N_{st} y la red auxiliar L_{st} del grafo expuesto en el ejemplo anterior.

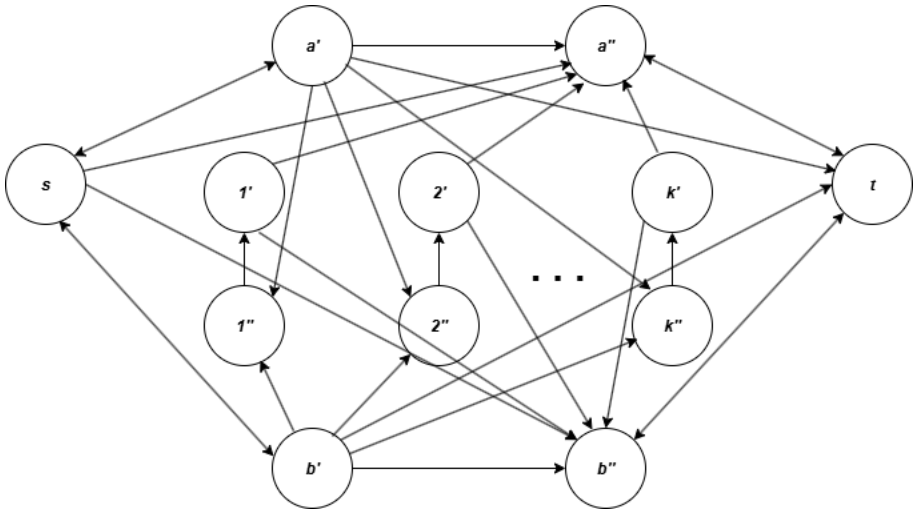

Figura 2.17: Grafo residual inverso del grafo de ejemplo.

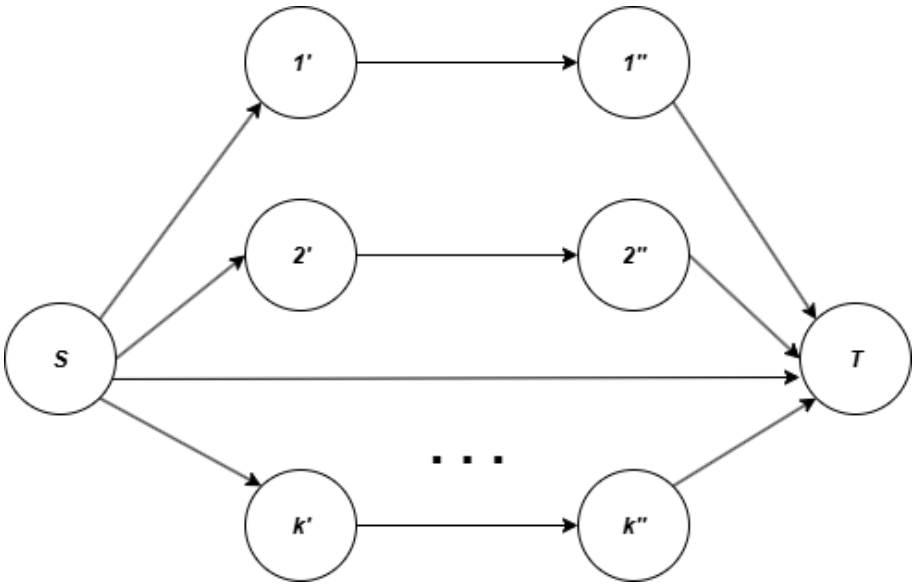

Figura 2.18: Red auxiliar del grafo residual inverso.

El grafo residual inverso de la figura 2.17 corresponde al flujo máximo a través de los caminos $s - a' - a'' - t$ y $s - b' - b'' - t$.

Ahora, vamos a llamar a un subconjunto C de vértices de L_{st} 'cerrado' si y solo si para cada vértice $v \in C$, todos sus predecesores también son miembros de C (es decir, todos aquellos vértices con aristas apuntando a v). Bajo esta definición, decimos que un corte $(S, \bar{S})$ que separa s de t en N_{st} es un corte mínimo solo si S es un conjunto cerrado que contiene a s y no a t. Es importante notar que la inversa no se cumple, pues no alcanza con que el conjunto sea cerrado para que sea mínimo. En el ejemplo de la figura podemos ver que el grafo G contiene un único conjunto mínimo de vértices separadores, compuesto por $\{a, b\}$. Pero, por otro lado, hay 2^k conjuntos cerrados X distintos que producen un corte en el grafo $\bar{G}$, todos conteniendo a $\{s, a', b'\}$ y distintas combinaciones del subconjunto dado por $\{v_1, v_2, ..., v_k\}$. Entonces, no podemos encontrar todos los conuntos cerrados de L_{st} porque eso requeriría tiempo exponencial, razón por la cual vamos a recurrir al método de Provan-Shier [12]. Para ello, nos vamos a basar en el siguiente lema: un conjunto C de aristas es un corte mínimo (s, t) de $\bar{G}$ si y solo si se puede expresar de la forma $C = (X, \bar{X})$ tal que:

- $s \in X$, $t \in \bar{X}$.
- X es un conjunto cerrado en $\bar{G}$.
- Para todo $x \in X$ hay un $v \in V(E_1)$ que es alcanzable en X desde x. Esto quiere decir que X no puede contener vértices aislados: todos deben estar conectados con alguna parte de la región activa del flujo.

Dado todo esto, X es el conjunto mínimo único tal que $(X, \bar{X}) = C$. Así, re-

ducimos el problema de encontrar los cortes mínimos (s, t) al problema de encontrar la colección Λ_{st} de conjuntos cerrados en L_{st} que satisfacen la condición del lema antes expuesto, sin repeticiones.

Kanevsky introduce el concepto de elemento pivote, como forma de recorrer sistemáticamente los conjuntos de Λ_{st}. Dado $X \in \Lambda_{st}$ y $v \in \bar{X}$, se define que v es un elemento pivote de X si $X \cup \{v\}$ está contenido en un elemento mínimo único $Y \in \Lambda_{st}$. El *conjunto pivote* asociado se define como:

$$I(X, v) = Y - X.$$

En este contexto, al hablar de "mínimo" nos referimos a la minimalidad con respecto a la inclusión dentro del conjunto parcialmente ordenado Λ_{st}, no al tamaño de los conjuntos (es decir, que no hay ningún conjunto $Z \in \Lambda_{st}$ tal que $X \subseteq Z \subset Y$). Intuitivamente, $I(X, v)$ representa el conjunto de vértices que deben añadirse a X para obtener el siguiente conjunto cerrado Y dentro de Λ_{st}. Por ejemplo, consideremos el conjunto $V = \{1, 2, 3, 4, 5\}$ y la colección I que comprende a los conjuntos $\{1, 2\}$, $\{1, 2, 3, 4\}$ y $\{1, 2, 3, 4, 5\}$. Tomemos $S = \{1, 2\}$; entonces, $\{1, 2, 3, 4\}$ es un elemento minimal de I tal que $S \cup \{3\}$, porque es el elemento más pequeño de I que los contiene. Entonces, $v = 3$ es un elemento pivote de S y $I(S, v) = \{3, 4\}$.

Basándose en el enfoque de Provan-Shier, Kanevsky propone un procedimiento recursivo que consiste en el descubrimiento de los pivotes para recorrer, sin duplicados, los elementos de Λ_{st}. A continuación, se muestra su pseudocódigo:

Algorithm 2: LIST(S,T) - construcción recursiva de Λ_{st}

Input: Subconjunto en construcción S, subconjunto descartado T

```
1 (v, I(S, v)) := PIVOT(S, T);
2 if I(S, v) = ∅ then
3 |   guardar S
4 LIST(S, T ∪ {v})                    # Rama que ignora v;
5 LIST(S ∪ I(S, v), T)                # Rama que agrega el conjunto pivote;
```

Aquí, S representa un subconjunto parcial en construcción y T es el conjunto de vértices descartados que ya fueron utilizados como pivotes y no deben ser usados nuevamente. La función `PIVOT` (sobre la cual ahondaremos más adelante) retorna un vértice pivote v y su conjunto $I(S, v)$. Si no existen pivotes válidos, el conjunto S se emite como un elemento completo de Λ_{st}, correspondiente a un corte mínimo (s, t).

Kanevsky demuestra que la validez y la complejidad del procedimiento expuesto depende de dos propiedades. Supongamos que la colección Λ_{st} de cortes tiene asociada una colección inicial de conjuntos Γ_{st} que satisfacen:

- **P1.** Existe una correspondencia uno-a-uno entre los cortes (s,t) mínimos $C \in \Gamma_{st}$ y los conjuntos iniciales $X \in \Lambda_{st}$, tal que:

$$X \in \Lambda_{st} \longleftrightarrow C = (X, \bar{X}) \in \Gamma_{st}$$

- **P2.** Para cada $X \in \Lambda_{st}$ y para todo subconjunto $S \subseteq X$, existe un vértice pivote v de S tal que $v \in X$.

Entonces, `LIST`$(\emptyset, \emptyset)$ lista correctamente los elementos de Λ_{st} y, por lo tanto, los elementos de Γ_{st} en tiempo igual a la complejidad de `PIVOT` por cada corte. La primera propiedad nos garantiza que los elementos se generan sin duplicados, mientras que la segunda propiedad nos permite listar los sets iniciales X listando recursivamente ciertas sub-colecciones de Λ_{st}, garantizando que siempre existe al menos un vértice dentro de X que permite extender el conjunto parcial S hasta reconstruir X completo.

La noción del elemento pivote es lo que permite construir de forma recursiva cada conjunto X, garantizando la cobertura total del espacio de soluciones, sin duplicados. Es interesante notar que S es el elemento pivote para $X = \emptyset$, dado que hay aristas que salen de S en L_{st} pero ninguna que entre.

Por último, y para poder finalmente listar los cortes mínimos, requerimos de una implementación de la función `PIVOT`, para lo cual Kanevsky propone lo siguiente: se define en el grafo acíclico L_{st} el conjunto de *vértices activos* A_0. Un vértice es activo si su componente de vértices asociada en N_{st} contiene al menos un elemento de $V(E_1)$ (es decir, aquellos supervértices de L_{st} por los que pasa flujo). Entre ellos, se considera el subconjunto $M_0 \subseteq A_0$ de vértices activos mínimos, según el orden topológico de L_{st}; es decir, ningún vértice de M_0 es alcanzable desde otro vértice de M_0. Si $M_0 \subseteq T$, entonces no existen pivotes y se devuelve $I(S, v) = \emptyset$. En caso contrario, se elige cualquier $v \in M_0 - T$ como pivote, y se construye el conjunto correspondiente $I(S, v)$ a partir de todos los vértices de L_{st} que pueden alcanzar a v:

$$I(S, v) = \{\, x \in V(L_{st}) : x \rightsquigarrow v \text{ en } L_{st} \,\}.$$

Este procedimiento permite hallar cada conjunto pivote en tiempo $O(n)$. Esto significa que cada corte (s,t) en $\bar{G}$ puede ser generado en tiempo $O(n)$.

Así, ya con los cortes del grafo auxiliar L_{st} listados, se traduce cada conjunto X al conjunto de aristas que forma el corte $(X, \bar{X})$ en el grafo dirigido G_{st}. Luego, se mapean esas aristas al conjunto de vértices correspondiente del grafo original G, y nos quedamos únicamente con los conjuntos de tamaño k, que son los que estamos buscando.

A continuación, se procede a listar un pseudocódigo que resume el algoritmo de Kanevsky:

Algorithm 3: Algoritmo de Kanevsky para encontrar todos cortes mínimos de nodos

Input: Grafo no dirigido $G = (V, E)$

Output: Todos los k-conjuntos mínimos que separan G

```
1  Calcular la conectividad k de G (los vértices de G están numerados
     v_1, v_2, ..., v_n);
2  Seleccionar los k vértices con mayor grado (x_1, x_2, ..., x_k);
3  Verificar si estos k vértices forman un k-conjunto separador de G;
4  for i ← 1 to k do
5      for j ← 1 to n do
6          if v_j ≠ x_i y v_j no es adyacente a x_i then
7              Definir s ← x_i y t ← v_j;
8              Calcular un flujo máximo φ en Ḡ_st desde s hasta t;
9              if |φ| = k then
10                 Construir el grafo residual inverso N_st con respecto al flujo
                     máximo φ;
11                 Contraer las componentes fuertemente conexas de N_st para
                     formar el grafo auxiliar L_st;
12                 Encontrar todos los conjuntos cerrados X en L_st que satisfacen
                     las condiciones del lema de Provan-Shier;
13                 Para cada conjunto cerrado X, hallar el conjunto de vértices
                     correspondientes en G y tomar el correspondiente k-conjunto
                     separador;
14         Añadir la arista (x_i, v_j) a G;
```

Para terminar el ejemplo gráfico que fue desarrollado en las figuras anteriores, vamos a tomar la figura 2.18. Sobre la misma, y a partir de la instrucción `LIST`, vamos a listar todos los conjuntos cerrados X pertenecientes a Λ_{st}. Con `LIST`$(\emptyset, \emptyset)$, la función `PIVOT` nos va a devolver el supervértice s como pivote. Ahora, si volvemos a ejecutar la recursión, por el camino `LIST`$(\{s\}, \emptyset)$, vamos a notar que no se obtiene ningún pivote nuevo, dado que por el resto de supervértices no corre flujo (pues, recordemos, el flujo máximo del grafo $\bar{G}$ solo corre a través de los vértices a y b hasta llegar a t). Por todo esto, el único conjunto X obtenido está dado por el supervértice s, tal que $X = \{s, a', b'\}$. Dado todo esto, solo queda mapear la colección obtenida a un conjunto de vértices del grafo original G. Teniendo en cuenta que la conectividad del grafo original G es de 2, y como el corte obtenido es $(X, \bar{X}) = \{(a', a''), (b', b'')\}$, sabemos que el mismo se corresponde con el conjunto original de vértices $\{a, b\}$,

que a su vez corresponde al k-conjunto mínimo separador obtenido para el par dado (s, t).

A fin de terminar este análisis, sólo resta evaluar la complejidad del algoritmo dado. Sabemos que la complejidad del primer paso, en el que calculamos la conectividad, es de $O(\text{máx}(k, \sqrt{n})\ k\ m\ \sqrt{n})$. El segundo paso, en el que seleccionamos los vértices con mayor grado, toma $O(m+n)$ en complejidad, porque debemos recorrer todo el grafo. Luego, pasan a realizarse dos iteraciones anidadas, que determinan que todos los pasos a continuación se repitan kn veces:

- Primero, se hacen chequeos de adyacencia e igualdad entre dos nodos, lo que incurre en un tiempo constante.
- Luego, se calcula el flujo máximo para el grafo. Kanevsky propone dos algoritmos distintos para esto, uno con cota $O(\text{mín}(k, \sqrt{n})(m+1))$, y otro con cota $O(\text{mín}(k(m+n), A))$, siendo A el valor del mejor flujo máximo en una red unitaria.
- Acto seguido, se construye el grafo residual y se contraen las componentes fuertemente conexas. Para ambos pasos, existen algoritmos con complejidad lineal $O(m+n)$.
- Para encontrar los conjuntos cerrados, vamos a definir M_{st} como el número de cortes entre un par (s, t). Como ya dijimos antes, cada corte requiere de una complejidad $O(n)$ para ser definido.

En definitiva, encontrar los cortes entre un par (s, t) incurre en una complejidad de $O(M_{st}n+C)$, con C la complejidad del algoritmo de flujo máximo. Yendo más allá, podemos definir M como el número total de cortes, tal que $M = \sum_{i=1}^{k} \sum_{j=1}^{n} M_{x_i,v_j}$. Entonces, la complejidad para encontrar todos los k-conjuntos separadores es de $O(Mn+knC)$. Kanevsky toma la definición $M = O(2^k(n^2/k))$ de uno de sus trabajos anteriores [13], para lo que termina tomando, finalmente, una cota superior de $O(v^3)$ en el peor de los casos.

2.4.3. Coloreo de grafos

Como Jensen y Toft explican [9], el problema del coloreo de grafos forma parte del largo grupo de procesos matemáticos relacionados con la necesidad de particionar un grupo de objetos en distintas clases según reglas determinadas. En el mismo, queremos separar los vértices pertenecientes al set V del grafo G, siendo que un par de vértices que están unidos por una arista incluida en E en dicho grafo no pueden pertenecer a la misma clase. Para distinguir las clases, usamos un set de colores C, y la división en clases está dada por un coloreo $\varphi : V(G) \to C$, donde

$\varphi(x) \neq \varphi(y)$ para todo $(x, y) \in E(G)$. De esta forma, cada grupo de colores forma un set independiente de vértices, en el que ningún par se encuentra unido directamente por una arista. El coloreo de grafos normalmente es utilizado para resolver problemas de *scheduling*, en los que se nos da un set de trabajos que tienen que ser realizados por ciertos individuos, con la restricción de que algunos trabajos no pueden ser realizados de forma simultánea porque deben ser realizados por el mismo agente. Si lo pensamos, este problema es muy similar a lo que buscamos resolver.

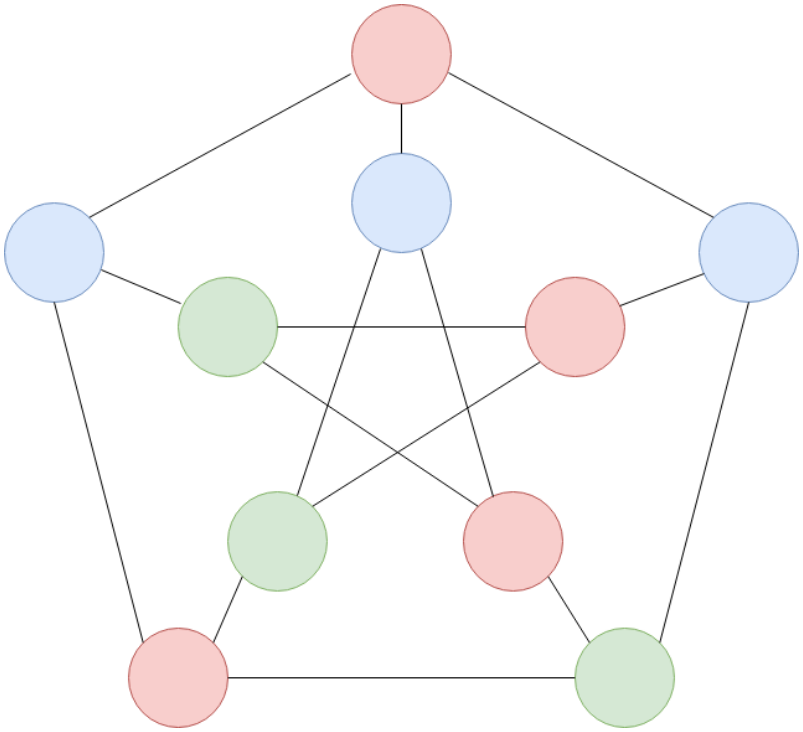

Figura 2.19: Coloreo válido en un grafo de ejemplo.

Se define al número cromático $\chi(G)$ de un grafo G como el mínimo número de colores que puede ser utilizado para colorear los nodos; es decir, representa la cantidad de colores que requeriría un coloreo óptimo del grafo. Por otro lado, el grado de un vértice v perteneciente a un grafo G es la cantidad de vecinos de dicho vértice. Definiendo a $\Delta(G)$ como el máximo grado de un grafo G, sabemos que es válida la siguiente relación: $\chi(G) \leq \Delta(G) + 1$. La misma nos indica que el número cromático está acotado por el máximo grado del grafo.

Como explican Garey y Johnson [15], desde el punto de vista computacional, encontrar un coloreo óptimo es un problema notoriamente difícil: ha sido demostrado que es **NP-Hard**, lo que implica que no se conoce ningún algoritmo que resuelva este problema en tiempo polinomial para el caso general. En consecuencia, se recurre comúnmente a algoritmos heurísticos que, si bien no garantizan la optimalidad, ofrecen soluciones cercanas al óptimo en un tiempo razonable. Dicho esto, ahora vamos a proceder a explicar algunos de estos algoritmos.

Algoritmo de Backtracking

Brown presenta un algoritmo basado en *backtracking* para encontrar el número cromático exacto de un grafo [17], lo cual nos ofrece a su vez un coloreo óptimo. A diferencia de los enfoques greedy que buscan una coloración válida sin garantizar optimalidad, como los que veremos a continuación, este algoritmo explora todas las posibles asignaciones de colores haciendo uso de una 'poda', tal que solo van a ser consideradas aquellas configuraciones que puedan conducir a una solución válida.

Dicho algoritmo consiste en colorear los nodos del grafo uno por uno, probando colores en orden y retrocediendo en caso de que no se pueda continuar; es decir, si se llega a un estado sin solución válida. Para limitar el tiempo de búsqueda, se podan las posibles soluciones considerando la detección de conflictos y la comparación con la mejor solución encontrada hasta el momento.

Algorithm 4: Algoritmo de Backtracking para coloreo

Input: Grafo $G = (V, E)$

Output: Asignación óptima de colores a los nodos

```
1  for k = 1 hasta n do
2      Inicializar arreglo coloreo[v_i] ← color nulo para todo v_i;
3      if colorear(v_1, k, coloreo) then
4          return color

5  Función colorear(nodo, k, coloreo):
6      if nodo > n then
7          return true
8      for c = 1 hasta k do
9          if ningún vecino de v_nodo tiene color c then
10             coloreo[v_nodo] ← c;
11             if colorear(nodo + 1, k, coloreo) then
12                 return true
13             coloreo[v_nodo] ← sin color
14     return false
```

Este algoritmo garantiza la obtención de una solución óptima, es decir, una coloración con el menor número posible de colores. De todas maneras, su complejidad computacional es exponencial, siendo de $O(k^v)$, pues el espacio de soluciones a explorar es enorme. Si bien se puede mejorar mediante el ordenamiento inicial de vértices (como muestran algunos papers más novedosos), la realidad es que la complejidad sigue siendo muy alta.

Por esta razón, el algoritmo de Brown suele ser muy útil en contextos en los que se requiere de una solución exacta y el tamaño del grafo es relativamente pequeño, como en el caso de algunas topologías reducidas o en la comparación contra soluciones heurísticas utilizando grafos de ejemplo.

Algoritmo de Welsh-Powell

Welsh y Powell [16] presentan una estrategia greedy para el problema del coloreo óptimo. La idea del mismo se basa en recorrer los vértices del grafo en orden, de manera decreciente según su grado, y asignar el menor color disponible que no esté siendo utilizado por sus vecinos. Este enfoque se basa en la intuición de que los vértices con mayor grado tienen más restricciones (pues tienen más vecinos con los que deben evitar compartir color), razón por la cual deben colorearse primero.

Algorithm 5: Algoritmo de Welsh-Powell

Input: Grafo $G = (V, E)$

Output: Asignación de colores a los nodos

```
1 Ordenar los nodos v_1, v_2, ..., v_n por grado en orden no creciente;
2 Inicializar todos los nodos como sin color;
3 color ← 1;
4 while hay nodos sin colorear do
5     for nodo v_i en el orden dado do
6         if v_i no está coloreado y ningún vecino tiene el color actual then
7             Asignar color c a v_i;
8     color ← color + 1;
```

Si bien este algoritmo no genera una coloración óptima (pues sigue tratándose de una aproximación), el mismo siempre da una coloración válida que se encuentra acotada por el grado máximo del grafo más uno. La complejidad de este algoritmo es de $O(v^2)$, pues, en el peor caso, se debe realizar un recorrido a lo largo de la lista de grados n veces. El ordenamiento de los vértices según el grado puede realizarse en $O(v \log v)$, razón por la cual no influye en la complejidad.

Por todo lo expuesto, ese algoritmo se destaca como una opción simple y eficiente para grafos de tamaño moderado.

Algoritmo D-Satur

Brélaz presenta en 1979 el algoritmo de D-Satur [8], llamado D-Satur por *Degree of Saturation* (grado de saturación), que se convirtió en una de las heurísticas

más conocidas a día de hoy para el problema del coloreo óptimo. Corresponde a una mejora del enfoque tradicional de Welsh-Powell.

Debemos definir el grado de saturación de un vértice v, que refiere a la cantidad de colores distintos utilizados por los vecinos de v. Este algoritmo se basa en la idea de seleccionar en cada paso el vértice con mayor grado de saturación, con el objetivo de colorear en primer lugar aquellos vértices con mayores restricciones, para reducir la posibilidad de conflictos. En caso de empate por la saturación, se utiliza como criterio el grado del vértice (como el algoritmo original).

Algorithm 6: Algoritmo D-Satur

Input: Grafo no dirigido $G = (V, E)$

Output: Función de coloreo $c : V \to \mathbb{N}$ que asigna un color a cada vértice

```
1 Inicializar c(v) ← sin color para todo v ∈ V;
2 Inicializar saturación(v) ← 0 para todo v ∈ V;
3 while existe algún vértice sin colorear do
4     Seleccionar el vértice v no coloreado con mayor saturación;
5     if hay empate then
6         Elegir el vértice con mayor grado (cantidad de vecinos);
7     Asignar a v el menor color que no esté usado por sus vecinos;
8     Actualizar la saturación de los vecinos no coloreados de v;
```

La complejidad de este algoritmo depende de su implementación. Si usamos un heap para la selección de los vértices de máxima saturación, podemos conseguir que dicha operación se realice en tan sólo $O(\log v)$. Por otro lado, la actualización de la saturación en cada iteración puede tomar $O(v)$. En el peor de los casos, debemos repetir estas iteraciones v veces, dado lo cual su complejidad es de $O(n^2)$. Incluso usando algunas estructuras más avanzadas, se puede conseguir una complejidad de $O(m + n \log n)$. Este algoritmo tampoco garantiza una coloración óptima, pero suele ser mucho más eficaz que el de Welsh-Powell, sobretodo en grafos densos, pues dicho algoritmo no tiene en cuenta la evolución del estado del grafo a lo largo del proceso de coloreo. Además, en lo que respecta a complejidad, es sustancialmente más rápido que algoritmos como el de *backtracking*, que en caso de tener topologías de muchos vértices se vuelve completamente inutilizable, más allá de que consiga la solución óptima. Por todo lo expuesto, y considerando su balance entre eficiencia y proximidad de la solución, se estará tomando el algoritmo de D-Satur para el desarrollo de esta tesis.

Capítulo 3
Reinicio por coloreo de grafos

En este capítulo se presenta la solución propuesta desarrollada a lo largo de esta tesis, y los conceptos en los que la misma se basa. Además, se realiza un análisis computacional de la misma, en base a los algoritmos de los que se sirve, y se ponderan sus ventajas y limitaciones.

3.1. Definición de la solución propuesta

La solución que se propone al problema exhibido deriva del dilema del **coloreo de grafos**, que fue expuesto en el capítulo anterior. El modelado de la red como un grafo permite aprovechar propiedades conocidas de la teoría de grafos que explicamos previamente. Además, el paralelismo con este problema se vuelve muy útil para este tipo de redes, y en particular para el contexto en el que queremos trabajar. Debemos tener en cuenta que el reinicio de un nodo podría dar lugar, dentro de la red, a que los nodos cambien de capa, tal como se definió en la sección 2.2.1, en función a la cantidad de saltos que debe hacer cada mensaje para llegar hasta la raíz del grafo. En este sentido, optamos por servirnos de un problema algorítmico que no se base en la diferenciación estricta entre las capas del grafo, pues podría ser una estrategia muy limitante a la hora de formar los grupos en que los distintos reinicios se van a llevar a cabo. De esta manera, la solución propuesta contempla la posibilidad de que un nodo modifique su rango dentro del DODAG, aumentando la distancia lógica del mismo respecto de la raíz en pos de preservar la conectividad de la red.

Para ilustrar esto, se muestra el ejemplo de la figura 3.1:

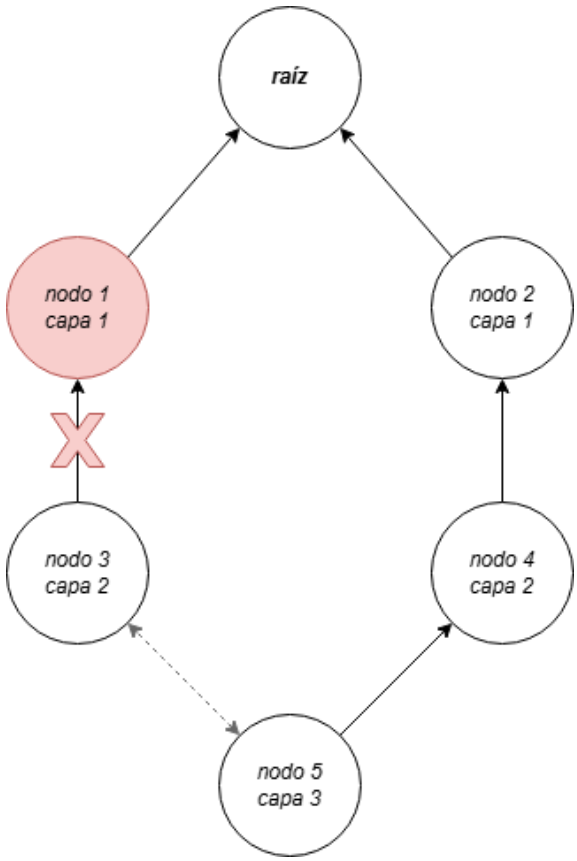

Figura 3.1: El nodo 1 cesa su funcionamiento, dejando al nodo 3 sin un padre preferido.

Como podemos ver en el mismo, el funcionamiento del nodo 1 se detuvo, dejando al nodo 3 sin conectividad con la raíz. En este caso, el nodo 3 no posee conectividad con ningún otro nodo de capa 1, y el único par al que puede conectarse directamente después de esto es al nodo 5, que forma parte de la capa 3. Evidentemente, y al menos hasta que el nodo 1 retome sus actividades, es preferible que el nodo 3 deba realizar una mayor cantidad de saltos para llegar a la raíz en vez de quedar completamente aislado de la red, y que su reconexión con el DODAG sea lo más rápida posible con tal de evitar la pérdida de paquetes. En este sentido, dadas situaciones como estas, se busca que la solución apueste por una ágil reorganización del DODAG que permita al nodo 3 conectarse lo antes posible al nodo 5, cómo se muestra en la figura 3.2:

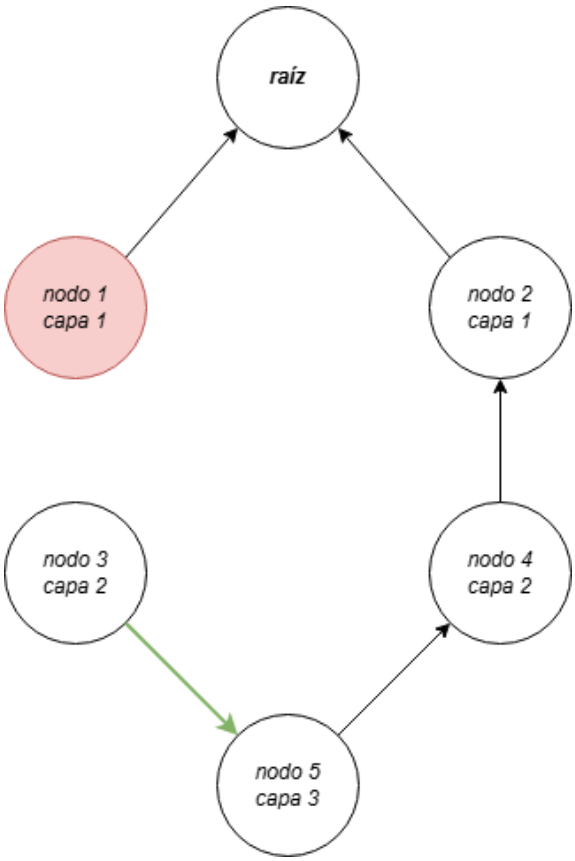

Figura 3.2: El nodo 3 encuentra un nuevo camino hacia la raiz mediante el nodo 5, de una capa inferior.

El algoritmo propuesto se divide conceptualmente en dos etapas principales: (i) asignación de *batches* mediante análisis estructural del grafo y (ii) ejecución temporal coordinada del reinicio. A continuación, se muestra el mismo en forma de pseudocódigo:

Algorithm 7: Algoritmo de planificación y ejecución de rebooting

Input: Topología de red con información de alcance

Output: Asignación de batches y ejecución coordinada

```
Construir grafo G = (V, E) a partir de la topología;
Eliminar nodos hoja recursivamente;
Detectar puntos de articulación en G;
Dividir G en subgrafos independientes;
foreach subgrafo G_i do
    Aplicar D-Satur para colorear G_i;
    while el coloreo genera desconexiones en G_i do
        Calcular cortes mínimos;
        Agregar aristas imaginarias;
        Recolorear el subgrafo;
Reincorporar nodos hoja con el color de su padre;
Agrupar nodos por color;
foreach batch i do
    Calcular T_i = (RT_max + CT + c) · i;
    Propagar T_i desde la raíz;
    Al alcanzar T_i, enviar Infinite Rank DIO;
    Ejecutar reboot;
```

Ahora, se procederá a explicar y justificar cada uno de los pasos. Para ello, comenzaremos por el proceso de asignación de *batches*:

1. Para empezar, debemos **crear un grafo a partir de la topología** en cuestión; en particular, no vamos a hacer uso del DODAG en sí. Dicho grafo va a poseer como nodos a cada uno de los dispositivos de la red, y va a existir una arista entre dos nodos sólo si los mismos pueden enviarse mensajes directamente entre sí (es decir, sin ningún intermediario). Esto requiere, claro, que tengamos información sobre el radio de transmisión de los dispositivos utilizados.

2. **Eliminamos los nodos hoja** de dicho grafo, de forma recursiva hasta que no quede ningún nodo hoja dentro del mismo o hasta llegar a la raíz.

3. Buscamos todos los **puntos de articulación** en el grafo, tal como fueron definidos en la sección 2.4.1, utilizando el algoritmo allí presentado, y **dividimos el grafo** en los múltiples subgrafos que se derivarían de separar dichos puntos de articulación del grafo original. Como mencionamos anteriormente, estos nodos constituyen los cuellos de botella y el obstáculo principal con el que nos encontramos a la hora de resolver este problema: su eliminación su-

pone la desconexión, indefectiblemente, de varios nodos. En vista de esto, se puede proponer una solución muy evidente: reiniciar, al mismo tiempo, todo el subgrafo que se deriva de cada punto de articulación. Esto puede ser muy bueno para evitar la pérdida de paquetes, pero puede ir en contra de los Service Level Agreements al tener un porcentaje muy grande de nodos caídos al mismo tiempo en una determinada zona geográfica, lo cual queremos evitar. Por eso, optamos por esta estrategia.

4. Aplicamos un **algoritmo de coloreo** a cada uno de los subgrafos...

   a) Si al reiniciar los dispositivos según dicho coloreo se producen desconexiones en el subgrafo, buscamos los **cortes mínimos**, agregamos una arista entre dos de los nodos en esos cortes mínimos, y repetimos el proceso de coloreo. Al agregar una arista 'imaginaria', evitamos que todos los nodos de dicho corte terminen con el mismo color y, consecuentemente, que se reinicien al mismo tiempo. Como se deriva de su definición, si reiniciáramos todos los nodos de un mismo corte a la vez, estaríamos produciendo indefectiblemente desconexiones en el grafo.

   b) Si al reiniciar los dispositivos según dicho coloreo no se producen desconexiones en el subgrafo, significa que **tenemos un coloreo válido** de nuestro grafo.

5. Agregamos a cada subgrafo los nodos hoja que previamente habían sido eliminados, y les asignamos un **color idéntico al de sus padres** correspondientes. De esta forma, estamos evitando que dichos nodos atraviesen un período de desconexión en el momento en que su único camino posible hacia la raíz sea eliminado, pues apuntamos a que tanto hijo como padre lleven a cabo su reinicio al mismo tiempo.

Una vez terminado este proceso, tendremos asignado un color a cada uno de los dispositivos participantes de la red, el cual representará el '*batch*' o grupo junto con el cual llevará a cabo su reinicio al mismo tiempo. Ahora, sólo queda llevar a cabo el proceso de *rebooting* en sí. Para esto, vamos a realizar el siguiente procedimiento:

1. El nodo raíz se va a encargar de **propagar a todos los nodos** la información necesaria para llevar a cabo el reinicio, que corresponde al *time-stamp* asignado al *batch* al que pertenece dicho nodo. Dicho *time-stamp* será diferente para cada grupo, con el propósito de esperar un cierto tiempo antes de continuar entre cada reinicio. A efectos prácticos, el cálculo de cada *time-stamp* podría realizarse de manera arbitraria en función de la estrategia que se requiera para este proceso; quizás algunos consideren preferible actualizar todos los nodos lo antes posible a costa de un porcentaje ligeramente mayor de pérdida

de paquetes, mientras que otros opten por *time-stamps* más espaciados entre sí con tal de cerciorarse que todos los nodos tengan un 'padre preferido' en RPL antes de continuar con las actualizaciones. Pero independientemente de ello, es necesario conocer de antemano el tiempo que toma un dispositivo genérico en reiniciarse, pues si la diferencia entre *time-stamps* fuera menor a dicho tiempo la solución propuesta carecería de utilidad, pues habría más de un *batch* produciéndose a la vez y no podríamos garantizar rutas disponibles hacia la raíz para los nodos, como estamos buscando. Para el caso particular de nuestra solución, se determinaron los *time-stamps* de la siguiente manera:

$$T_i = (RT_{max} + CT + c) * i$$

Según la fórmula expuesta, el *time-stamp* va a corresponder a la suma entre la cota máxima del tiempo de reinicio de un nodo, el tiempo de convergencia de RPL para la red en cuestión, y una constante real positiva arbitraria. Respecto al tiempo de convergencia, vamos a suponer que ya tenemos dicha información previa sobre la red de antemano, pues podemos obtenerla calculando el tiempo que toma la misma en que todos los nodos posean un 'padre preferido' desde el momento en que la red se pone en funcionamiento.

2. Al llegar al turno de un *batch* en específico, los nodos pertenecientes al mismo se encargan de **enviar un Infinite Rank DIO** a cada uno de los nodos en su rango de transmisión. Esto se realiza con el objetivo de ejecutar una reparación local de aquellos 'hijos' que hayan tomado como 'padre preferido' a aquellos nodos que estén a punto de llevar a cabo su actualización de firmware, de forma que actualicen su *path* hacia el nodo raíz a través de otro 'padre candidato'. De esta manera, evitamos que se pierda la mayor cantidad de paquetes posibles, pues cada nodo activo es consciente de los nodos 'envenenados' en la red que no pueden continuar propagando mensajes.

De esta forma, el máximo posible de nodos va a tener siempre un camino disponible hacia la raíz durante el proceso de *rebooting*. La excepción de ello van a ser los nodos que posean en su camino uno o más **puntos de articulación del grafo**, los cuales inevitablemente van a sufrir de pérdida de paquetes. De todas maneras, para el marco de esta investigación, asumimos que van a ser pocos los casos en los que tengamos que lidiar con topologías de red que tengan esa particularidad, dado que los dispositivos Wi-SUN cuentan con un rango de transmisión de hasta 1 kilómetro y, además, que se pretende que haya varios de estos dispositivos en un área de ese tamaño.

3.1.1. Ejemplo gráfico

A continuación, se muestra un ejemplo del funcionamiento del algoritmo de coloreo en acción a partir de la siguiente topología ilustrativa:

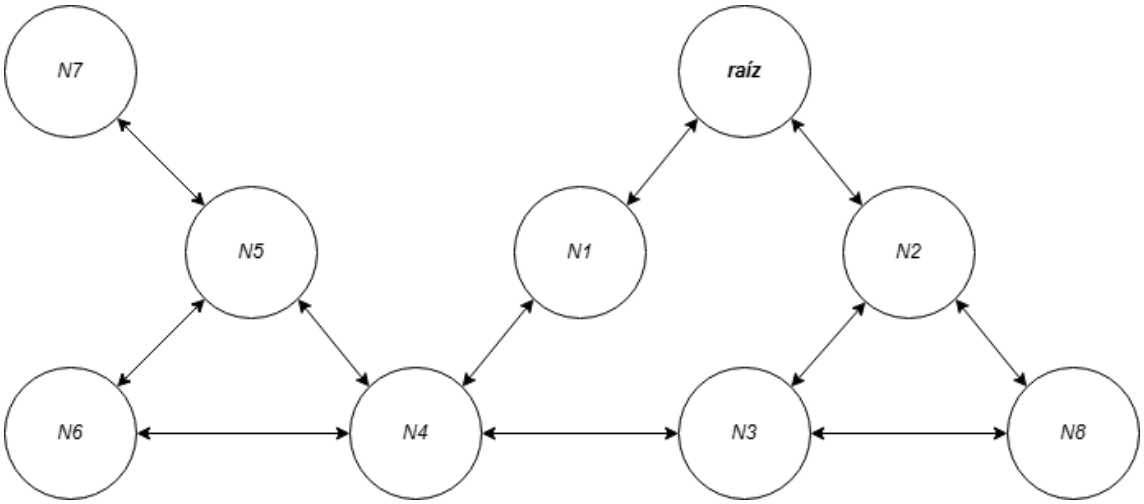

Figura 3.3: Topología de ejemplo, compuesta por 9 nodos.

En el grafo mostrado, las aristas representan la posibilidad de conectarse directamente entre pares.

Dicho esto, se procede a mostrar un seguimiento paso a paso de la solución propuesta. En primer lugar, vamos a eliminar los nodos hoja presentes en el mismo, que en este caso corresponde únicamente al nodo 7. Luego, procedemos a identificar los puntos de articulación del grafo. Como podemos ver, si eliminamos el nodo 4, generamos la desconexión de la red de los nodos 5 y 6, lo que nos da la pauta de que dicho nodo corresponde a un punto de articulación.

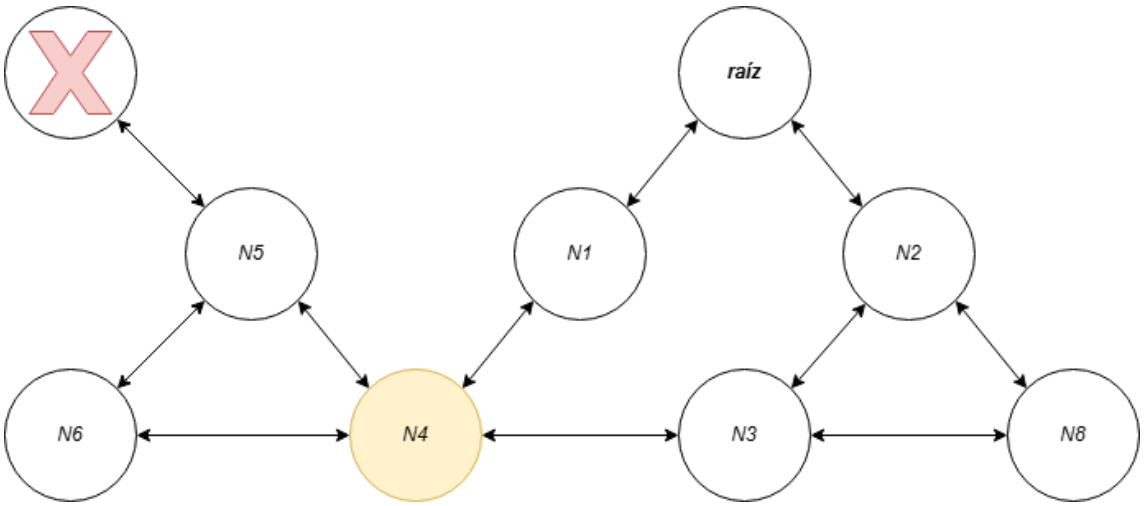

Figura 3.4: Eliminación de los nodos hoja e identificación de los puntos de articulación.

Con los puntos de articulación identificados, podemos proceder a separar el grafo en los subgrafos correspondientes, en este caso sólo dos. Como podemos ver, en el nuevo subgrafo creado, el nodo 4 se convierte en la nueva 'raíz virtual' del mismo, y por tal razón no va a recibir un coloreo en dicho subgrafo, sino en el subgrafo original al que pertenecía. Esto es así porque queremos evitar que se produzcan más tiempos de desconexión de los necesarios.

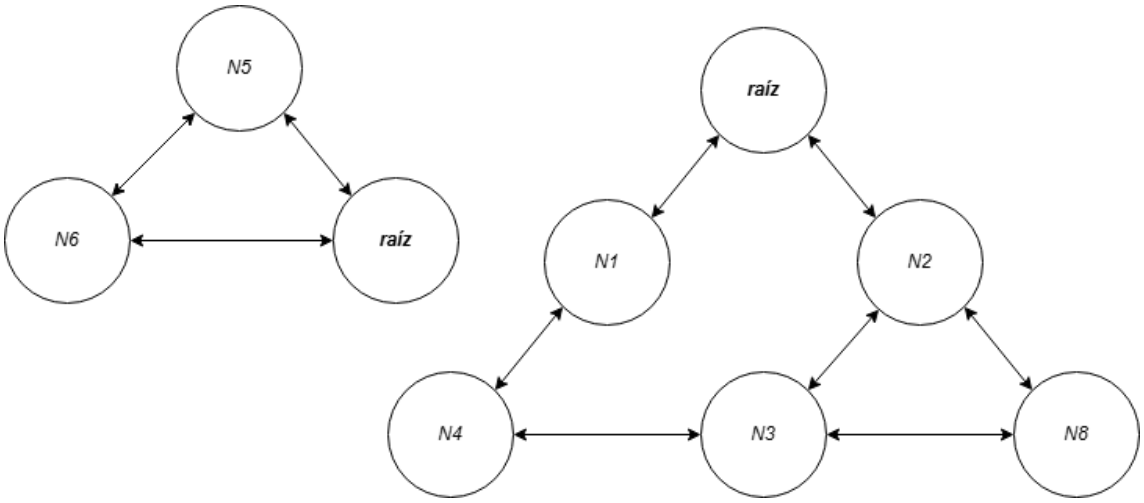

Figura 3.5: Separación en subgrafos según puntos de articulación.

Una vez separados los subgrafos, podemos proceder a ejecutar el algoritmo de coloreo seleccionado. Recordemos que, haciendo uso de un algoritmo de coloreo por aproximación, el resultado no necesariamente va a ser óptimo. De todas maneras, y con el fin de la ilustración, se muestra el siguiente coloreo óptimo de los subgrafos:

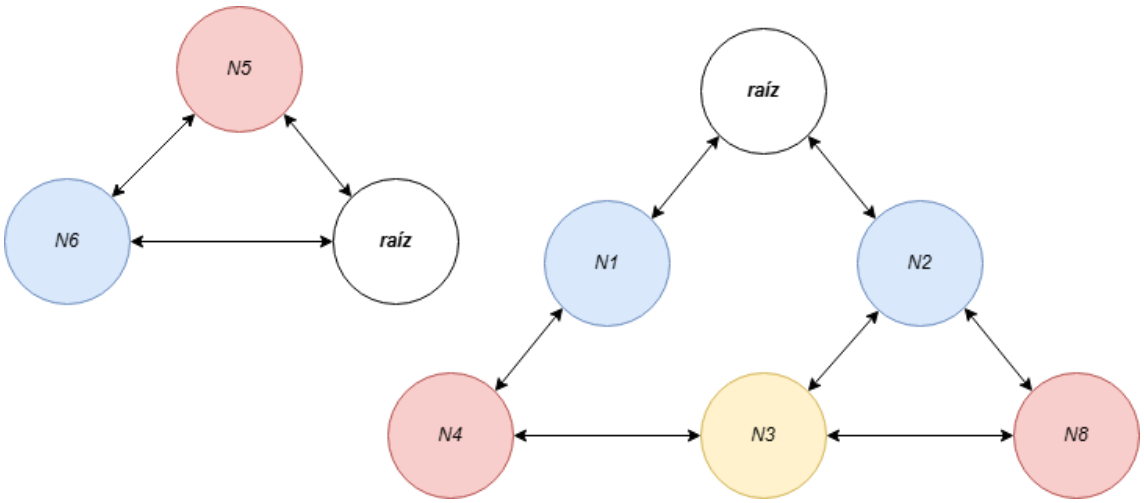

Figura 3.6: Coloreo óptimo de los subgrafos obtenidos.

Ahora, imaginemos que vamos a llevar a cabo el proceso de reinicio, y comenzamos con los nodos de color azul. Rápidamente, vamos a notar que al desconectar los nodos 1 y 2 al mismo tiempo, se produce la desconexión de los nodos 3, 4 y 8:

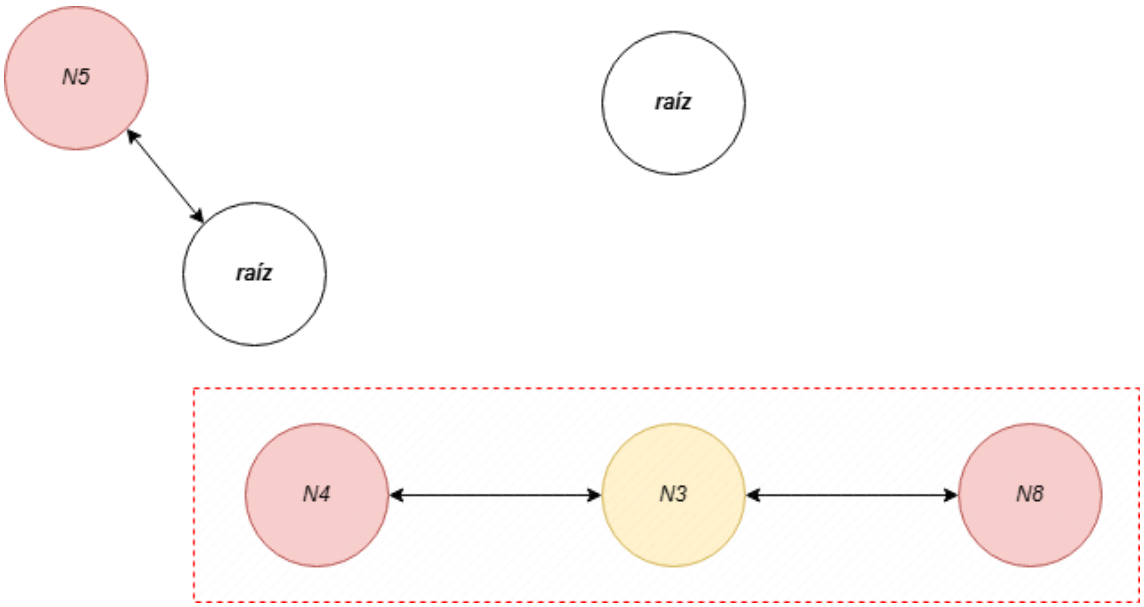

Figura 3.7: El reinicio de los nodos azules produce la desconexión de 3 nodos.

Como el coloreo conseguido produce la desconexión de los nodos, vamos a determinar los cortes mínimos en el subgrafo correspondiente. Usando el algoritmo correspondiente, inmediatamente vamos a notar que los nodos 1 y 2 corresponden a un corte mínimo, lo que nos da la pauta de que tenemos que agregar entre ellos una arista virtual, que indique que ambos nodos no pueden ser reiniciados al mismo tiempo. La misma situación ocurre para los pares 2 y 4 (que desconectan a los nodos 3 y 8), y 1 y 3 (que desconectan al nodo 4), razón por la cual agregamos aristas para ellos también. Una vez hecho esto, el coloreo es llevado a cabo nuevamente:

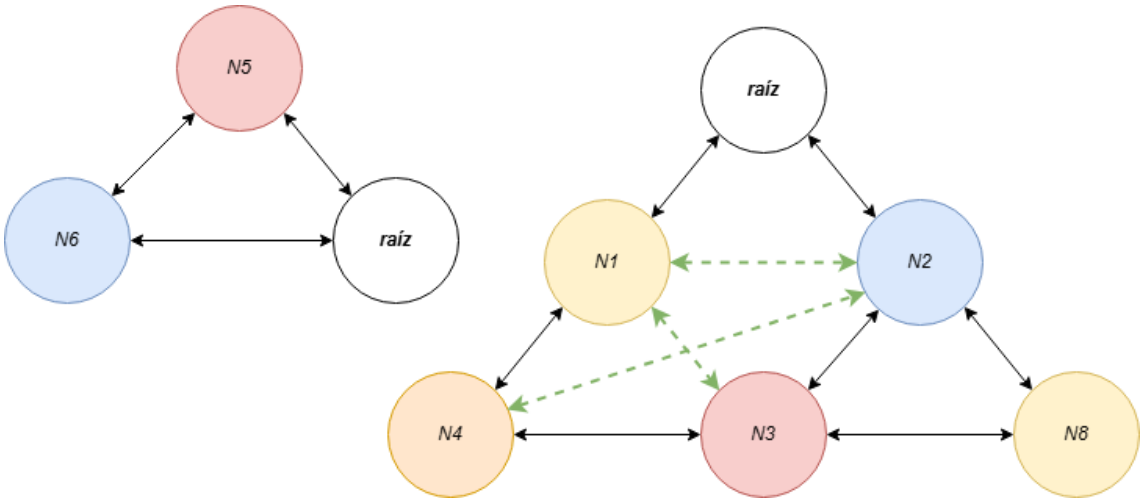

Figura 3.8: Agregado de una arista virtual y re-coloreo del subgrafo correspondiente.

Una vez con este coloreo, vamos a notar que ya no se produce ningún tipo de desconexión en el grafo a la hora de llevar a cabo reinicios, lo cual nos da la pauta de

que tenemos un coloreo válido para los subgrafos. Con esto, solo nos resta agregar los nodos hoja que habíamos eliminado anteriormente y colorearlos del mismo color que su padre correspondiente (por lo cual, el nodo 7 pasa a ser rojo), y volver a unir el grafo, lo cual queda de la siguiente manera:

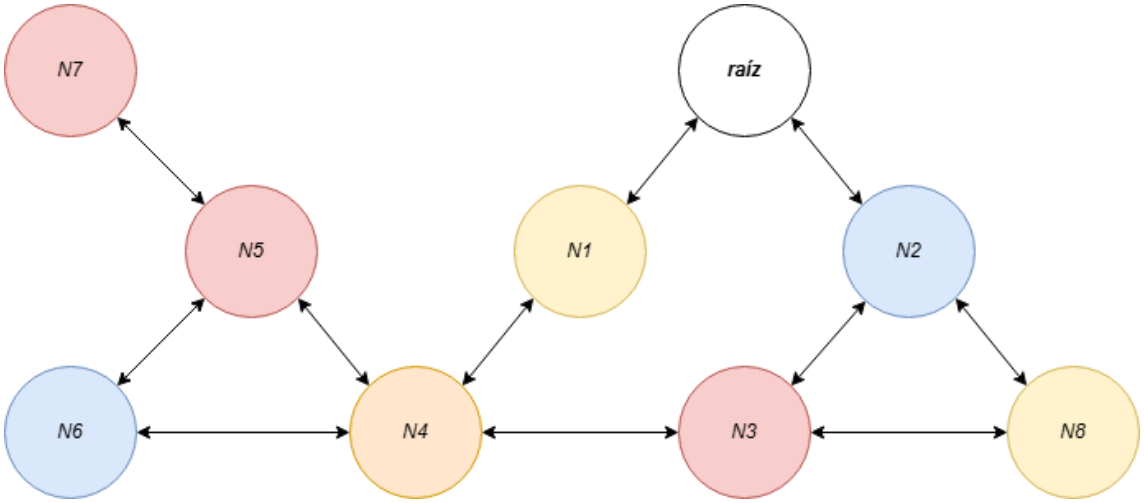

Figura 3.9: Grafo completo coloreado según la solución propuesta.

Con esto, ya tenemos completada la asignación de los distintos *batches*, y solo queda realizar la propagación de los mensajes para indicar el *time-stamp* correspondiente a cada nodo.

3.2. Complejidad computacional

Para analizar la complejidad computacional de la solución propuesta, se modeló la topología de la red como un grafo simple no dirigido $G = (V, E)$, donde V representa el conjunto de nodos de la red y E el conjunto de enlaces entre ellos. Denotamos con $v = |V|$ la cantidad de vértices y, con $e = |E|$, la cantidad de aristas del grafo.

Siguiendo esta notación, se analiza a continuación la complejidad temporal de cada etapa del algoritmo, estimando su comportamiento en el peor caso y respetando su orden de ejecución correspondiente:

1. **Construcción del grafo a partir de la topología**: para construir el grafo, debemos realizar un recorrido por todos y cada uno de los nodos y vértices de la topología en cuestión. Si suponemos que la red posee v nodos y e aristas, y suponiendo que hacemos uso de una clásica búsqueda DFS [28], la complejidad de este paso corresponde a

$$\mathcal{O}(v + e)$$

2. **Eliminación de nodos hoja**: para eliminar los nodos hoja, tenemos que recorrer todos los nodos del grafo. Sabiendo que tenemos v nodos, entonces la complejidad corresponde a

$$\mathcal{O}(v)$$

3. **Detección de puntos de articulación y división en subgrafos**: para detectar los puntos de articulación de un grafo, tenemos que utilizar el **algoritmo de Tarjan** [10], que es una versión modificada de búsqueda en profundidad (DFS) optimizada para este problema. Dicho algoritmo posee una complejidad lineal respecto a la cantidad de nodos y aristas del grafo, tal como se vio en la sección 2.4.1. Luego, en lo que respecta a la división en subgrafos, sólo debemos eliminar los puntos de articulación del grafo y determinar las componentes conexas restantes, que también requiere de un algoritmo lineal de DFS. Finalmente, este proceso corresponde a

$$\mathcal{O}(v + e)$$

4. **Aplicación de algoritmo de coloreo**: para aplicar el coloreo de cada subgrafo, vamos a hacer uso del **algoritmo de D-Satur** [8] antes expuesto, en el que en cada paso el nodo no coloreado con mayor saturación es seleccionado. Esto es así pues debemos recordar que el problema del coloreo óptimo corresponde a un problema *NP-Hard*, por lo tanto debemos recurrir a una optimización. El algoritmo de D-Satur es cuadrático respecto de la cantidad de nodos, como fue explicado en la sección 2.4.3. En el peor de los casos, el subgrafo coloreado va a suponer la totalidad del grafo, de forma que la complejidad de este paso corresponde a

$$\mathcal{O}(v^2)$$

5. **Búsqueda de cortes minimos y adición de aristas imaginarias**: como bien dijimos antes, en caso de que el coloreo induzca desconexiones, el algoritmo procede a buscar los cortes mínimos de un cierto tamaño, y unir los nodos pertenecientes a dichos cortes con el objetivo de evitar que posean todos el mismo color. Esto es algo que podemos realizar a partir del **algoritmo de Kanevsky** [11], que busca los subconjuntos de nodos de menor tamaño que desconectan el grafo. Dicho algoritmo es cúbico respecto de la cantidad de nodos, como se justificó en la sección 2.4.2. Vamos a suponer que dicho proceso no se va a repetir más de 3 veces para cada subgrafo, de forma tal que la complejidad de este proceso corresponde a

$$\mathcal{O}(v^3)$$

6. **Asignación de colores a nodos hoja**: este último paso es muy sencillo, pues solo requiere que cada nodo hoja tome el mismo color que el de su padre correspondiente, algo que sería lineal respecto del tamaño del grafo en el peor de los casos, correspondiendo su complejidad a

$$\mathcal{O}(v)$$

En conclusión, el proceso que supone la mayor complejidad algorítmica en el desarrollo de la solución es la **búsqueda de cortes mínimos**. Por todo lo expuesto, podemos asumir que la complejidad final, en el peor caso, va a estar dada por $\mathcal{O}(v^3)$.

3.3. Beneficios y limitaciones de la solución propuesta

En esta sección, se detallarán las principales fortalezas y debilidades del algoritmo propuesto en la tesis, con el objetivo de tener un mejor entendimiento del comportamiento del mismo en distintos escenarios según la topología de red correspondiente.

3.3.1. Ventajas

Repasando las ventajas de la solución propuesta, tenemos:

- **Preservación de la conectividad a lo largo del proceso de reinicio**: esta estrategia permite minimizar la desconexión de los nodos lo máximo posible, encontrando un balance entre estabilidad de la red y velocidad de ejecución.
- **Independencia de las capas del DODAG**: a diferencia de otros posibles enfoques que se basan en el número de saltos entre nodos, la solución propuesta no requiere de esta información, lo que permite una mayor adaptabilidad a cambios topológicos, que pueden ocurrir constantemente.
- **Flexibilidad en la planificación**: el sistema por *batches* es muy útil para adaptar el tiempo entre reinicios en función de las necesidades de cada aplicación. En este sentido, la solución esta pensada para que se pueda acotar o alargar el tiempo de espera entre *batches* según los diferentes casos de uso, eligiendo priorizar velocidad o estabilidad.

3.3.2. Desventajas

Si bien la estrategia presentada en esta tesis arroja resultados muy alentadores en múltiples de las métricas elegidas (como vamos a ver en el próximo capítulo), es

importante hacer una mención a aquellas limitaciones que se derivan de la misma, como:

- **Requerimiento de conocimiento global de la topología**: el algoritmo presupone un acceso completo a la conectividad entre los distintos nodos de la red, lo que puede que no ocurra siempre.
- **Dependencia de sincronización**: el funcionamiento por *batches* requiere que los nodos estén completamente sincronizados temporalmente, algo que siempre es problemático en sistemas distribuidos, y más tratándose de dispositivos IoT con recursos más limitados.
- **Desconexiones en cuellos de botella**: sea en la solución propuesta o en cualquiera que refiera a este problema, los cuellos de botella son siempre problemáticos. Dado que por las características del problema no podemos reiniciar todos los nodos de un subgrafo a la vez (pues implicaría apagar la totalidad de los nodos en una cierta zona geográfica), estamos obligados a sufrir ciertas desconexiones en aquellos grafos con puntos de articulación.

Más allá de todos estos inconvenientes, las ventajas son mucho más llamativas, y requieren del análisis del funcionamiento de la solución al ser aplicada en simulaciones.

Capítulo 4

Simulaciones del algoritmo de reinicio por coloreo de grafos

En este capítulo, se expondrán los resultados obtenidos a partir de las simulaciones realizadas en el marco de esta tesis. En particular, se evaluará el algoritmo propuesto anteriormente, en comparación con otras posibles soluciones para llevar a cabo las actualizaciones de firmware correspondientes, tomando como base algunos *KPIs (Key Performance Indicators)*, que son aquellas métricas que consideraremos fundamentales para analizar la eficiencia de nuestra solución propuesta y que serán definidas a continuación.

Para estas simulaciones, optamos por hacer uso del simulador **COOJA**, del sistema operativo **Contiki**. Dicho software fue la herramienta principal en el desarrollo de esta investigación, pues nos proveyó de un entorno sumamente flexible para modificar gran cantidad de configuraciones, desde la cantidad de nodos y los protocolos utilizados, hasta la frecuencia con que los nodos envían paquetes y su radio de transmisión y recepción correspondiente. En este sentido, se evitó realizar simulaciones directamente sobre hardware IoT por varias razones. En primer lugar, el uso de hardware dificulta la posibilidad de reproducir las configuraciones de ciertas simulaciones en particular, dado que hay ciertas variables difíciles de reproducir con exactitud (como el posicionamiento exacto de los dispositivos y la posibilidad de interferencia). Por otro lado, porque trabajar con hardware implicaría interiorizarse con ciertas interfaces que podrían llegar a dilatar innecesariamente las simulaciones, mientras que al usar un simulador como COOJA sólo deberíamos preocuparnos por el diseño de los algoritmos y la implementación utilizada por los nodos. Finalmente, y la razón más importante, es que la solución propuesta se beneficia particularmente de tener una densidad no muy baja de dispositivos. Tenemos que tener en cuenta que el protocolo de Wi-SUN permite un rango de transmisión de, aproximadamente, hasta 1 kilómetro, lo cual aplicado a un contexto realista en el que dichos dispositivos se utilizan, por ejemplo, para medición del consumo eléctrico en hogares, implica la comunicación directa de cada dispositivo con múltiples de otros pares, evitando así situaciones de cuello de botella. En el caso de una simulación, esto sería muy complicado de replicar si utilizáramos directamente hardware, pero con nuestro simulador sólo implica la configuración de un único parámetro, lo cual facilita nuestro

trabajo.

Con todo esto expuesto, se pasará a detallar cada una de las métricas consideradas como KPIs y la razón por la cual esto es así, para después adentrarse en la configuración particular de las simulaciones realizadas, y finalmente poder exponer los resultados junto con las conclusiones que de ellos se derivan.

4.1. Métricas evaluadas

A continuación, y como mencionamos anteriormente, se presentarán aquellas métricas fundamentales utilizadas para el análisis de la performance y comparación del algoritmo propuesto.

4.1.1. Packet Loss Ratio (PLR)

Esta métrica refiere a la proporción de paquetes de datos que, al ser diseminados a lo largo de una red, fallan al tratar de llegar a su destino correspondiente, del total de paquetes que fueron enviados. Es decir:

$$PLR = \frac{Pkt_{perdidos}}{Pkt_{perdidos} + Pkt_{recibidos}}$$

Por ejemplo, si 10 paquetes fueran enviados, y 1 de ellos no lograra llegar a su destino, consideramos que el PLR sería de 0,1. Esta métrica es un parámetro significativo de la calidad de servicio ofrecida, e inversamente proporcional a la performance de la red. A medida que el PLR disminuye, significa que la red en cuestión provee un servicio más confiable. En este sentido, el PLR es fundamental para nuestra investigación, considerando que nuestro objetivo es el de mantener valores de calidad de servicio dentro del rango especificado por ciertos *Service Level Agreements*. Debemos recordar que los dispositivos IoT envían paquetes que contienen información derivada del trabajo de ciertos sensores y que son enviados con el objetivo de salir del DODAG para hacer llegar dichos resultados al organismo correspondiente. Entonces, para computar esta métrica dentro de nuestras simulaciones, vamos a enfocarnos en los paquetes que la raíz del DODAG correspondiente debería recibir, y no en computar los paquetes perdidos salto a salto en transmisiones directas entre dos pares (es decir, nos centraremos en si el paquete fue capaz de atravesar el camino completo o no). En este sentido, consideraremos que un paquete fue recibido en caso de que, luego de ser enviado por el nodo hijo correspondiente, arribe al nodo raíz. Por otro lado, consideraremos que un paquete se perdió en caso de no arribar a dicho nodo. Esta métrica está fundamentalmente relacionada con

los nodos que quedan desconectados en cada *batch*; es decir, con aquellos nodos que durante los períodos de reinicio no poseen ningún vecino disponible. Por esta razón, vamos a incluir también una medición de la cantidad de nodos en esta situación, para fundamentar la razón por la cual el PDR de ciertas simulaciones puede ser más alto o más bajo.

4.1.2. Latencia promedio y jitter

La métrica de latencia refiere a la diferencia entre el tiempo en que un paquete logra arribar a su destino correspondiente y el tiempo en que dicho paquete fue enviado. Es decir:

$$Latencia = t_{recibido}(Pkt) - t_{enviado}(Pkt)$$

Por ejemplo, si un paquete es enviado al segundo 3 de la ejecución, y el mismo es recibido al segundo 5, entonces la latencia es de 2 segundos.

Cuando hablamos de latencia promedio, nos referimos a la sumatoria de la latencia dada para una cantidad n de paquetes distribuidos, dividida entre dicha cantidad n.

Por otro lado, el jitter, también conocido como fluctuación de retardo, es la variación en el tiempo de llegada de los paquetes de datos en una red. Es decir, corresponde a la diferencia entre la latencia máxima de entrega de un paquete y la latencia mínima en la red:

$$Jitter = t_{max}(Pkts) - t_{min}(Pkts)$$

Como ocurre para el PLR, estas métricas nos hablan de la calidad del servicio ofrecido por la red y la congestión de la misma, siendo inversamente proporcional a su performance. En redes que exhiben una latencia promedio más baja, los paquetes enviados son recibidos de forma más rápida. Esto es particularmente importante cuando hablamos de dispositivos IoT, porque existen ciertos métodos que requieren de retardos imperceptibles para evitar errores de precisión. Un ejemplo de esto podría ser el caso de realizar cirugía robótica a distancia, en los que un cirujano hace uso de un sistema de telepresencia para operar a un paciente en otra ciudad haciendo uso de un brazo robótico IoT. Si dicho instrumento tardara, por ejemplo, más de medio segundo en recibir instrucciones por parte del cirujano, el robot actuaría con retardo y podría causar complicaciones en maniobras delicadas como las de cortar tejido.

La latencia de los paquetes puede verse afectada por gran cantidad de factores, como pueden ser las limitaciones dadas por la capacidad de la red, la distancia física entre los dispositivos y el rendimiento de los dispositivos a la hora de propagar paquetes. De todas maneras, si bien es cierto que la solución propuesta implica la inyección dentro de la red de una cantidad de paquetes no menor, y dadas las características del simulador que estamos utilizando, estos parámetros no tienen incidencia en los resultados que vamos a obtener, pues la implementación de la solución en COOJA no incluye el envío de paquetes con el *time-stamp* correspondiente de *reboot*, sino que dicho proceso se ejecuta manualmente sobre cada dispositivo, debido a las limitaciones que ofrece COOJA para la implementación de los nodos. Más allá de todo lo mencionado, el componente agregado en latencia es, en realidad, inevitable para cualquier posible solución a este problema que no lleve a cabo los reinicios de manera manual, pues el proceso de *rebooting* implica la necesidad del envío de paquetes con información, por lo que este inconveniente no influye realmente en los resultados.

Con todo lo expuesto, y retomando las posibles causas para la latencia, vamos a considerar como causa principal la longitud de los caminos hacia la raíz para cada nodo. Es evidente que, a mayor cantidad de nodos deba atravesar un paquete para llegar a su destino, mayor será la latencia con la que dicho paquete cumplirá su recorrido. Para complementar el estudio de la latencia y justificar los resultados, se va a calcular la longitud del camino hacia la raíz para cada nodo que no esté atravesando un proceso de rebooting en cada uno de los distintos *batches*, para poder así diagramar una función de distribución acumulativa. Esto nos permite entender el estado de la topología luego de reiniciar un grupo de dispositivos, y cómo esto afecta a la latencia de los paquetes.

4.1.3. Tiempo de reconexión

Otra de las métricas que vamos a tener en cuenta para medir la eficiencia de nuestro algoritmo va a ser el tiempo de reconexión, que lo vamos a considerar como el momento en que todos los reinicios fueron realizados y el último nodo desconectado logró reconectarse. Evidentemente, esto va a estar muy ligado a la cantidad de *batches* que la solución lleve a cabo, y una solución que reinicie todos los nodos al mismo tiempo va a tener una amplia ventaja con respecto a esta métrica. Por esta razón, no vamos a darle demasiado peso a la misma dentro de nuestro análisis, pero sí la tendremos en cuenta en nuestras comparaciones, porque, en este caso, medir la eficiencia debería ponderar no sólo la cantidad de paquetes perdidos sino también la velocidad con la que todos los nodos fueron capaces de llevar a cabo la actualización correspondiente.

4.1.4. Cantidad de DIOs enviados

Una métrica muy útil para evaluar la performance de nuestra solución es la cantidad de mensajes DIO enviados a lo largo del proceso de reinicio. Como bien fue explicado en la sección 2.2.1, los mensajes DIO son fundamentales en el protocolo RPL porque permiten que un nodo se una a una red y conozca a sus pares. En este sentido, es fundamental evaluar la frecuencia con que dichos mensajes son enviados. Una cantidad muy alta de mensajes DIO puede indicar una congestión muy alta en la red, lo que se traduce en una alta latencia, mientras que una cantidad muy baja de mensajes DIO nos habla de una red poco reactiva a los cambios. En nuestro caso, es importante encontrar un balance: al producir reparaciones locales 'artificiales' de los nodos en nuestra solución, la cantidad de DIOs enviados va a aumentar naturalmente, pero tenemos que cerciorarnos de que esto no se traduzca también en un aumento de la latencia.

4.2. Configuración de las simulaciones

Antes de adentrarnos en los resultados, vamos a repasar los aspectos clave de las simulaciones realizadas con el objetivo de evaluar el impacto de nuestra solución propuesta.

En este apartado, primero se mencionan los diferentes algoritmos examinados, que nos sirven como punto de partida para realizar comparaciones con el algoritmo de coloreo. Es importante recordar que, a día de hoy, no existen papers ni investigaciones sobre este problema en particular, por lo que se van a realizar comparaciones con algunas soluciones 'estándar' que se detallarán a continuación. Luego, se nombran algunas constantes relacionadas a la temporización dentro de la simulación, que fueron determinadas de forma arbitraria en base a la información que se tiene del problema. Finalmente, se indican cuestiones más generales relacionadas con la topología de red.

4.2.1. Algoritmos evaluados

Se compararon cuatro algoritmos diferentes para determinar el orden y la forma en que se agrupan los distintos nodos en la red para ser reiniciados:

- **Reinicio por coloreo de grafos**: como ya fue explicado anteriormente, la solución se basa en técnicas de teoría de grafos, particularmente en la identificación de puntos de articulación, cortes mínimos y una coloración del grafo para definir los lotes de reinicio. Así, se busca minimizar interrupciones y preservar la conectividad hacia la raíz de la red durante todo el proceso.

- **Reinicio por coloreo limitado**: similar a la anterior, pero limitando el número de conexiones (aristas) por nodo a un máximo de 4. Entre esas aristas, se priorizan los nodos de menor rango a la hora de conservar conexiones, con el fin de evitar cambios bruscos en la latencia (al elegir caminos más cortos hacia la raíz). La razón por la que se limita la cantidad de conexiones está relacionada con el funcionamiento del algoritmo de coloreo de grafos. Por ejemplo, si un grupo de 10 nodos estuviera completamente interconectado entre sí, implicaría que los mismos serían asignados 10 colores distintos, y por lo tanto se traduciría en 10 *batches* distintos a la hora de ejecutar los reinicios. Esto puede todavía empeorar más en el caso de que trabajemos sobre una red de alta densidad, lo cual no sería un caso despreciable considerando el alto rango de transmisión que los dispositivos en Wi-SUN poseen. En realidad, un nodo no requiere de tantos caminos alternativos para llegar hacia la raíz, por lo que se optó por limitar a 4 la cantidad de conexiones. Esto sólo afecta a la hora de realizar el coloreo para reducir el tiempo total del proceso, por lo que en la práctica un nodo todavía podría comunicarse directamente con otro de sus pares descartados.

- **Asignación aleatoria de *batches***: en este caso, los nodos se agrupan aleatoriamente en lotes de igual tamaño a los de la alternativa anterior, sin considerar la topología o conectividad de la red. Esta estrategia nos sirve como una referencia para comparar nuestras métricas clave frente a métodos más estructurados como los antes mencionados.

- **Reinicio por mitades**: consiste en seleccionar, de manera aleatoria, una mitad de los nodos para ser reiniciados en un primer *batch*, para luego reiniciar la otra mitad restante. Nuevamente, este enfoque nos es muy útil para comparar con el resto de algoritmos, y en cierta forma es un digno competidor: más allá de su simplismo, el hecho de que los reinicios se lleven a cabo en tan sólo dos *batches* implica que el proceso de actualización estaría terminado en un tiempo menor al resto de métodos. De todas maneras, queda evaluar la fiabilidad de la red, en cuanto a pérdida de paquetes y latencia, que implicaría esta forma.

Estas estrategias fueron elegidas representando diferentes niveles de 'conocimiento' respecto de la topología en cuestión. La realidad es que, a día de hoy, no hay ningún estudio sobre posibles soluciones a este problema, razón por la cual se eligieron estos enfoques para comparar.

4.2.2. Temporización de las simulaciones

Respecto a la gestión del tiempo en las simulaciones, se tomaron las siguientes consideraciones:

- Cada simulación comienza con un período de 2 minutos de operación normal antes del primer reinicio, para permitir la convergencia inicial de la red. No se toman métricas de dicho intervalo.
- Cada *batch* se reinicia de manera simultánea, y toma un tiempo igual al tiempo de reinicio de cada dispositivo. Tratándose de dispositivos IoT, podemos asumir que dicho proceso toma un tiempo del rango de los segundos. Por esta razón, tomamos de manera arbitraria un tiempo de reinicio de 30 segundos.
- Antes de proceder al reinicio del siguiente *batch*, se espera a que:
  - Todos los nodos recién reiniciados elijan un nuevo padre preferido en la topología de red.
  - Se cumpla un temporizador adicional de 20 segundos, con el objetivo de permitir la estabilización de la red.

  En el caso práctico real, para esperar a que todos los nodos reiniciados elijan a un padre, deberíamos hacer uso de un mensaje DAO de RPL, que nos indica que dichos nodos lograron conseguir una ruta hacia la raíz. De todas maneras, un nodo reiniciado no debería tomar en realidad mucho tiempo en reinsertarse en el DODAG, por lo cual esto no es realmente necesario y sólo lo hacemos para evitar el impacto que posibles bugs en el simulador pueden tener en los resultados.
- El tiempo total de simulación corresponde a la suma entre el tiempo de espera inicial de 2 minutos y el tiempo de ejecución del proceso completo de reinicio, multiplicado entre dos:

$$tiempo\_simulacion = (240s + tiempo\_reinicio) * 2$$

  El objetivo de esto es el de tomar mediciones respecto a las métricas de la red pasado un tiempo luego de que el proceso haya finalizado, para poder analizar el impacto que estos métodos tienen en la red.

4.2.3. Topología y parámetros de red

Para simular condiciones variadas y realistas, se generaron diferentes topologías aleatorias con las siguientes características:

- **Cantidad de nodos**: se realizaron simulaciones con 30, 50, 70 y 100 nodos (obviando el nodo raíz).
- **Distribución espacial**: los nodos fueron posicionados de manera aleatoria dentro de una superficie bidimensional de 250 x 250 metros, con un rango de transmisión para cada nodo de 50 metros. De esta manera, en función de la cantidad de nodos incluidos, se hacen pruebas sobre redes más ligeras y otras más densas.
- **Variabilidad de semillas**: para cada tamaño de red, se emplearon diferentes semillas aleatorias para diversificar los escenarios. Esto nos permite evitar sesgos frente a ciertos eventos aleatorios incluidos dentro del simulador, como la interferencia y el envío de paquetes.
- **Tráfico de red**: durante la simulación, los nodos generaron tráfico en la capa de aplicación, enviando paquetes con una periodicidad que varía entre 28 y 32 segundos. Con esta frecuencia, se puede estar seguro de que varios de estos paquetes van a ser enviados durante el proceso de reinicio.

4.3. Resultados obtenidos

En esta sección, se van a mostrar gráficos y tablas con todos los resultados obtenidos para cada una de las métricas antes mencionadas, y se van a obtener conclusiones a partir de ellos.

Es importante notar que la mayoría de los KPIs se muestran a través de la función de distribución acumulativa empírica (*Empirical Cumulative Distribution Function*, **ECDF**), que se trata de una función escalonada que nos permite representar cómo se distribuyen los valores de una métrica a partir de muestras. Dada una serie de mediciones, y para cada valor x, la ECDF muestra qué proporción de las mismas es menor o igual a dicho valor. En este sentido, y para las métricas que estamos evaluando, como la latencia o el *packet loss ratio*, curvas desplazadas hacia la izquierda (es decir, asociadas a valores menores) implican un desempeño mejor del algoritmo en cuestión.

4.3.1. Packet Loss Ratio

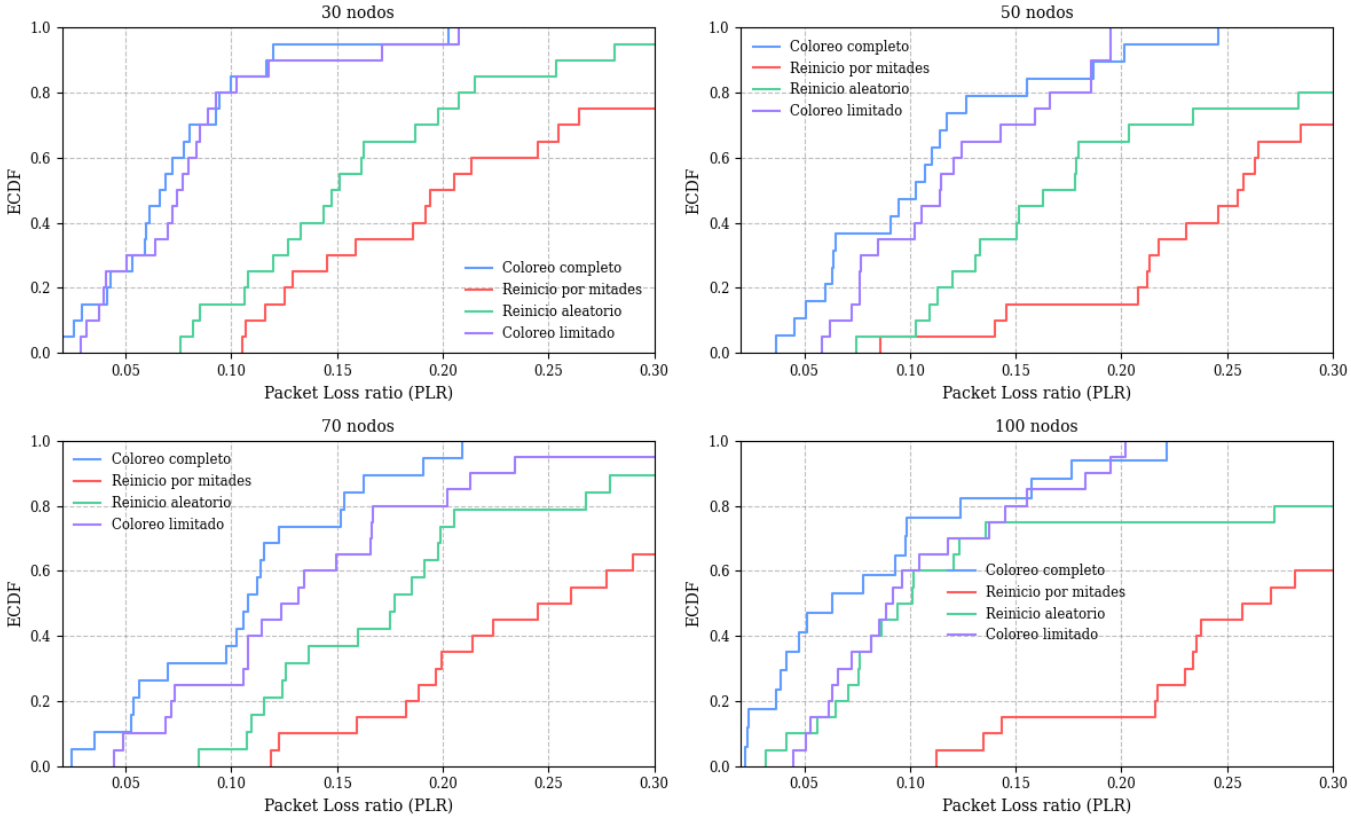

Figura 4.1: ECDF del Packet Loss Ratio para simulaciones de 30, 50, 70 y 100 nodos.

Los resultados obtenidos son claros: las soluciones de coloreo propuestas logran mantener niveles de Packet Loss Ratio notablemente inferiores a los de las soluciones aleatorias, lo cual nos indica que nuestra estrategia, basada en el coloreo de grafos, es sumamente efectiva. Si nos concentramos en las simulaciones de 30 nodos, vemos que la brecha entre la solución de aristas limitadas y la solución aleatoria es considerablemente grande, lo que nos dice que nuestra solución es particularmente buena en densidades normales. Echando un vistazo a las simulaciones de 50 y 70 nodos, vamos a notar que la brecha no es tan amplia como para las simulaciones de 30 nodos, pero sigue siendo de un tamaño notable y que nos indica que esta solución es muy útil para densidades altas. Pasando a las simulaciones de 100 nodos, en las que la densidad llega a niveles un poco exagerados, vamos a notar que la brecha se termina de acortar, por fuera de algunos 'outliers' para la solución random. Esto no es de extrañar, porque densidades muy altas nos hablan de muchos posibles caminos hacia la raíz, y en tales casos, el orden en que reiniciemos los nodos no es determinante en la pérdida de paquetes, pues las probabilidades de tener siempre un nodo vecino disponible son considerablemente más altas. Es interesante notar cómo la solución que hace uso de todas las aristas tiende a performar considerablemente

mejor que el resto de soluciones en todos los casos.

Para justificar estos resultados, se procede a mostrar una ECDF de la cantidad de nodos desconectados por *batch* en cada una de las simulaciones.

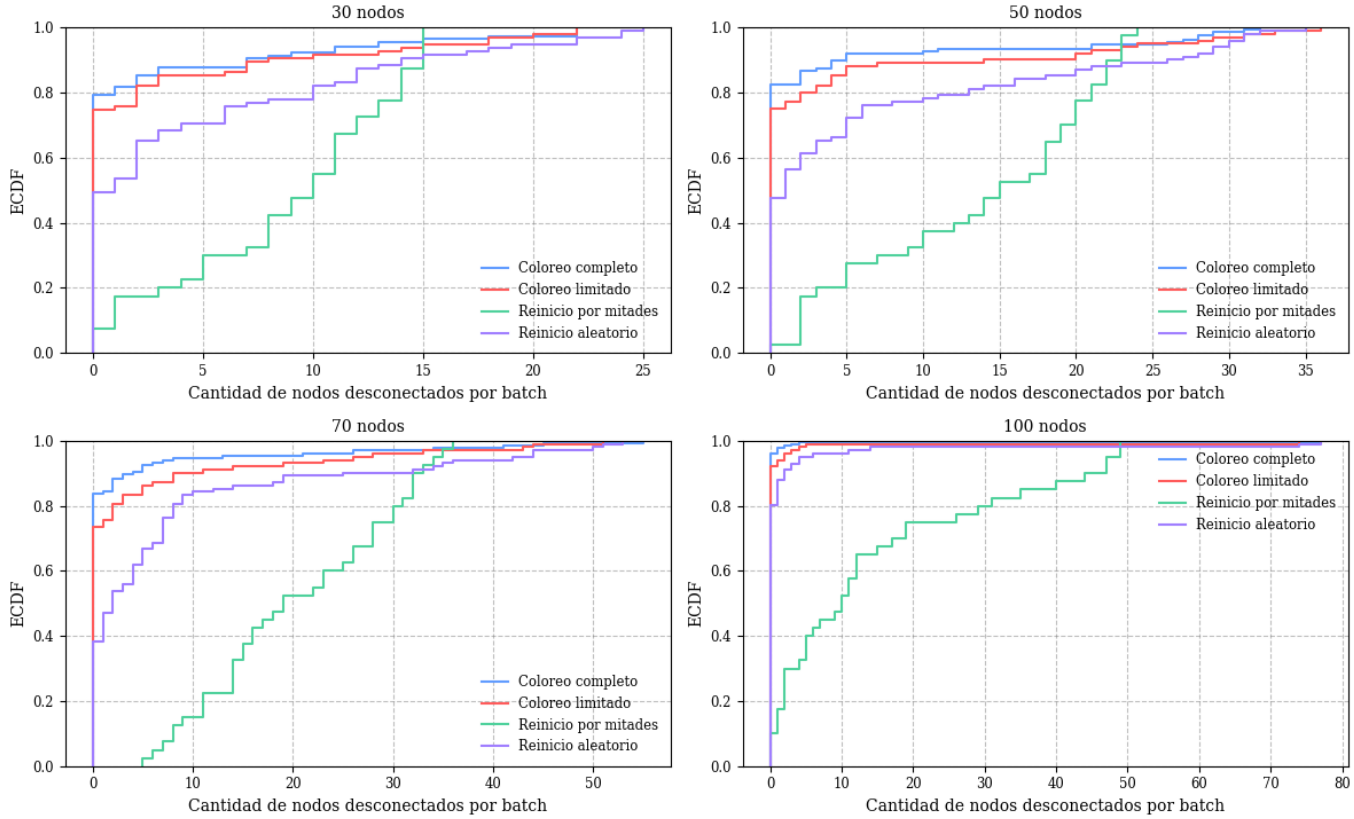

Figura 4.2: ECDF de la cantidad de nodos desconectados por *batch* para simulaciones de 30, 50, 70 y 100 nodos.

Como podemos ver en los gráficos, las brechas entre nuestras soluciones y las soluciones randomizadas se condicen con los resultados obtenidos para el PLR: las mismas se acortan cada vez más a medida que aumenta la densidad de la simulación. Como fue mencionado anteriormente, esto no es algo de extrañar. En definitiva, nuestra solución es efectiva en todo tipo de topologías, y ve aún mejores resultados en aquellas de densidades que tienden a bajas.

4.3.2. Latencia y Jitter

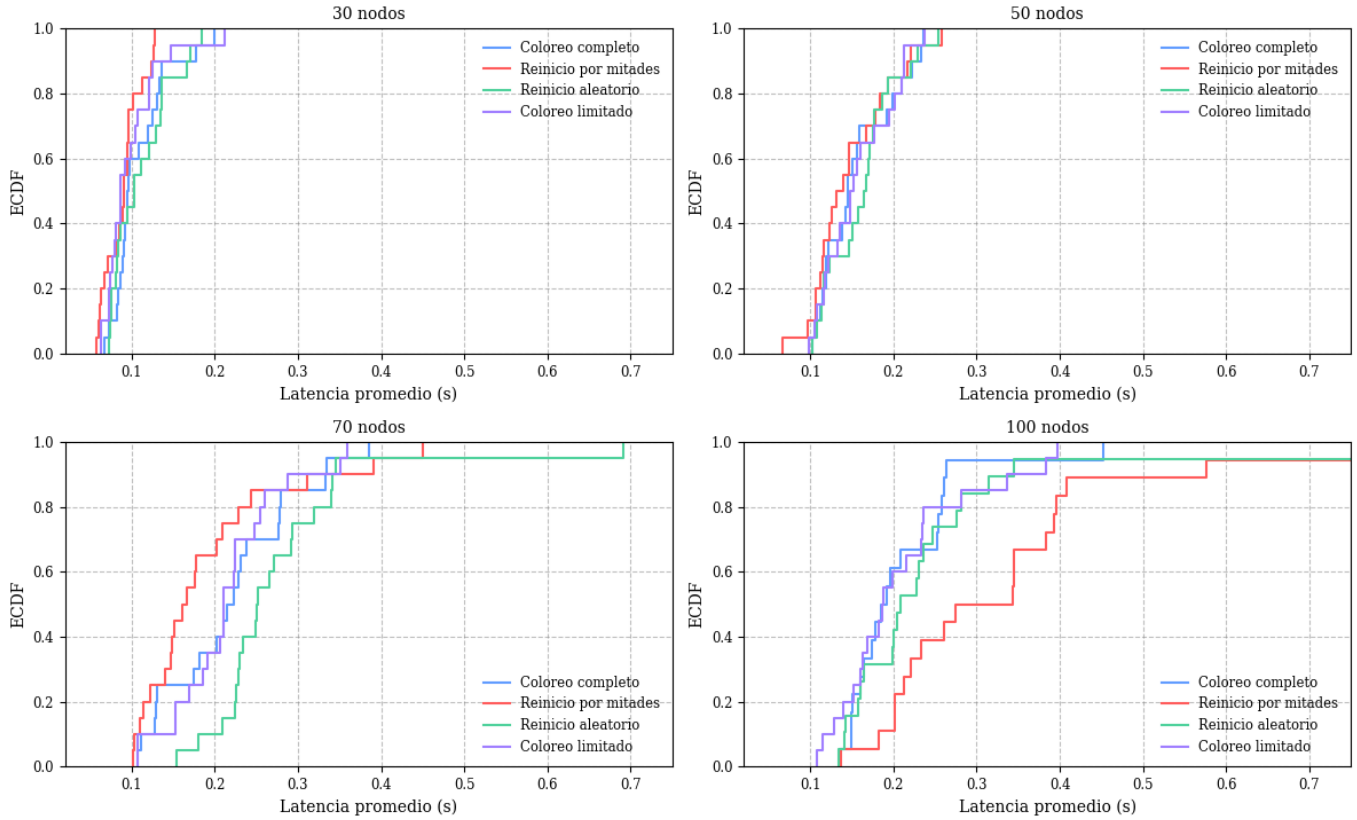

Figura 4.3: ECDF de la latencia para simulaciones de 30, 50, 70 y 100 nodos.

En lo que respecta a la latencia obtenida, notamos algo interesante. Para simulaciones de 30 y 50 nodos, la brecha entre soluciones es mínima, lo cual nos dice que nuestras soluciones logran ser altamente efectivas en lo que respecta a pérdida de paquetes sin sacrificar la performance de la red. Además, los niveles de latencia se mantienen en rangos considerablemente pequeños, lo cual probablemente tenga que ver con la densidad de la red. En lo que respecta a las simulaciones de 70 nodos, vemos que la brecha se agranda un poco más, y la solución por mitades tiende a valores ligeramente inferiores. Finalmente, llegando a las simulaciones de 100 nodos, vemos que la brecha se inverte llamativamente, de forma que la solución por mitades termina incurriendo en valores de latencia promedio considerablemente mayores, indicándonos así su mala performance respecto del resto de soluciones. Esto posiblemente esté dado por el hecho de reiniciar la mitad de la topología al mismo tiempo: esto incurre en una gran cantidad de mensajes de señalización de RPL que tienden a desequilibrar la topología, dando lugar así a estos resultados. En definitiva, en lo que respecta a la latencia promedio, vemos que para la mayoría de densidades la diferencia no es sustancial, pero, llegando a densidades particularmente altas, realizar un proceso de reinicio en una mayor cantidad de *batches* nos evita latencias altas.

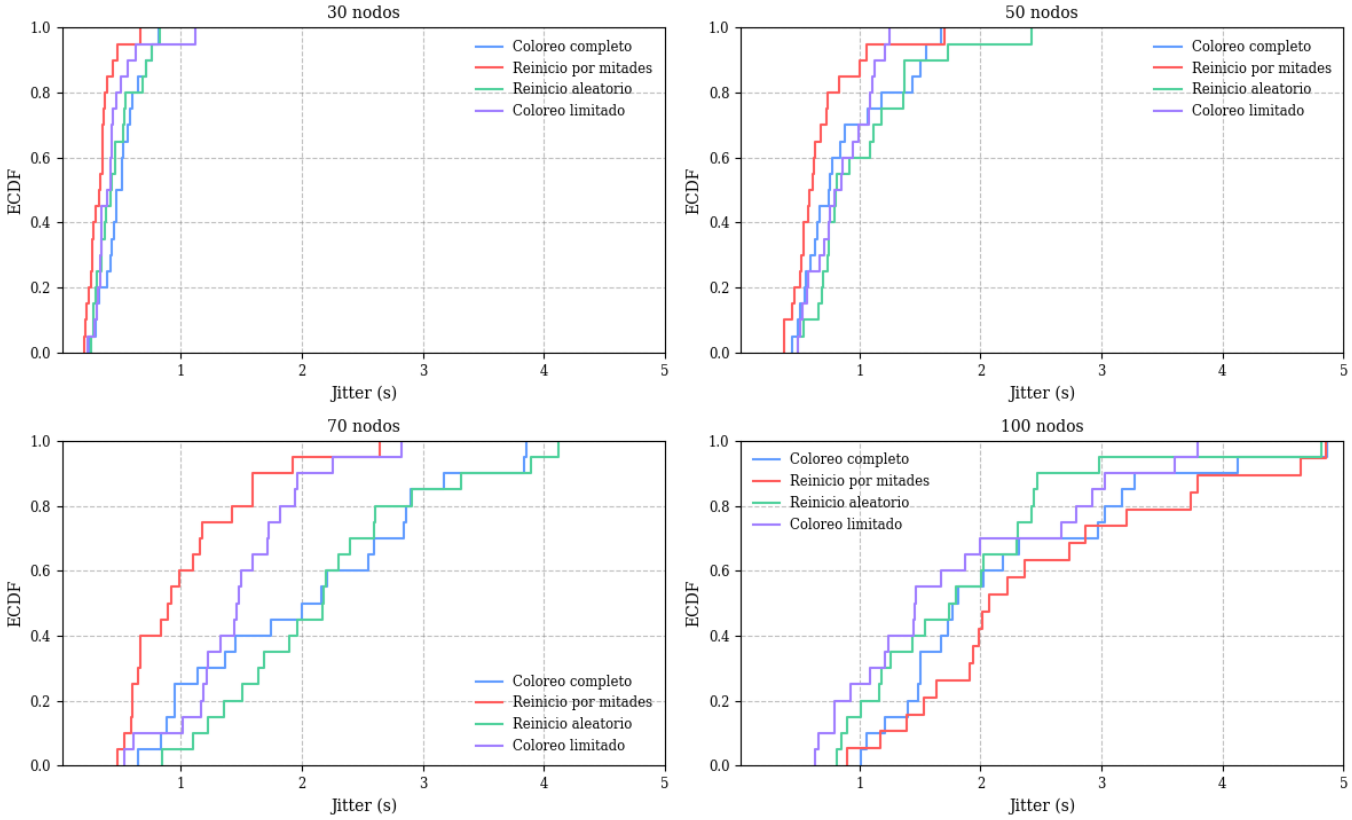

Figura 4.4: ECDF del jitter para simulaciones de 30, 50, 70 y 100 nodos.

Si nos fijamos en los resultados del jitter, la tendencia es similar a los resultados obtenidos para la latencia promedio: para simulaciones de 30 y 50 nodos la brecha es pequeña, para simulaciones de 70 nodos la brecha está ligeramente a favor de la solución por mitades, y para simulaciones de 100 nodos la solución por mitades tiene una performance considerablemente peor. Para el resto de soluciones, nuevamente, los resultados son bastante parecidos, lo cual nos indica que esta métrica está muy relacionada con la cantidad de *batches* en los que el proceso de reinicio se lleva a cabo.

Para justificar estos resultados obtenidos, se calculó el tamaño de camino promedio por *batch* para cada una de las simulaciones.

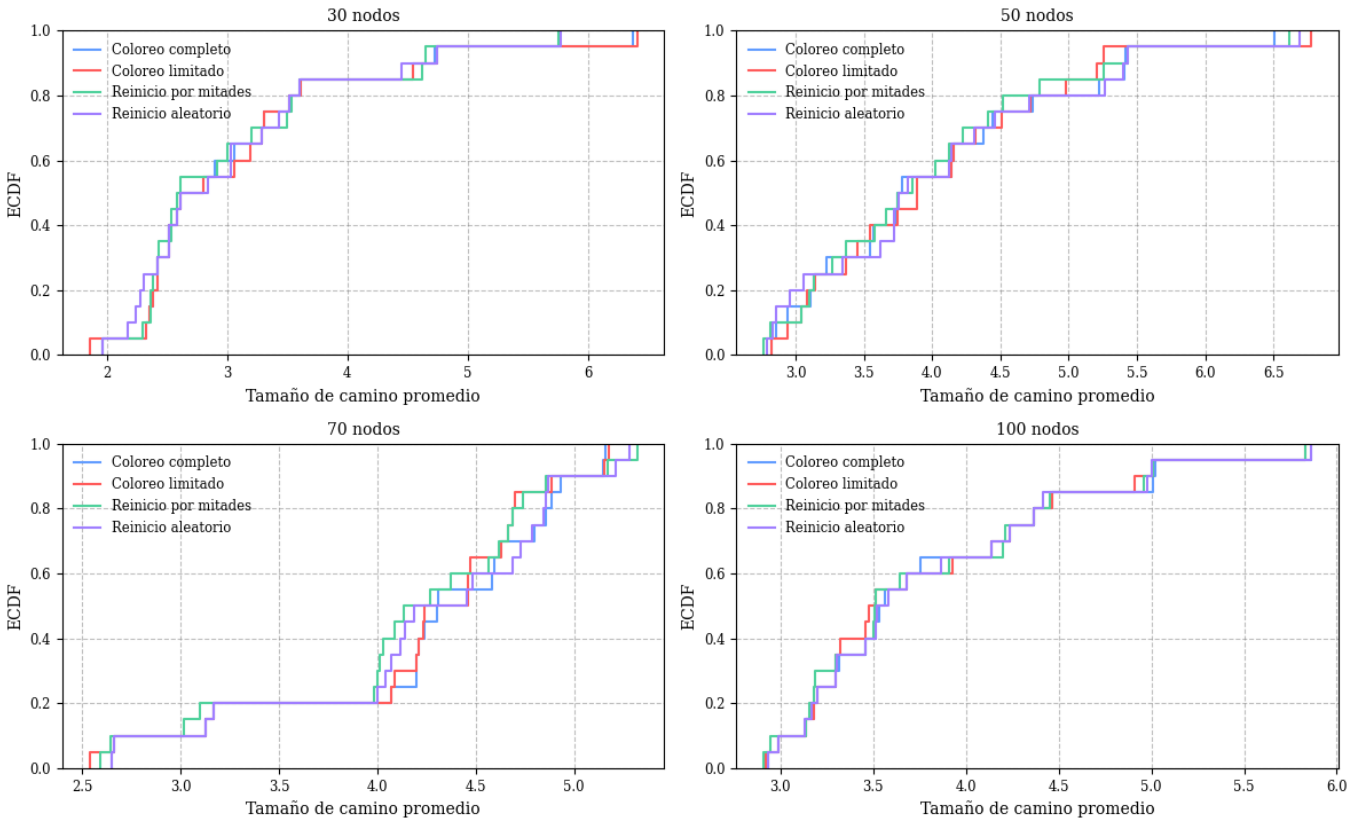

Figura 4.5: Tamaño de camino promedio por *batch* para simulaciones de 30, 50, 70 y 100 nodos.

Estos resultados son la explicación perfecta de por qué nuestras soluciones y la randomizada tienden a tener valores de latencia tan similares. Independientemente de la estrategia que llevemos a cabo, el tamaño de camino promedio para cada uno de los *batches* realizados se mantiene en valores muy similares. Esto se traduce en una menor cantidad de saltos a realizar por cada uno de los paquetes para llegar a su destino, la raíz, lo cual implica una menor latencia. En este sentido, y con esto expuesto, no es de extrañar que los valores de latencia se mantengan similares, y nos habla de un rendimiento de red estable aún a lo largo de procesos de reinicio.

4.3.3. DIO enviados por *batch*

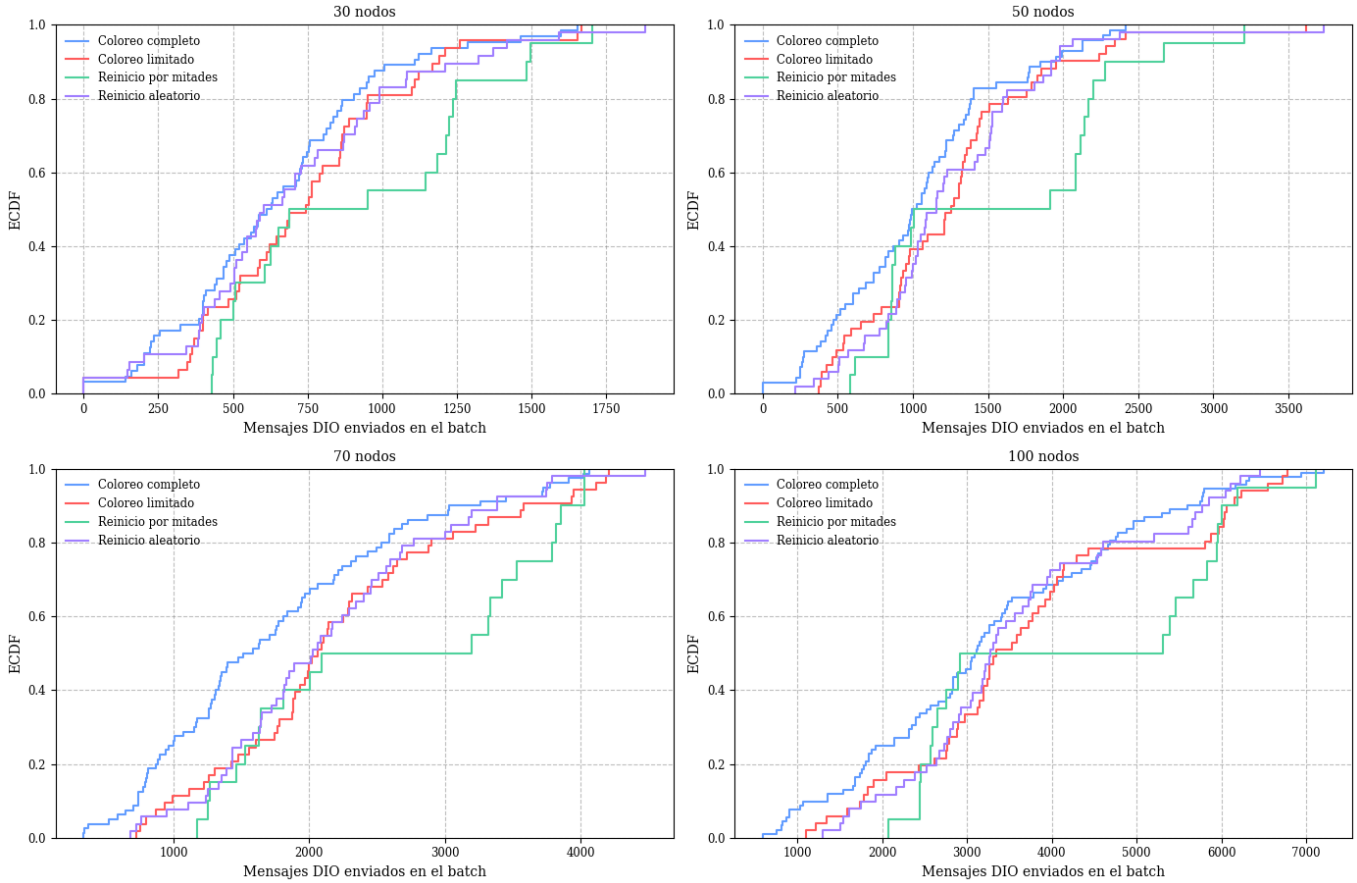

Figura 4.6: Cantidad de mensajes DIO enviados en cada *batch* en simulaciones de 30, 50, 70 y 100 nodos.

En lo que respecta a la cantidad de mensajes DIO enviados en cada *batch*, volvemos a tener resultados muy interesantes. Para simulaciones de todas las densidades, la solución que hace uso de todas las aristas es la que tiende a enviar una menor cantidad de mensajes, aunque es paradójicamente la que mayor cantidad de nodos tiene en funcionamiento al mismo tiempo (pues, a mayor cantidad de *batches*, los mismos son de menor tamaño). Esto nos dice que dicha solución no genera tanta inestabilidad en la topología, razón por la cual es muy efectiva. Por otro lado, la solución de reinicio por mitades tiene una performance notablemente peor en este sentido, lo cual no es de extrañar, pues muchos nodos son reiniciados a la vez lo que implica un envío de una alta densidad de mensajes RPL que pueden terminar saturando la red. En lo que respecta a las otras dos soluciones, sus resultados son bastante similares, lo cual probablemente tenga que ver con la cantidad de *batches* en que se ejecutan sus procesos de actualización. En definitiva, ejecutar las actualizaciones en *batches* más pequeños es beneficioso para evitar un envío desmedido de paquetes RPL y, así también, una reducción en la performance de la red. Por todo esto, la solución no limitada ofrece mejores resultados para esta métrica.

4.3.4. Tiempo de reconexión

Métrica	Limitada	Coloreo	Aleatoria	Mitades
Finalización del reboot (s)	275.21	368.79	276.70	**98.73**
75 % reconectados (s)	283.78	379.72	277.90	**101.90**
90 % reconectados (s)	284.50	379.84	280.31	**104.61**
100 % reconectados (s)	287.76	383.64	286.87	**110.92**

Cuadro 4.1: Resultados promedios para las simulaciones de 30 nodos.

Métrica	Limitada	Coloreo	Aleatoria	Mitades
Finalización del reboot (s)	335.37	424.98	311.36	**101.91**
75 % reconectados (s)	338.73	434.51	326.80	**106.27**
90 % reconectados (s)	342.13	435.01	344.89	**111.15**
100 % reconectados (s)	344.56	435.11	346.29	**115.24**

Cuadro 4.2: Resultados promedios para las simulaciones de 50 nodos.

Métrica	Limitado	Coloreo	Aleatoria	Mitades
Finalización del reboot (s)	314.16	512.36	312.04	**98.53**
75 % reconectados (s)	317.51	514.57	319.92	**100.82**
90 % reconectados (s)	318.90	515.86	322.85	**102.80**
100 % reconectados (s)	334.86	516.78	334.33	**111.15**

Cuadro 4.3: Resultados promedios para las simulaciones de 70 nodos.

Métrica	Limitado	Coloreo	Aleatoria	Mitades
Finalización del reboot (s)	311.33	564.90	316.02	**97.81**
75 % reconectados (s)	312.97	565.57	318.35	**99.58**
90 % reconectados (s)	313.11	569.01	319.05	**100.91**
100 % reconectados (s)	325.37	579.68	327.89	**114.88**

Cuadro 4.4: Resultados promedios para las simulaciones de 100 nodos.

En lo que respecta al tiempo de reconexión, obtenemos resultados esperables: la solución por mitades termina siempre antes que el resto de las soluciones. Esto no es algo que nos llame la atención, pues tenemos que tener en cuenta que, mientras menor sea la cantidad de *batches* en la que llevamos a cabo el proceso de actualización, mejores resultados temporales vamos a obtener. Si bien esto nos indica una rapidez de la solución por mitades para ocuparse de las actualizaciones, la realidad es que

esto no es suficiente si estamos poniendo en peligro la estabilidad de nuestra red e induciendo la pérdida de una gran cantidad de paquetes, lo que puede llegar a ir en contra de los *Service Level Agreements* en cuestión. Por otro lado, la solución de coloreo total toma una cantidad considerablemente mayor de tiempo a la solución limitada y la aleatoria, a la vez que su efectividad en lo que refiere a la pérdida de paquetes es muy superior. En este sentido, ponderar entre la solución de coloreo total y la de coloreo limitado va a depender de las necesidades que debamos atender a la hora de actualizar: si la urgencia de actualizar es dominante, entonces vamos a optar por la solución limitada; si no hay razón para apresurar el proceso en pos de evitar la pérdida de paquetes, optaremos por la solución de coloreo completo.

Capítulo 5
Conclusiones

A lo largo de esta tesis, se desarrolló y evaluó una estrategia basada en coloreo de grafos para organizar el proceso de actualización de firmware en redes inalámbricas malladas, con el objetivo de minimizar la pérdida de paquetes y preservar la conectividad de los nodos durante los reinicios secuenciales.

5.1. Ponderación de los resultados obtenidos

Los resultados obtenidos muestran que tanto la solución propuesta (basada en grafos completos con análisis de puntos de articulación y cortes mínimos) como la versión limitada (con un número acotado de aristas por nodo) presentan desempeños significativamente superiores frente a estrategias aleatorias o simplificadas como el reinicio por mitades.

- En términos de Packet Loss Ratio, ambas estrategias estructuradas logran mantener pérdidas muy bajas, especialmente en topologías de densidad media, donde el orden del reinicio y la planificación de *batches* tienen un mayor impacto. La solución propuesta presenta los mejores resultados en este sentido, al costo de una mayor complejidad y mayor duración total del proceso de actualización.
- Respecto al tiempo de reconexión, la solución limitada logra completar el proceso en menor tiempo, mostrando así un balance entre velocidad y eficiencia que puede ser deseable en escenarios donde el tiempo de actualización es crítico.
- La latencia y el jitter se mantuvieron controlados en ambas estrategias, lo cual implica que las mismas logran llevar a cabo las actualizaciones con una baja pérdida de paquetes y sin penalizar la estabilidad de la red. Además, la cantidad de mensajes de señalización generados durante el proceso fue menor en la solución propuesta, lo que demuestra su capacidad de reorganizar la topología de forma más estable.

En resumen, la solución propuesta se destaca por su robustez y confiabilidad, siendo

ideal en contextos donde se prioriza la continuidad operativa y la minimización de fallas. Por otro lado, la solución limitada representa un buen compromiso entre performance y velocidad de ejecución, lo cual puede ser útil en contextos donde el tiempo total del proceso debe acotarse.

5.2. Trabajos futuros

Este trabajo abre la puerta a múltiples posibles mejoras a realizar en el área:

- **Implementación en hardware real**, para evaluar como se comporta el algoritmo teniendo en consideración otra enorme cantidad de variables que podrían llegar a manifestarse en condiciones realistas, como interferencia y consumo energético.
- **Adaptación dinámica de los reinicios**, en la que los *batches* se puedan ajustar en tiempo de ejecución a determinadas condiciones de red que puedan estar dándose. Por ejemplo, podría ocurrir que la ejecución de uno de los *batches* produzca un aumento brusco de la latencia, en base a lo cual podríamos posponer la ejecución del *batch* consecutivo con tal de no generar inestabilidad en la red.
- **Uso de buffering para nodos desconectados**, que nos pueden ser sumamente útiles para aquellos nodos que representan 'cuellos de botella', y que si son desconectados implican sí o sí la desconexión de un subgrafo (independientemente del orden en que llevemos a cabo los reinicios). En estos casos, podríamos un buffering acorde a la frecuencia en que los datos de la capa de aplicación son enviados, de forma que no se pierda ningún paquete. Debemos tener en cuenta que estamos trabajando con dispositivos IoT de bajo costo y que por lo tanto tienen capacidades de memoria muy reducidas, por lo que dicho agregado debe ser mínimo en la medida de lo posible.
- **Incorporación de prioridades por tipo de nodo**, de forma que ciertos dispositivos (como ciertos sensores críticos) puedan ser reiniciados en momentos específicos, según su rol dentro de la red. Esta priorización podría surgir automáticamente a partir de métricas de centralidad topológica o tráfico cursado históricamente.
- **Optimización del algoritmo de coloreo**, reemplazando o combinando el enfoque actual con heurísticas más eficientes o incluso técnicas de inteligencia artificial, como algoritmos genéticos. Esto permitiría reducir aún más la cantidad de *batches* sin comprometer la conectividad de la red, lo cual acortaría el tiempo total del proceso de actualización. Debemos tener en cuenta que

estamos utilizando un algoritmo de coloreo por aproximación, y por lo tanto la solución a dicho problema puede no ser óptima. Además, los algoritmos de detección de cortes mínimos implican una gran complejidad algorítmica que puede significar mucho tiempo de procesamiento en topologías muy grandes.

Referencias

[1] J. F. Kurose and K. W. Ross (2021). Çomputer Networking: A Top-Down Approach"8th ed., Pearson.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci (2002). "Wireless Sensor Networks: A Survey", Computer Networks, vol. 38, no. 4, pp. 393–422, Elsevier.

[3] F. Liu and Y. Bai (2012). .ᴬn Overview of Topology Control Mechanisms in Multi-Radio Multi-Channel Wireless Mesh Networks", EURASIP Journal on Wireless Communications and Networking, vol. 2012, no. 1, Article 324.

[4] L. Atzori, A. Iera, and G. Morabito (2010). "The Internet of Things: A survey", Computer Networks, vol. 54, no. 15, pp. 2787–2805, Elsevier.

[5] S. Adeoye (2025). Ïnternet of Things (IoT): A Vision, Architectural Elements and Future Directions,Çognizance Journal of Multidisciplinary Studies, vol. 5, no. 1, pp.

[6] B. Heile, B. Liu, M. Zhang and C. Perkins (2017). "Wi-SUN FAN Overview,Ïnternet-Draft draft-heile-lpwan-wisun-overview-00, Wi-SUN Alliance / Huawei / Futurewei.

[7] G. Montenegro, N. Kushalnagar, J. Hui and D. Culler (2007). "Transmission of IPv6 Packets over IEEE 802.15.4 Networks (6LoWPAN),RFC 4944, Internet Engineering Task Force (IETF).

[8] Brélaz, D. (1979). New methods to color the vertices of a graph. *Communications of the ACM*, 22(4), 251–256.

[9] Jensen, T. R., & Toft, B. (1995). *Graph Coloring Problems*. Wiley-Interscience.

[10] Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 146–160.

[11] Kanevsky, D. M. (1993). Finding all minimum-size separating vertex sets in a graph. *Networks*, 23(6), 533–541.

[12] Provan, J. S & Shier, D. R. (1991). A paradigm for listing (s-t)-cuts in graphs. *Technical Report UNC/OR TR91-3*, Department of Operation Research, University of North Carolina at Chapel Hill.

[13] Kanevsky, D. M. (1988). Compact representacion of the separating k-sets of a graph. *Techincal Report ACT-88*, Coordinated Science Laboratory, University of Illinois.

[14] Winter, T., Thubert, P., et al. (2012). RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. *RFC 6550*, IETF.

[15] Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[16] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[17] Brown, J. R. (1972). Chromatic scheduling and the chromatic number problem. *Management Science*, 19(4-Part-1), 456–463.

[18] Shirey, R. W. (1969). Implementation and Analysis of Efficient Graph Planarity Testing Algorithms. *Ph.D. Thesis*, University of Wisconsin Computer Sciences Department.

[19] S. Even and R. E. Targan, Network flow and testing graph connectivity. SIAM J. Comput. 4 (1975) 507- 518.

[20] Z. Galil, Finding the vertex connectivity of graphs. SIAM J . Cornput. 9 (1980) 197-199.

[21] J. C. Picard and M. Queyranne, On the structure of all minimum cuts in a network and applications. Math. Program. Study 13 (1980) 8-16.

[22] C. Gungogan, D. Barthel, E. Baccelli, DIS Modifications Draft. Internet Engineering Task Force (2019).

[23] Sourailidis, D., Koutsiamanis, R.-A., Papadopoulos, G. Z., Barthel, D., and Montavont, N. (2020). RFC 6550: On minimizing the control plane traffic of RPL-based industrial networks. In 2020 IEEE 21st International Symposium on .ᴬ World of Wireless, Mobile and Multimedia Networks"(WoWMoM) (pp. 157–163). IEEE.

[24] J. Guo, C. Han, P. Orlik and J. Zhang (2012). "Loop-Free Routing in Low-Power and Lossy Networks, SENSORCOMM,İnternational Conference on Sensor Technologies and Applications.

[25] A. L. Kampen, K. Ovsthus and Ø. Kure (2014). Reconnection strategies in WSN running RPL,"39th Annual IEEE Conference on Local Computer Networks Workshops, Edmonton, AB, Canada, pp. 602-609.

[26] A. Dunkels, B. Grönvall and T. Voigt (2004). Çontiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors,"Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN). IEEE.

[27] F. Österlind, A. Dunkels, J. Eriksson, N. Finne and T. Voigt (2006). Çross-Level Sensor Network Simulation with COOJA,"Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN). IEEE.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein (2009). Ïntroduction to Algorithms,"3rd Edition. MIT Press.