INSTITUCIÓN UNIVERSITARIA POLITÉCNICO GRANCOLOMBIANO
INGENIERÍA DE TELECOMUNICACIONES
TRABAJO DE GRADO

IMPLEMENTACIÓN DE ALGORITMO AODV EN LENGUAJE DE PROGRAMACIÓN GO PARA REDES SIN INFRAESTRUCTURA APLICADO A IoT

AUTOR:
SAÚL ELÍAS ORTEGA MENESES
CÓDIGO: 1530650366

ASESORES:
MSc. WILMAR JAIMES FERNÁNDEZ
MSc. RICARDO GÓMEZ VARGAS

JUNIO DE 2017

RESUMEN

En el presente documento se analiza y estudia el protocolo de enrutamiento para redes sin infraestructura AODV, y posteriormente se desarrolla una implementación de éste en lenguaje de programación Go. Se usa el lenguaje de programación Go debido a que no existe una implementación de AODV en él, y es un lenguaje relativamente reciente que apunta a ser muy versátil debido a su sencillez y sintaxis moderna. Además, existen implementaciones de Go para programar en diferentes dispositivos de hardware orientados a la internet de las cosas (IoT), el cual será el objetivo final de esta implementación, pudiendo así implementarse este protocolo en dispositivos conectados en redes sin infraestructura con gran independencia del hardware utilizado.

ABSTRACT

This paper analyzes and studies the routing protocol for mobile ad hoc networks AODV, and later develops an implementation of this in Go programming language. The Go programming language is used because there is no AODV implementation in it, and it is a relatively recent language that aims to be very versatile because of its simplicity and modern syntax. In addition, exists implementations of Go to program in different hardware devices oriented to the internet of things (IoT), which will be the final objective of this implementation, being able to implement this protocol in devices connected in mobile ad hoc networks with great independence of the hardware used.

PALABRAS CLAVE

MANET, IoT, AODV, Go, Redes sin infraestructura, Internet de las cosas, Protocolo de enrutamiento, RFC 3561.

KEYWORDS

MANET, IoT, AODV, Go, Mobile ad hoc networks, Internet of Things, Routing protocol, RFC 3561.

TABLA DE CONTENIDO

1. INTRODUCCIÓN

El algoritmo AODV (Ad hoc On-Demand Distance Vector) establece un enrutamiento dinámico y de autodescubrimiento para la comunicación entre nodos en una red sin infraestructura. Estos nodos pueden ser móviles y pueden obtener rutas de nuevos destinos de manera rápida, gracias a su capacidad para adaptarse a enlaces dinámicos. AODV se caracteriza por un bajo consumo de memoria y procesamiento, y poca utilización de red. AODV además permite que los nodos respondan oportunamente a cambios en la topología de la red, e impide la formación de bucles infinitos (asociados a protocolos de vector distancia).

El presente documento describe la implementación del algoritmo AODV en código para el lenguaje de programación Go de manera secuencial acorde a cada etapa de AODV. Este documento sigue más o menos la misma estructura y parte del contenido establecidos en la especificación RFC 3561 de IETF, desde donde se extrae información para explicar el funcionamiento del algoritmo, y se agrega detalles técnicos y código sobre su implementación en el lenguaje de programación Go.

JUSTIFICACIÓN

Se prevé que las aplicaciones de internet de las cosas crezcan exponencialmente en los próximos años[1], y esta es un área que en muchos casos puede requerir de implementación de redes sin infraestructura para la comunicación de dispositivos. Actualmente existen entornos de desarrollo en diferentes lenguajes de programación para muchas plataformas de hardware, sin embargo las implementaciones de algoritmos para comunicación en redes sin infraestructura son escasas en cualquier lenguaje, según las infructuosas búsquedas realizadas[2]. Por este motivo se plantea la implementación de un algoritmo de comunicación en redes MANET para el lenguaje Go, debido a que su popularidad crece continuamente[3], y además tiene entornos de desarrollo para diferentes plataformas de hardware, incluyendo Arduino, CHIP, y otras, sin mencionar plataformas de microcomputadores como Raspberry Pi o Intel Edison.

La implementación de un algoritmo de comunicación de redes MANET se enmarca dentro del área de las redes sin infraestructura, pero también es un componente de gran importancia en proyectos de internet de las cosas, dado que permite el despliegue de pequeños dispositivos conectados en enjambre formando una red descentralizada.

Los algoritmos para redes MANET se dividen principalmente en 3 tipos: Reactivos, proactivos, e híbridos. Cada uno tiene sus ventajas y desventajas, de tal manera que su elección depende de la aplicación específica para cada red MANET. Se desarrollará la implementación específicamente del algoritmo AODV (Ad-hoc Ondemand Distance Vector) con base en la especificación 3561 de IETF. Este es un algoritmo reactivo, lo que significa

[1] http://www.datagora.es/2016/06/03/el-crecimiento-de-iot-sobrepasa-todas-las-expectativas/

[2] https://www.google.com.co/#q=%2Bcode+%2Bimplementation+of+%2Balgorithms+%2Bmanet

[3] http://www.microsiervos.com/archivo/ordenadores/lenguaje-programacion-ano-google-go-segun-tiobe.html

que el algoritmo se encarga de encontrar una ruta bajo demanda, según una solicitud de ruta. El algoritmo AODV, como los demás algoritmos reactivos, transmiten menor cantidad de datos que los proactivos, y, a diferencia de los proactivos, tiene una rápida reacción sobre la reestructuración de la red o fallas de algún nodo. Además, se elige el algoritmo AODV por sobre otros algoritmos reactivos debido a que este cuenta con el apoyo de la IETF[4].

Este proyecto es de suma relevancia tanto en el área de las redes sin infraestructura como en el área de internet de las cosas, debido a que las implementaciones del algoritmo AODV son escasas, e incluso inexistentes para entornos compatibles con plataformas de hardware. Además, no existe una implementación de este algoritmo en el lenguaje Go[5], el cual tiene un futuro prometedor en el desarrollo de hardware debido a las nuevas plataformas de microcomputadores con puertos GPIO para conexión de sensores y dispositivos de hardware.

OBJETIVO GENERAL

Desarrollar una implementación del algoritmo AODV en el lenguaje de programación Go para implementación en dispositivos orientados a IoT en redes sin infraestructura.

OBJETIVOS ESPECÍFICOS

Estudiar y analizar el funcionamiento del algoritmo AODV.

Desarrollar rutinas de programación de programación acorde a cada una de las etapas y funciones del algoritmo AODV.

Ejecutar pruebas en un entorno con varios nodos ejecutando AODV.

Generar documentación explicando las rutinas de programación de la implementación.

[4] https://tools.ietf.org/html/rfc3561

[5] https://www.google.com.co/#q=%2Bimplementation+of+%2Balgorithm+%2Baodv+%2Bmanet+for+%2Bgo+golang

2. MARCO TEÓRICO

2.1. AODV

AODV define tres tipos de mensajes: Solicitudes de ruta (RREQ), respuestas de ruta (RREP), y errores de ruta (RERR). Existe también un cuarto mensaje de acuse de recibo (RREPACK). Estos mensajes se envían y reciben mediante UDP en una red TCP/IP. Los mensajes RREQ son enviados por el nodo solicitante a la dirección de broadcast 255.255.255.255. Cuando un nodo necesita conocer una ruta hacia un destino, envía una solicitud broadcast. Esta ruta puede ser encontrada ya sea porque el nodo encontró directamente el destino, o porque otro nodo intermediario la conoce. Una ruta es válida si su destino tiene un número de secuencia asociado igual o superior al que conoce el nodo solicitante y que fue enviado en el encabezado del mensaje RREQ. Si el nodo que recibió el mensaje RREQ tiene una ruta válida, responde al nodo solicitante un mensaje RREP con la información de la ruta.

Los nodos monitorean el estado del enlace hacia las rutas activas. Cuando un nodo detecta que un enlace hacia una ruta activa se cae, notifica la pérdida del enlace a otros nodos mediante un mensaje RERR. El mensaje RERR indica que ese destino ya no se puede alcanzar por la ruta establecida. Este mecanismo de monitoreo y reporte funciona mediante una “lista de precursores” que contiene la dirección IP de cada nodo que constituye la ruta. Estas listas de precursores son usadas cuando un nodo debe responder una solicitud de ruta.

Toda esta información obtenida de nodos vecinos es organizada y con ella cada nodo crea una tabla de enrutamiento con todas las rutas a todos los destinos conocidos. Esta tabla de enrutamiento contiene los siguientes campos:

- Dirección IP de destino
- Número de secuencia de destino
- Marca de validez del número de secuencia de destino
- Otras marcas de estado y enrutamiento (validez, reparabilidad, etc.)
- Interfaz de red por donde se llega al destino
- Cantidad de saltos requeridos para alcanzar el destino
- Dirección IP del salto más próximo en la ruta hacia el destino
- Lista de precursores
- Tiempo de vida de la ruta

La administración del número de secuencia es muy importante para evitar bucles infinitos. Cuando un enlace se cae, la ruta será marcada como no válida en la tabla de enrutamiento, y su número de secuencia cambiará.

Las comunicaciones UDP se realizan utilizando el puerto 654.

2.2. GO

Go es un lenguaje de programación compilado conciso, limpio, y eficiente. Al ser un lenguaje bastante moderno (lanzado en 2009), combina características aplaudidas en otros lenguajes, en particular combina una sintaxis parecida a C con otras características y facilidades de Python u otros lenguajes dinámicos.

Go es un lenguaje concurrente, lo cual permite ejecutar procesos paralelos con comunicación entre ellos, algo indispensable en la implementación de AODV debido a la necesidad de ejecución rutinas de escucha y espera de respuestas de nodos vecinos, que pueden tomar algo de tiempo, y que la especificación de AODV, de hecho, garantiza la espera de un tiempo dado. La concurrencia de Go ocurre mediante las llamadas "Go-rutinas", que usan una tecnología de comunicación entre ellas por canales más segura que los hilos o procesos de otros lenguajes como Java o C#. Esta seguridad en la comunicación de diferentes procesos es indispensable para la implementación de un desarrollo crítico, como lo es la implementación de este protocolo de enrutamiento en red, que debe garantizar una adecuada comunicación entre sus procesos internos.

Al ser un lenguaje fuertemente tipado obliga al desarrollador a ser más cuidadoso con la declaración y uso de variables, de tal manera que cualquier error humano es normalmente advertido por el compilador a la hora de la compilación, evitando así errores en tiempo de ejecución. Esta es una característica importante para la comunicación de dispositivos conectados en una red, pues se debe minimizar al máximo cualquier error de implementación.

2.3. INFRAESTRUCTURA

Esta implementación de AODV está enfocada a la conexión de dispositivos de la internet de las cosas (IoT). Esto no es una limitación para su implementación en otras plataformas, ya que puede ser usada en cualquier dispositivo que esté conectado a una red TCP/IP y cuyo sistema operativo pueda ejecutar los binarios de AODV. Go puede generar sus binarios para sistemas Windows, Mac, Linux, y sus fuentes pueden ser portadas a cualquier sistema operativo compatible con POSIX, lo que asegura una amplísima portabilidad.

Las plataformas de microcomputadores que han emergido recientemente con soporte para extensión de hardware mediante puertos de Entrada/Salida de propósito general (GPIO) y sistemas operativos basados en Linux facilitan la implementación de dispositivos para IoT en redes sin infraestructura. Estas son algunas de las plataformas de hardware que pueden usarse implementando AODV:

- Raspberry Pi
- Tinker Board
- Beaglebone
- Intel Joule
- C.H.I.P.

3. ESTRATEGIA METODOLÓGICA

Para el desarrollo de esta implementación se analiza la especificación RFC 3561 en su totalidad para tener un panorama general del funcionamiento del protocolo, ¡y para poder establecer esta estrategia metodológica!

Una vez se tenga una visión general de la implementación, se entrará a analizar en detalle la estructura binaria de cada tipo de mensaje para poder decidir cómo implementarlos en el código. Se prevé la creación de una estructura por cada uno de los cuatro tipos de mensajes que implementa AODV con métodos de manejo con un estilo similar a la orientación a objetos.

Tras la implementación de la estructura de cada mensaje se procederá a desarrollar un cliente web que servirá inicialmente a modo de pruebas desde el cual se enviarán órdenes de descubrimiento de rutas para así poder hacer las primeras pruebas de envío y recepción de paquetes en red. Este cliente posteriormente se usará para la demostración en la exposición final del presente trabajo, permitiendo lanzar órdenes de descubrimiento de rutas, envío de paquetes, y monitorización del estado de las rutas.

Una vez finalizadas las primeras pruebas de comunicación en red se desarrollará la base de datos en la que cada nodo mantendrá su tabla de enrutamiento actualizada, así como su implementación en la tabla de enrutamiento del sistema operativo.

Finalmente se implementarán los funcionamientos más minuciosos que incluyen tiempos de espera de mensajes, condiciones de tratamiento de mensajes entrantes, control y prevención de bucles, lectura y manipulación de bits de marcas de los mensajes, condiciones de manejo de marcas de mensajes, conectividad local, configuración de parámetros predeterminados, consideraciones de seguridad, entre otros.

4. DESARROLLO E IMPLEMENTACIÓN

4.1. DEFINICIÓN DE PAQUETES AODV

La comunicación del algoritmo AODV se basa en cuatro paquetes de comunicación principales, que se detallarán a continuación.

4.1.1. RREQ (Route Request): Paquete enviado mediante difusión broadcast por un nodo cuando requiere conocer una ruta a un destino.

0 1 2 3 4 5 6 7	8	9	0	1	2	3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Tipo	**J**	**R**	**G**	**D**	**U**	**Reservado** / **Cuenta de saltos**
Identificador de mensaje						
Dirección IP de destino						
Número de secuencia de destino						
Dirección IP de origen						
Número de secuencia de origen						

Tabla 1: Estructura del paquete RREQ

Tipo:	1
J	Marca de unión, reservado para multicast
R	Marca de reparación, reservado para multicast
G	Marca para respuesta RREP al destino. Indica si se debería responder un mensaje RREP a la dirección indicado en el campo “Dirección IP de destino”
D	Marca de sólo destino. Indica que sólo el destino puede responder a esta solicitud.
U	Marca de número de secuencia desconocido. Indica que se desconoce el número de secuencia de destino.
Reservado	Enviar un 0. Se ignora en la recepción.
Cuenta de saltos	La cantidad de saltos por diferentes nodos desde el origen al nodo actual.
Identificador de mensaje	Un número único que identifica el mensaje RREQ.
Dirección IP de destino	La dirección IP del nodo destino para el que se quiere conocer la ruta.
Número de secuencia de destino	El número de secuencia que más recientemente ha recibido el origen para cualquier ruta hacia el destino.
Dirección IP de origen	La dirección IP del nodo que envía el mensaje RREQ.
Número de secuencia de origen	El número de secuencia del nodo actual para ser usado en la ruta hacia el origen de la solicitud.

Tabla 2: Definición del paquete RREQ

4.1.2. RREP (Route Reply): Paquete de respuesta de un nodo que conoce una ruta al destino solicitado en una petición RREQ.

0 1 2 3 4 5 6 7	8	9	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9 0 1
Tipo	**R**	**A**	**Reservado**	**Tam. prefijo**	**Cuenta de saltos**
Dirección IP de destino				
Número de secuencia de destino				
Dirección IP de origen				
Tiempo de vida				

Tabla 3: Estructura del paquete RREP

Tipo:	2
R	Marca de reparación, usado para multicast
A	Acuse de recibo requerido
Reservado	Enviar un 0. Se ignora en la recepción.
Tamaño de prefijo	Si es diferente a 0, indica que el próximo salto indicado puede ser usado para cualquier nodo con el mismo prefijo como destino.
Cuenta de saltos	La cantidad de saltos por diferentes nodos desde el origen RREQ al destino.
Dirección IP de destino	La dirección IP del nodo destino para el que el origen solicitó una ruta.
Número de secuencia de destino	El número de secuencia del destino asociado a la ruta.
Dirección IP de origen	La dirección IP del nodo que originó el mensaje RREQ al que se está respondiendo.
Tiempo de vida	Tiempo en milisegundos durante el que los nodos que reciben el paquete RREP considerarán la ruta como válida.

Tabla 4: Definición del paquete RREP

4.1.3. RERR (Route Error): Paquete de información de error en el procesamiento de un paquete previamente recibido.

0 1 2 3 4 5 6 7	8	9 0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9 0 1
Tipo	**N**	**Reservado**	**Cuenta de destinos**
Dirección IP de destino inalcanzable (1)			
Número de secuencia de destino inalcanzable (1)			
Dirección IP de destino inalcanzable adicional (opcional)			
Número de secuencia de destino inalcanzable adicional (opcional)			

Tabla 5: Estructura del paquete RERR

Tipo:	3
N	Marca de no borrar; establecida cuando un nodo ha realizado una reparación local de un enlace, pero ningún nodo debe eliminar esa ruta.
Reservado	Enviar un 0. Se ignora en la recepción.

Cuenta de destinos	La cantidad de destinos inalcanzables incluidos. Debe ser de al menos 1.
Dirección IP de destino inalcanzable	La dirección IP del destino inalcanzable debido a una ruptura del enlace.
Número de secuencia de destino inalcanzable	El número de secuencia del destino inalcanzable en la tabla de enrutamiento para el destino inalcanzable.

Tabla 6: Definición del paquete RERR

4.1.4. RREPACK (Route-Reply Ack): Paquete de acuse de recibo de RREP, si el nodo que lo originó lo solicita.

0 1 2 3 4 5 6 7	8 9 0 1 2 3 4 5
Tipo	**Reservado**

Tabla 7: Estructura del paquete RREPACK

Tipo:	4
Reservado	Enviar un 0. Se ignora en la recepción.

Tabla 8: Definición del paquete RREPACK

4.2. CONSTRUCCIÓN DE PAQUETES AODV EN GO

Una vez conocida y estudiada la estructura de los paquetes AODV, se procederá a implementarlos en el lenguaje de programación Go. Como se vio en la sección anterior, cada paquete está conformado por 24, 20, o 2 octetos. Dado que tienen un tamaño fijo, se decidió usar tipos personalizados basados en arrays para cada paquete.

```
type RREQ [24]byte

type RREP [20]byte

type RERR [20]byte

type RREPACK [2]byte
```

Figura 1: Definición de tipos personalizados basados en arrays de bytes

Como se vio anteriormente en la especificación de los paquetes, éstos están divididos en secciones a las que debemos acceder para obtener y manipular los datos. Para lograr esto debe haber funciones destinadas al acceso de cada uno de estos datos. Para ello se implementan métodos definidos en las estructuras de cada tipo.

Estas secciones comprenden desde un bit hasta cuatro bytes; lo que significa que para manejar estas secciones se requiere manipular desde un bit individual en un byte hasta cuatro bytes en conjunto, por lo que se recurre a funciones adicionales que hagan la respectiva manipulación bajo los parámetros de cada sección.

Para las secciones de marcas que ocupan un solo bit se usarán funciones externas para manipulación y obtención del bit a partir del byte, como se muestra en la imagen a continuación.

```go
func (paq *RREQ) J(v string) byte {
    if v == "0" || v == "1" {
        i, _ := strconv.Atoi(v)
        paq[1] = manipularBit(paq[1], 128, i)
    }
    return obtenerBit(paq[1], 128)
}

func (paq *RREQ) R(v string) byte {
    if v == "0" || v == "1" {
        i, _ := strconv.Atoi(v)
        paq[1] = manipularBit(paq[1], 64, i)
    }
    return obtenerBit(paq[1], 64)
}

func (paq *RREQ) G(v string) byte {
    if v == "0" || v == "1" {
        i, _ := strconv.Atoi(v)
        paq[1] = manipularBit(paq[1], 32, i)
    }
    return obtenerBit(paq[1], 32)
}

func (paq *RREQ) D(v string) byte {
    if v == "0" || v == "1" {
        i, _ := strconv.Atoi(v)
        paq[1] = manipularBit(paq[1], 16, i)
    }
    return obtenerBit(paq[1], 16)
}

func (paq *RREQ) U(v string) byte {
    if v == "0" || v == "1" {
        i, _ := strconv.Atoi(v)
        paq[1] = manipularBit(paq[1], 8, i)
    }
    return obtenerBit(paq[1], 8)
}
```

Figura 2: Métodos para manipulación de marcas de un bit en paquete RREQ

Las secciones que ocupan un byte completo son las más fáciles de manipular y se puede hacer directamente sin recurrir a funciones externas. La imagen a continuación muestra la sencillez en tres funciones para manipulación de saltos sin requerir funciones externas.

```go
func (paq *RREQ) Saltos() uint {
    return uint(paq[3])
}

func (paq *RREQ) EstablecerSaltos(slts uint) {
    paq[3] = byte(slts)
}

func (paq *RREQ) IncrementarSalto() {
    if paq[3] < 255 {
        paq[3]++
    } else {
        log.Print("Se alcanzaron más de 255 saltos.")
    }
}
```

Figura 3: Métodos para manipulación de un byte entero en paquete RREQ

Sin embargo, las secciones que agrupan más de un byte sí que pueden requerir el tratamiento de funciones externas para su manipulación, como se muestra en la imagen a continuación.

```go
func (paq *RREQ) Destino(ip string) string {
    Ip := net.ParseIP(ip)
    if Ip != nil {
        paq[8]=Ip[12]
        paq[9]=Ip[13]
        paq[10]=Ip[14]
        paq[11]=Ip[15]
    }
    return formatearIP(paq[8], paq[9], paq[10], paq[11])
}

func (paq *RREQ) ObtenerNúmeroSecuencia(n string) uint {
    B1, B2, B3, B4 := paq.BytesNúmeroSecuencia(n)
    bin1 := decimalABinario(uint(paq[B1]), 8)
    bin2 := decimalABinario(uint(paq[B2]), 8)
    bin3 := decimalABinario(uint(paq[B3]), 8)
    bin4 := decimalABinario(uint(paq[B4]), 8)
    bin := bin1+bin2+bin3+bin4
    num := binarioADecimal(bin)
    return uint(num)
}
```

Figura 4: Métodos para manipulación de cuatro bytes en paquete RREQ

Se usan algunas funciones predeterminadas de las librerías propias de Go siempre que sea posible, sin embargo en la mayoría de casos se requiere construir funciones propias para manipulación de bits y múltiples bytes. Estas funciones no se exponen directamente en este documento, pero se encuentran en el código fuente anexo *aodv_general.go*.

4.3. TABLAS DE ENRUTAMIENTO Y BASES DE DATOS

AODV requiere mantener una tabla de enrutamiento con todas las rutas actualizadas. Para ello se define que se mantendrá esta tabla en una base de datos, a partir de la cual se podrá posteriormente actualizar la tabla de enrutamiento nativa del sistema operativo. Se establece que este método es más eficiente y rápido que la consulta y manipulación directa de la tabla de enrutamiento nativa del sistema operativo, además de permitir al código ser más fácilmente portable al no requerir diferentes métodos de manipulación de tablas de enrutamiento para diferentes sistemas operativos, sin contar además que muchos sistemas operativos basados en Unix tienen diferencias en la implementación de las tablas de enrutamiento. Esto no significa que pueda haber diferencias en las tablas de enrutamiento, puesto que se asegurará que en la misma función de actualización de la base de datos se dispare una go-rutina que actualice la tabla nativa del sistema operativo.

```
CREATE TABLE rutas (
    destino VARCHAR(15) NOT NULL PRIMARY KEY,
    secuencia INTEGER NOT NULL,
    saltos INTEGER NOT NULL,
    proximo_salto VARCHAR(15) NOT NULL,
    vigencia INTEGER NOT NULL,
    valida INTEGER NOT NULL DEFAULT 0
);
```

Figura 5: Creación de tabla de enrutamiento en base de datos

Además de la tabla de enrutamiento, se requerirán también otras tablas indispensables para la operación de AODV: Tablas para el almacenamiento temporal de paquetes RREQ y RREP enviados y recibidos con el fin de validar el reenvío de paquetes broadcast para impedir el reprocesamiento de paquetes, evitando así bucles infinitos; y también para comparar la información de diferentes paquetes de posibles diferentes rutas a un mismo destino para tomar la mejor decisión. Por último, también se requerirá una tabla donde almacenar y mantener actualizada la información de direccionamiento y número de secuencia del propio nodo, necesaria para el descubrimiento de nuevas rutas.

```
CREATE TABLE rrep_enviados (
    origen VARCHAR(15) NOT NULL,
    destino VARCHAR(15) NOT NULL,
    identificador INTEGER NOT NULL,
    tiempo TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL
);
CREATE TABLE rreq_recibidos (
    origen VARCHAR(15) NOT NULL,
    destino VARCHAR(15) NOT NULL,
    identificador INTEGER NOT NULL,
    tiempo TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL
);
```

Figura 6: Creación de tablas temporales de paquetes RREQ y RREP enviados y recibidos

Como motor de base de datos se usa SQLite debido a que permite mantener de manera muy sencilla una alta portabilidad sin depender de una instalación adicional en el sistema

operativo. Además, los requerimientos no requieren de un motor muy robusto, puesto que la mayoría de tabla sólo almacenarán datos temporales y la tabla de enrutamiento no podrá crecer demasiado debido a que las rutas antiguas expiran. Dadas estas características, el uso de SQLite es muy adecuado y eficiente.

Las funciones de acceso a la base de datos se manejan externas a los métodos principales asociados a los paquetes, y son llamadas desde éstos.

```
func buscarRecienteRREPBD(origen, destino string) bool {
    segs := int(PATH_DISCOVERY_TIME/1000);
    segsDesde := strconv.Itoa(segs)
    bd := conectarBD()
    defer bd.Close()
    filas, err := bd.Query("SELECT origen FROM rrep_recibidos
        WHERE origen='"+origen+"' AND destino='"+destino+"' AND
        DATETIME(tiempo)>DATETIME('now','-"+segsDesde+" seconds')
        LIMIT 1")
    defer filas.Close()
    if err != nil {
        log.Fatal("Error en consulta de RREP reciente:", err)
    }
    var exst bool
    for filas.Next() {
        exst = true
    }
    if !exst {
        agregarRecienteRREPBD(origen, destino)
    }
    return exst
}
```

Figura 7: Función de consulta de paquetes RREP en base de datos

En el código fuente anexo *aodv_bd.go* se encuentran todas las funciones de acceso a base de datos para la verificación de paquetes broadcast, consulta de información del propio nodo, y acceso y actualización a la tabla de enrutamiento.

4.4. RECEPCIÓN Y MANEJO DE PAQUETES

La recepción de paquetes se hará mediante una función maestra que estará escuchando permanentemente las conexiones UDP entrantes. Al recibir datos desde una conexión UDP, se enviarán a otra función que validará si se trata de un paquete AODV correcto, en cuyo caso será tratada con otros métodos asociados al paquete.

```
func Escuchar(){
    dirEscucha, _ := net.ResolveUDPAddr("udp", origenDeEscucha)
    escucha, _ := net.ListenUDP("udp", dirEscucha)
    for {
        msj := make([]byte, 24)
        ctdd, orgn, _ := escucha.ReadFromUDP(msj)
        var paqAODV bool
        var tipoMsj string
        paqAODV, tipoMsj = AnalizarPaquete(msj, ctdd)
        if paqAODV {
            if tipoMsj == "RREQ" {
                rreq := new(RREQ)
                rreq.Componer(msj)
                rreq.Manejar(orgn.IP.String())
            } else if tipoMsj == "RREP" {
                rrep := new(RREP)
                rrep.Componer(msj)
                rrep.Manejar(orgn.IP.String())
            } else if tipoMsj == "RERR" {
                rerr := new(RERR)
                rerr.Componer(msj)
                rerr.Manejar(orgn.IP.String())
            } else if tipoMsj == "RREPACK" {
                rrepack := new(RREPACK)
                rrepack.Componer(msj)
                rrepack.Manejar(orgn.IP.String())
            }
        }
    }
}
```

Figura 7: Función de consulta de paquetes RREP en base de datos

En la *Figura 7* se muestra la función maestra de escucha de conexiones UDP entrantes: Abre la dirección y el puerto de escucha, y al recibir datos los almacena en un slice y los envía a la función *AnalizarPaquete* para verificar que se trate de un paquete AODV correcto.

En la *Figura 8* se ve cómo la función *AnalizarPaquete* verifica los datos recibidos en función de la cantidad de bytes total recibidos, y buscando valores válidos para bytes indispensables según el tipo de paquete, como direcciones IP de origen y destino o cuenta de saltos.

```go
func AnalizarPaquete(pqt []byte, ctdd int) (bool, string) {
    var valido bool
    var tipo string
    if pqt[0] == 1 && pqt[8] > 0 && pqt[11] > 0 && pqt[16] > 0 &&
        pqt[19] > 0 && ctdd == 24 {
        //Tipo 1, Primer (8) y último (11) byte de destino, y
        primero (16) y último (19) de origen, y de 24 bytes
        tipo = "RREQ"
    } else if pqt[0] == 2 && pqt[4] > 0 && pqt[7] > 0 && pqt[12] >
         0 && pqt[15] > 0 && ctdd == 20 {
        //Tipo 2, Primer (4) y último (7) byte de destino, y
        primero (11) y último (15) de origen, y de 20 bytes
        tipo = "RREP"
    } else if pqt[0] == 3 && pqt[3] > 0 && pqt[4] > 0 && pqt[7] >
        0 && ctdd == 20 {
        //Tipo 3, Cuenta de destinos inalcanzables (3), primer
        (4) y último (7) byte de destino, y de 20 bytes
        tipo = "RERR"
    } else if pqt[0] == 4 && ctdd == 2 {
        //Tipo 4, y de 2 bytes
        tipo = "RREPACK"
    }
    if tipo != "" {
        valido = true
    }
    return valido, tipo
}
```

Figura 8: Análisis de paquete recibido, validación de paquete AODV correcto

Si el paquete entrante no es un paquete AODV válido, simplemente se descarta. En caso de validarse como un paquete AODV válido, se procede a formatear en su tipo de paquete y se le da el manejo necesario. Para ello cada tipo de paquete cuenta con dos métodos que realizan esta validación: *Componer* y *Manejar*.

```go
func (paq *RREQ) Componer(r []byte) {
    for i := 0; i <= 23; i++ {
        paq[i]=r[i]
    }
}

func (paq *RREP) Componer(r []byte) {
    for i := 0; i <= 19; i++ {
        paq[i]=r[i]
    }
}
```

Figura 9: Métodos Componer de paquetes RREQ y RREP

El método *Componer*, como se muestra en la *Figura 9*, copia los datos entrantes de una conexión UDP a un paquete de su propio tipo, para posteriormente ser manipulado fácilmente con los métodos descritos en la sección 4.2.

Por último, el método *Manejar* hace validaciones para tratar cada mensaje entrante, según la especificación del protocolo. En la *Figura 10* se ve por ejemplo el manejo de los paquetes RREP entrantes.

```
func (paq *RREP) Manejar(origenUDP string) {
    origen := paq.Origen("")
    destino := paq.Destino("")
    yaRecibido := buscarRecienteRREPBD(origen, destino)
    if origenUDP == direcciónLocal || origen == direcciónLocal ||
        yaRecibido {
        //Descartado
    } else {
        paq.IncrementarSalto()
        if origen == direcciónLocal {
            rutaDestinoActualizarBD(destino, paq.
                NúmeroSecuenciaDestino(), paq.Saltos(), origenUDP)
        } else {
            //Buscar ruta de destino:
            exst, ruta := rutaDestinoBuscarBD(destino)
            if exst && ruta.Vigente && ruta.Válida && ruta.
                Secuencia>=paq.NúmeroSecuenciaDestino() {
            } else {
                rutaDestinoActualizarBD(destino, paq.
                    NúmeroSecuenciaDestino(), paq.Saltos(),
                    origenUDP)
            }
            paq.Enviar()
        }
    }
}
```

Figura 10: Método de manejo de paquetes RREP entrantes

El método de la figura anterior se muestra simplificado, sin manejo de errores, y sin depuración. En el código fuente anexo *aodv.go* se encuentra más detallado, así como los demás métodos de manejo de los otros tipos de paquetes. También cabe destacar que este método llama a funciones externas e incluso otros métodos que se verán en una sección posterior (como *Enviar*), que ayudan a organizar de manera modular el código, permitiendo una gran simplificación.

Este manejo de paquetes entrantes sigue la especificación AODV, que se ilustra a manera de diagrama de flujo en las figuras a continuación.

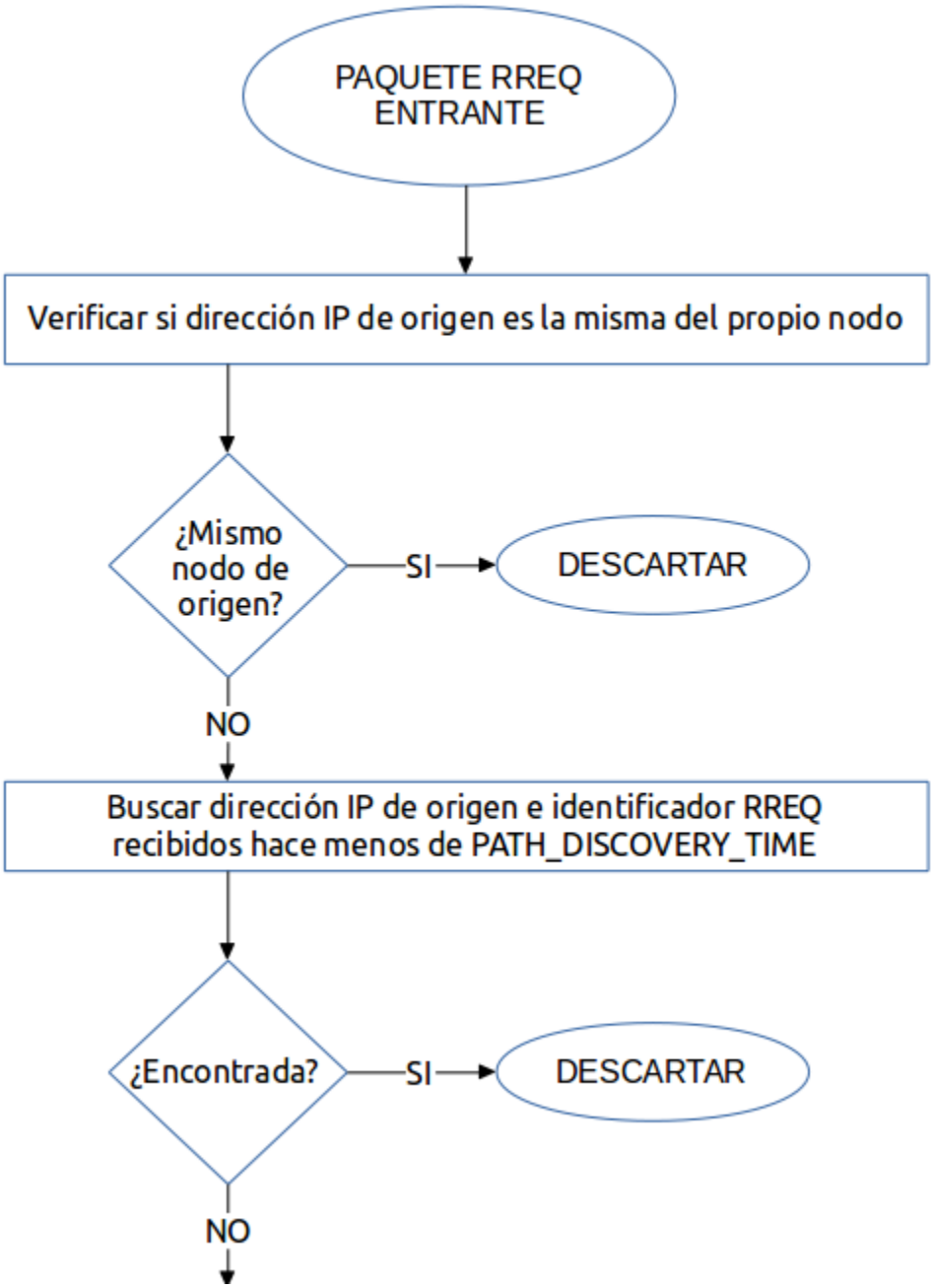

Figura 11: Diagrama de flujo de manejo de paquete RREQ entrante (parte 1)

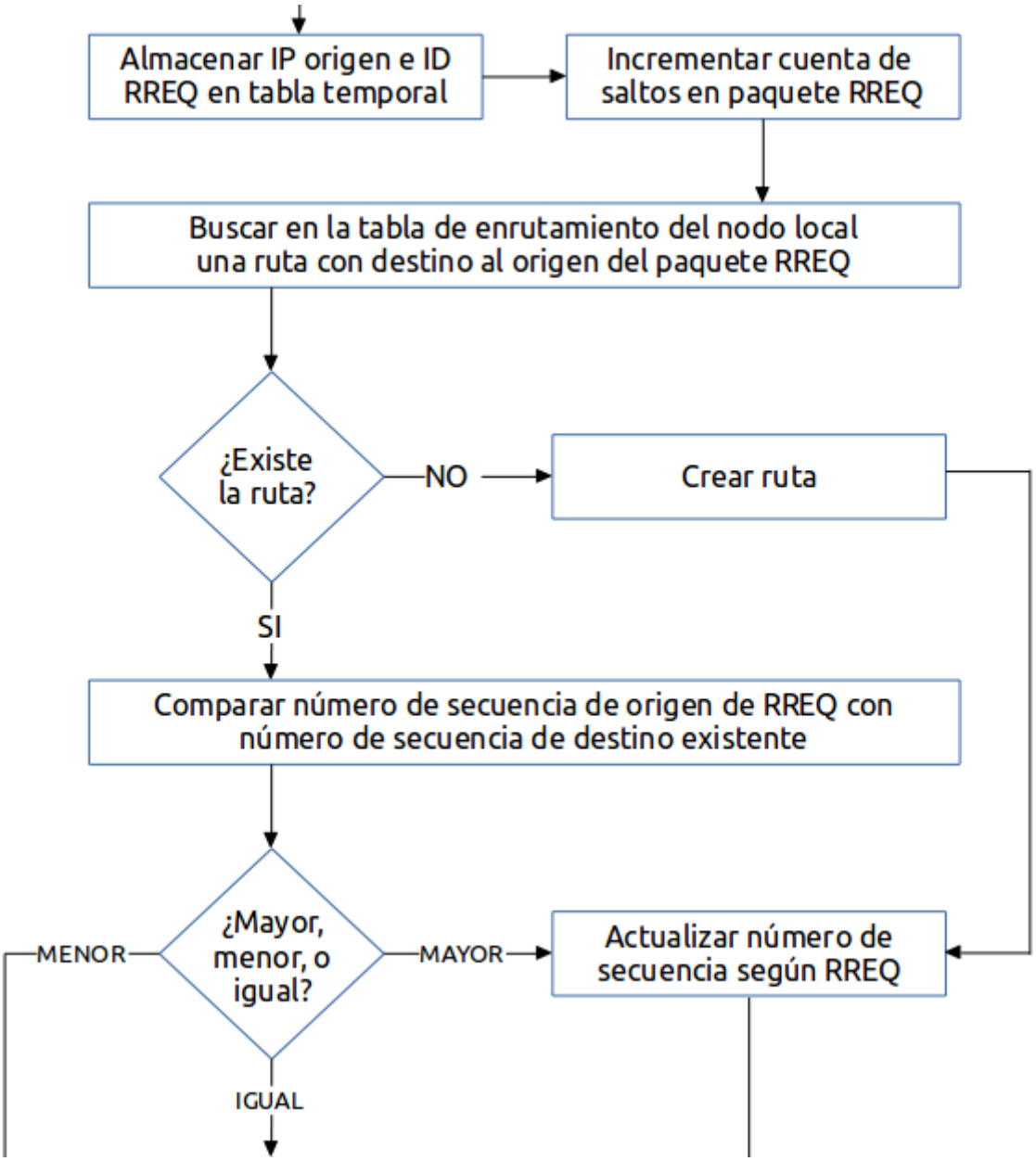

Figura 12: Diagrama de flujo de manejo de paquete RREQ entrante (parte 2)

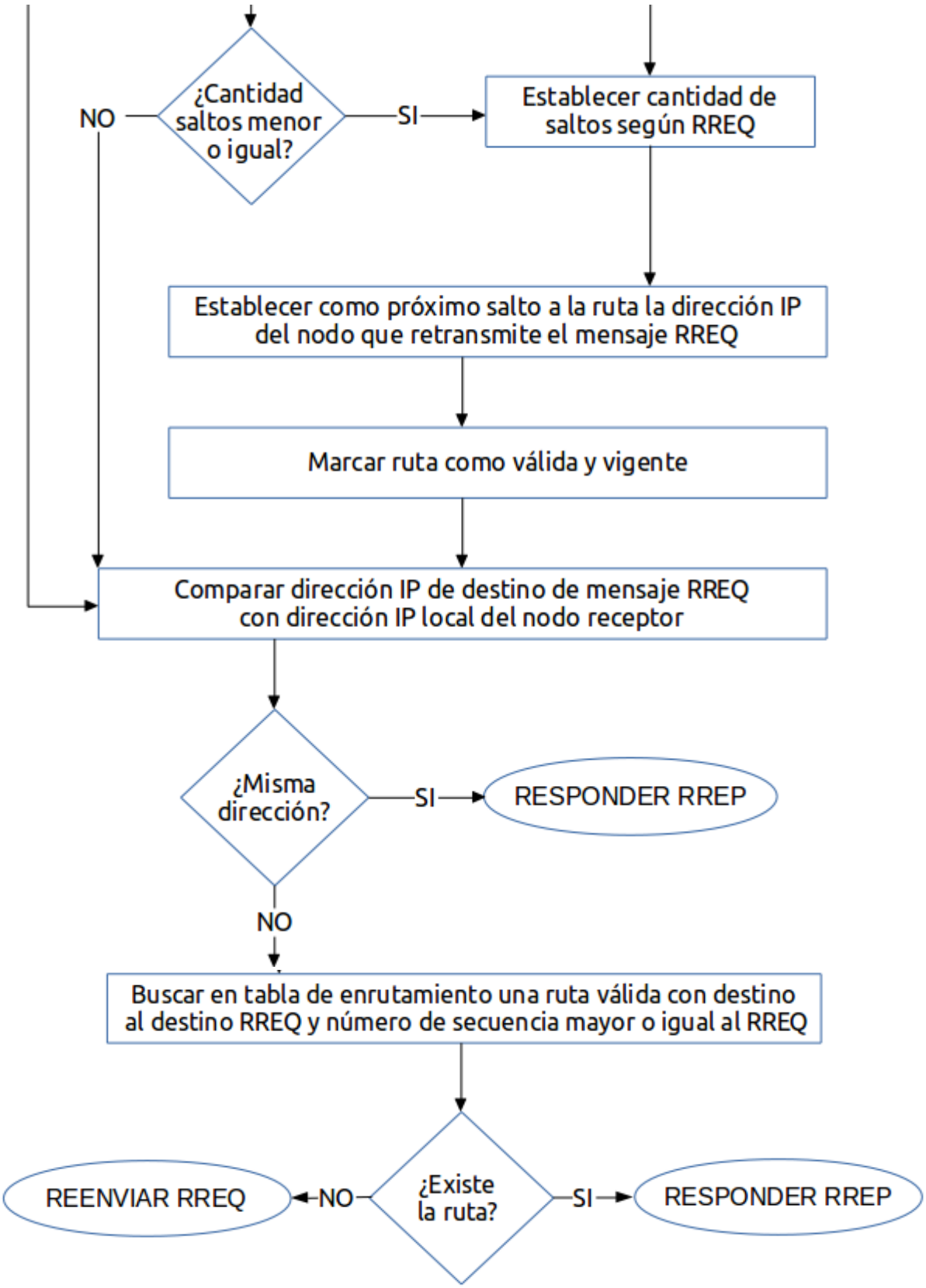

Figura 13: Diagrama de flujo de manejo de paquete RREQ entrante (parte 3)

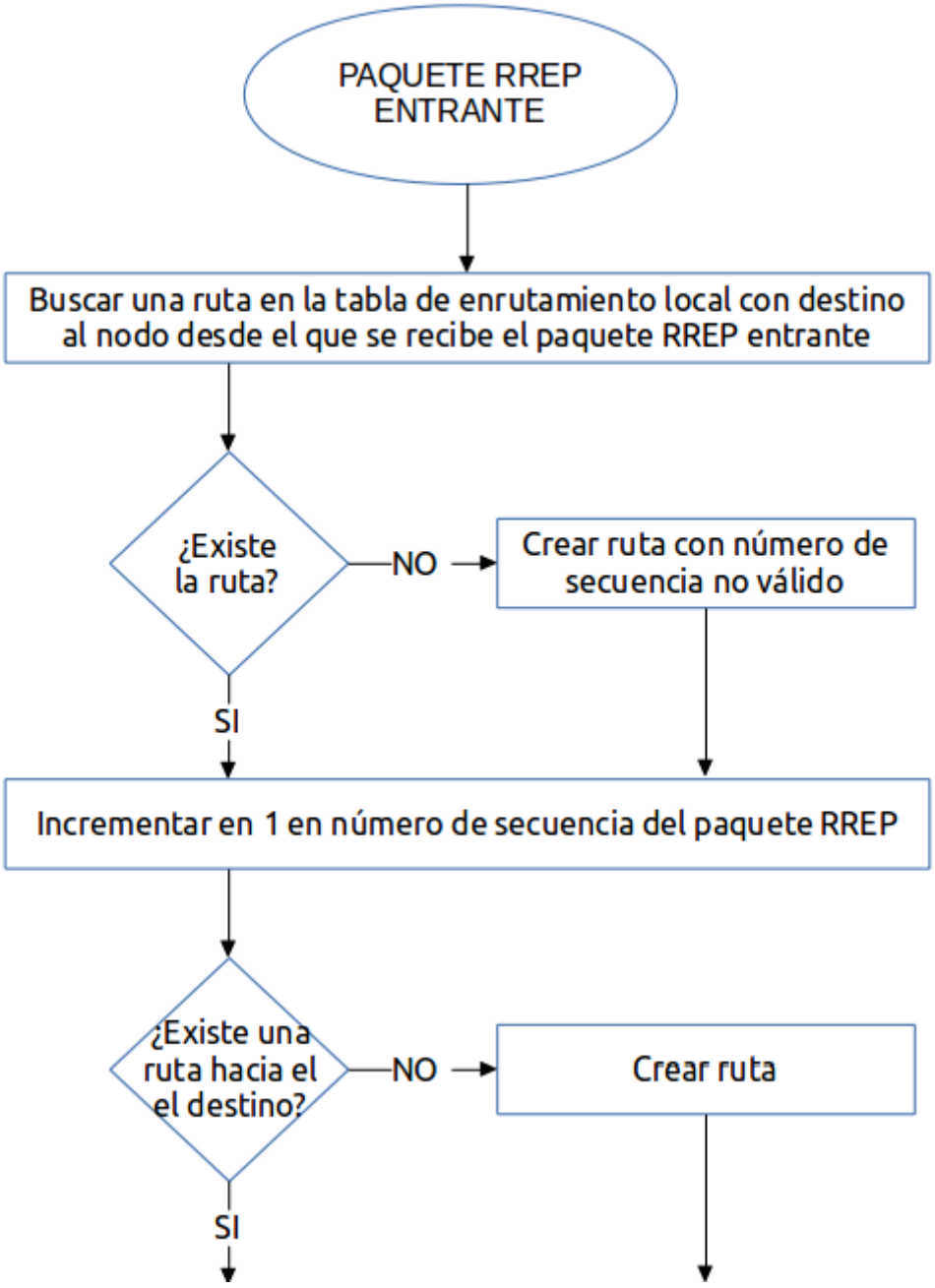

Figura 14: Diagrama de flujo de manejo de paquete RREP entrante (parte 1)

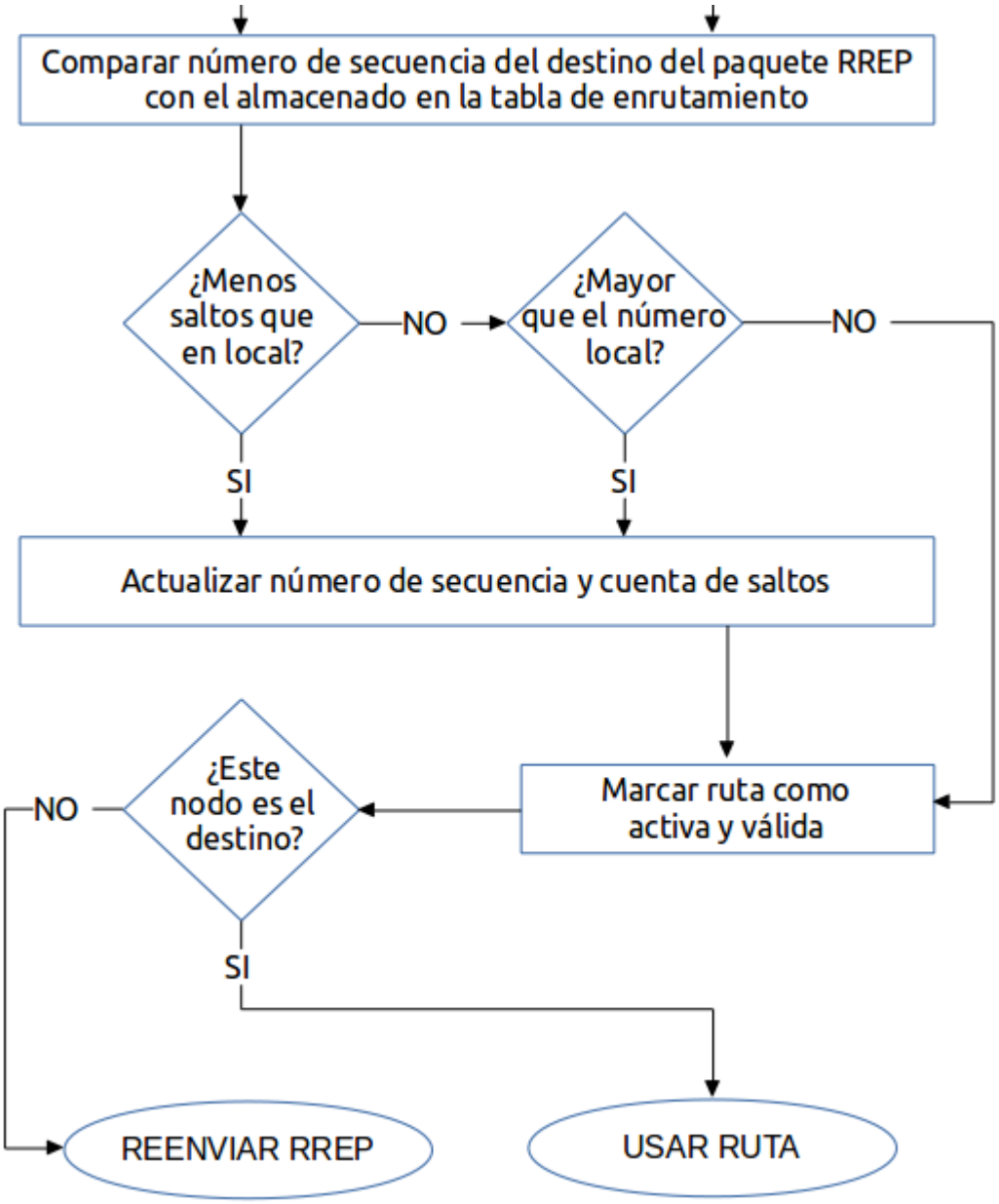

Figura 15: Diagrama de flujo de manejo de paquete RREP entrante (parte 2)

4.5. CONSTRUCCIÓN Y ENVÍO DE PAQUETES

Anteriormente se vio el manejo de paquetes entrantes. Ahora se detallará el originamiento de paquetes por el mismo nodo.

Cuando un nodo requiere conocer la ruta hacia un destino, origina un paquete RREQ, para lo cual se usa el método indicado en la *Figura 16*, en el cual se observa el establecimiento de los campos de origen y destino, los saltos, y el número de secuencia de origen o una marca de número de secuencia desconocido.

```
func (paq *RREQ) Construir(destino string) {
    if !fmtIP.MatchString(destino) {
        log.Println("FORMATO DE DIRECCIÓN DE DESTINO INCORRECTO")
        log.Fatal("\tRecibido:\t"+destino)
    }
    exstBD, ruta := rutaDestinoBuscarBD(destino)
    paq[0] = 1
    paq.Destino(ruta.Destino)
    paq.Origen(direcciónLocal)
    paq.EstablecerSaltos(ruta.Saltos)
    if !exstBD {
        paq.U("1")
    }
    paq.EstablecerNúmeroSecuenciaOrigen()
}
```

Figura 16: Método de construcción de un paquete RREQ

La construcción de un paquete RREP se realiza a partir de un mensaje RREQ, si el nodo conoce la ruta al destino o es el propio destino, como se observa en la *Figura 17*. Se observa el establecimiento de datos a partir del paquete RREQ entrante.

```
func (paq *RREP) Construir(rreq *RREQ, slts uint, sec uint) {
    paq[0] = 2
    paq.Destino(rreq.Destino(""))
    paq.Origen(rreq.Origen(""))
    paq.EstablecerSaltos(slts)
    paq.EstablecerNúmeroSecuenciaDestino(sec)
}
```

Figura 17: Método de construcción de un paquete RREP

Una vez creados, los paquetes se envían por difusión broadcast UDP, como se observa en la *Figura 18*.

```go
func (paq *RREQ) Enviar() {
    addr, err := net.ResolveUDPAddr("udp", destinoDeDifusión)
    if err != nil {
        log.Fatal(err)
    }
    c, err := net.DialUDP("udp", nil, addr)
    p := []byte(paq[:])
    c.Write(p)
    if depuración {
        log.Print("PAQUETE RREQ ENVIADO:")
        log.Print("\tPaquete:\t", p)
    }
}
```

Figura 18: Envío de paquete RREQ por difusión

4.6. PRUEBAS Y CLIENTE WEB

Se incluye el desarrollo de un cliente web para pruebas y posterior simulación de aplicación que solicite rutas y envío de paquetes. Su código en detalle se encuentra en el anexo *aodv_web.go*.

```go
func servidorWeb(){
    receptor := http.NewServeMux()
    receptor.HandleFunc("/", manejadorWeb)
    servidor := &http.Server{
        Addr:           ":6565",
        Handler:        receptor,
        ReadTimeout:    30 * time.Second,
        WriteTimeout:   30 * time.Second,
        MaxHeaderBytes: 1 << 20, //1MB
    }
    e := servidor.ListenAndServe()
    if e != nil {
        log.Fatal(e)
    }
}
```

Figura 19: Servidor web para ambiente de pruebas

Este cliente permitirá la realización de pruebas iniciales, y demostración a modo de simulación de aplicación en red (Se enviarán órdenes reales de descubrimiento de rutas y envío de paquetes). Éste se ejecutará en un proceso en paralelo con el proceso de escucha de mensajes entrantes, y el proceso de envío de datos.

Para iniciar con las pruebas en un entorno simulado, se requiere la conexión de dos dispositivos a una red inalámbrica Ad-Hoc, por lo que se empezará con la configuración de esta red. Para ello se usará dos portátiles en los que se configurará la red Ad-Hoc.

```
saulortega@saul-eos:~$ sudo iwconfig wlan1 mode ad-hoc
saulortega@saul-eos:~$ sudo iwconfig wlan1 essid "AODV"
```

Figura 20: Configuración de red Ad-Hoc

Una vez configurada, se puede conectar a la red desde otra terminal, como se ve en las imágenes a continuación.

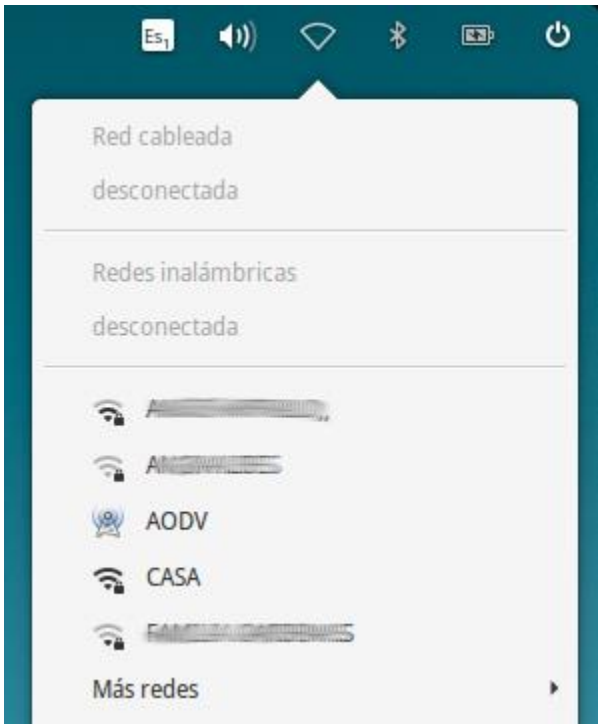

Figura 21: Red Ad-Hoc desde una máquina Linux

Figura 22: Red Ad-Hoc desde una máquina Windows

Una vez conectada la red, se procede a compilar en un ejecutable el código desarrollado para distribuirlo en las máquinas de la red en que se probará. Se usan dos máquinas con sistema operativo Linux, en una de las cuales se usará el cliente web previamente descrito para realizar las pruebas de descubrimiento de rutas.

Figura 23: Interfaz web para pruebas de descubrimiento de rutas

El funcionamiento de la interfaz del cliente web para pruebas es tan sencillo como escribir una dirección IP y pulsar un botón para que el algoritmo AODV haga la búsqueda de la ruta. El código desarrollado cuenta además con un amplio registro de logs para poder hacer seguimiento y diagnóstico de los paquetes.

La primera prueba a realizar consta de la búsqueda de una ruta con destino hacia el mismo nodo de origen, es decir que el origen y el destino es el mismo. Se muestra el resultado a continuación.

Figura 24: Descubrimiento de ruta con destino al mismo nodo de origen

Como era de esperarse, se retorna la ruta con 0 saltos, pues el destino es el mismo origen. Esto es simplemente una prueba superficial para verificar que al menos el cliente web funcione correctamente. Ahora se hará una prueba con destino a la otra máquina conectada en la red.

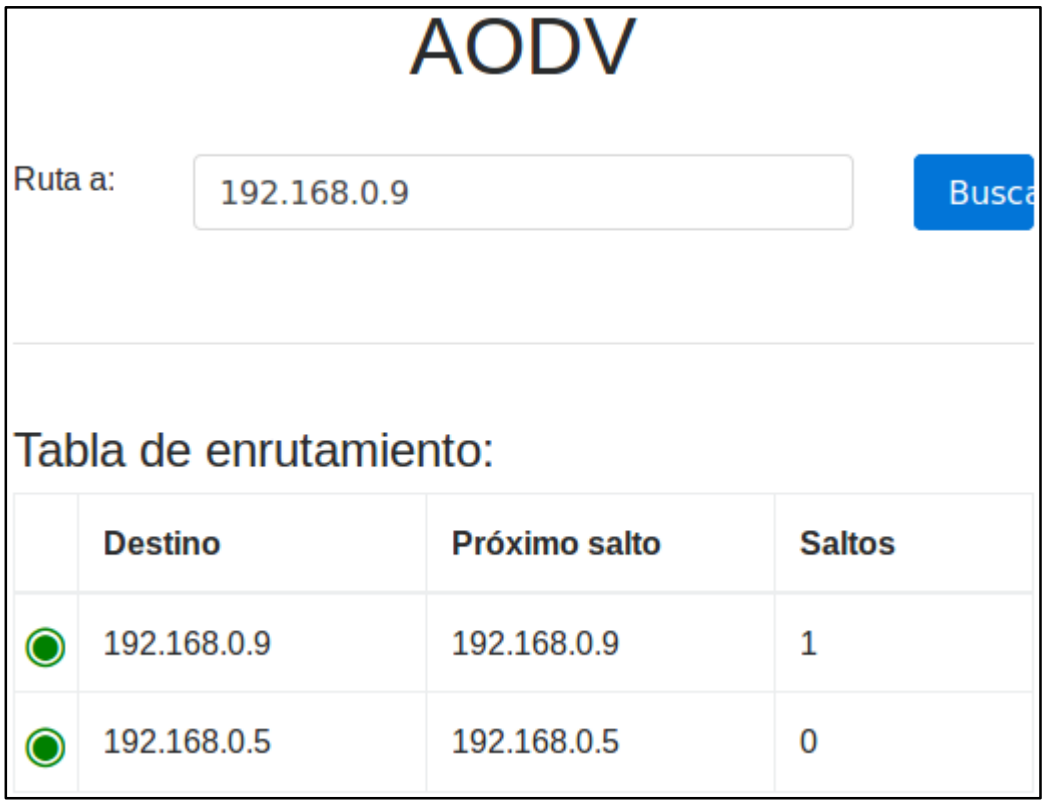

Figura 25: Descubrimiento de ruta con destino a otro nodo en la red

En este caso, se muestra la respuesta del nodo destino con un salto de 1, lo que indica que no hay ningún nodo intermediario entre ellos. Por eso mismo, el próximo salto de la tabla de enrutamiento es el mismo destino.

Pero es necesario ver el registro en la consola de ambos nodos para ver el comportamiento del algoritmo.

```
NÚMERO DE SECUENCIA DE ORIGEN INCREMENTADO:
    Número de secuencia antes:      32
    Número de secuencia después:    33
IDENTIFICADOR ACTUAL RREQ BUSCADO:
    Destino:        192.168.0.9
    Identificador:  24
CONEXIÓN UDP ENTRANTE:
    Bytes recibidos:        24
    Dirección origen:       192.168.0.5:57643
    Contenido recibido:     [1 0 0 0 0 0 0 25 192 168 0 9 0 0 0 0 192 168 0 5 0
    Paquete AODV válido:    true
PAQUETE RREQ ENVIADO:
    Paquete:        [1 0 0 0 0 0 0 25 192 168 0 9 0 0 0 0 192 168 0 5 0 0 0 33]
```

Figura 26: Log del comportamiento del envío del paquete RREQ

En la *Figura 26* se ve cómo en principio se incrementa el número de secuencia el propio nodo de origen, y se obtiene el identificador del destino en base de datos local. Posteriormente se construye y envía por difusión broadcast el paquete RREQ.

En la *Figura 27* se ve el comportamiento del nodo destino que recibe el paquete RREQ:

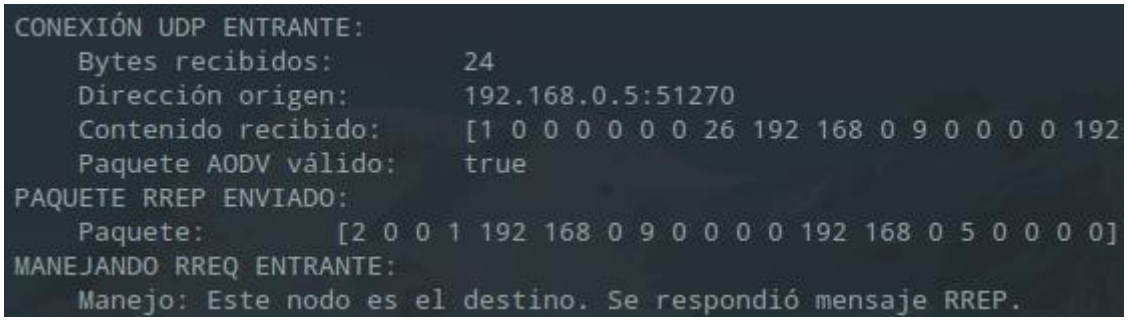

Figura 27: Log del comportamiento de la recepción del paquete RREQ y envío de RREP

Cuando el nodo recibe el paquete RREQ, comprueba el destinatario. Como él es el destino, procede a construir un paquete RREP y responderlo. En la *Figura 28* se ve cómo, de vuelta en el nodo origen, se recibe el paquete RREP del destino.

Figura 28: Log del comportamiento de la recepción del paquete RREP

Aquí se recibe adecuadamente la respuesta a la solicitud y se procede a guardar la ruta en la tabla de enrutamiento. En el ambiente de pruebas, también se responde al cliente web la información sobre la ruta.

En caso de que se rompa el enlace con una ruta establecida, AODV, al ser un protocolo reactivo, reaccionará la próxima vez que se requiera una ruta a éste destino. Para probar esto, se procede a desconectar la red en el nodo destino, y se realiza nuevamente la petición. En la siguiente imagen se verá el resultado en el cliente web.

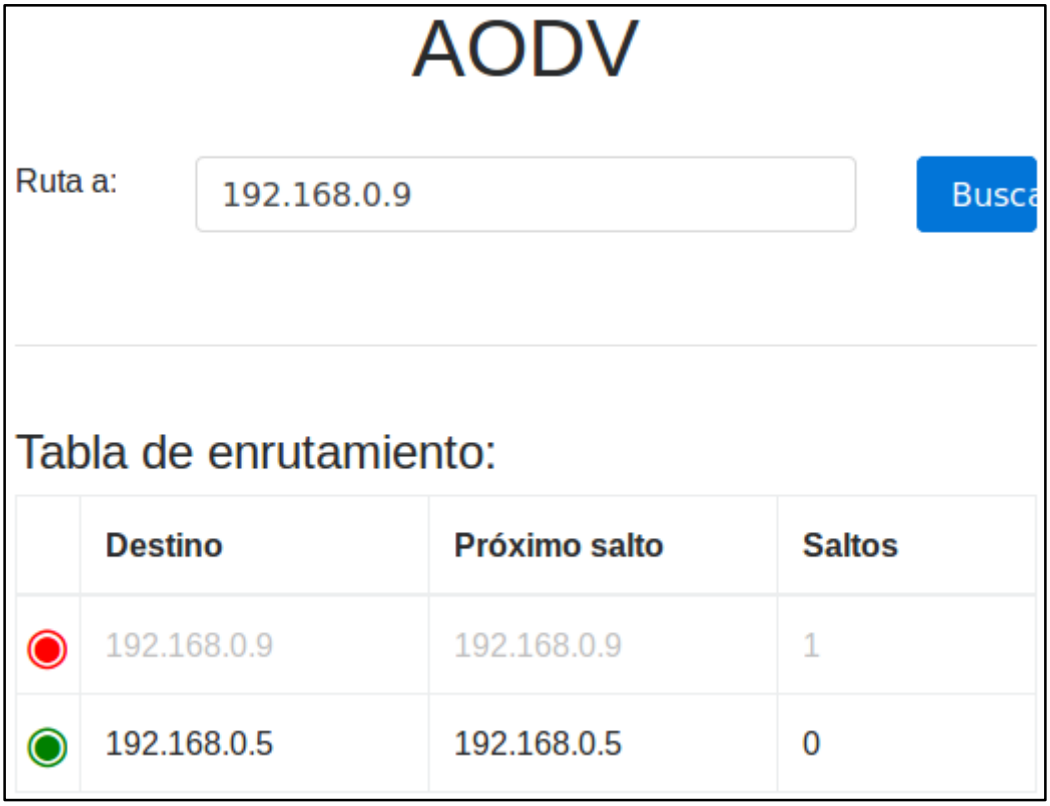

Figura 29: Cliente web mostrando enlace caído

En cualquier momento que se restablezca el enlace, ante una nueva solicitud de ruta, el algoritmo detectará y restablecerá el enlace.

4.7. PRUEBAS EN UN ENTORNO REALÍSTICO CON RASPBERRYS

El algoritmo fue compilado para arquitectura ARM y se procedió a crear una red con 5 Raspberrys Pi para la simulación del entorno. A continuación se muestra el esquema de red.

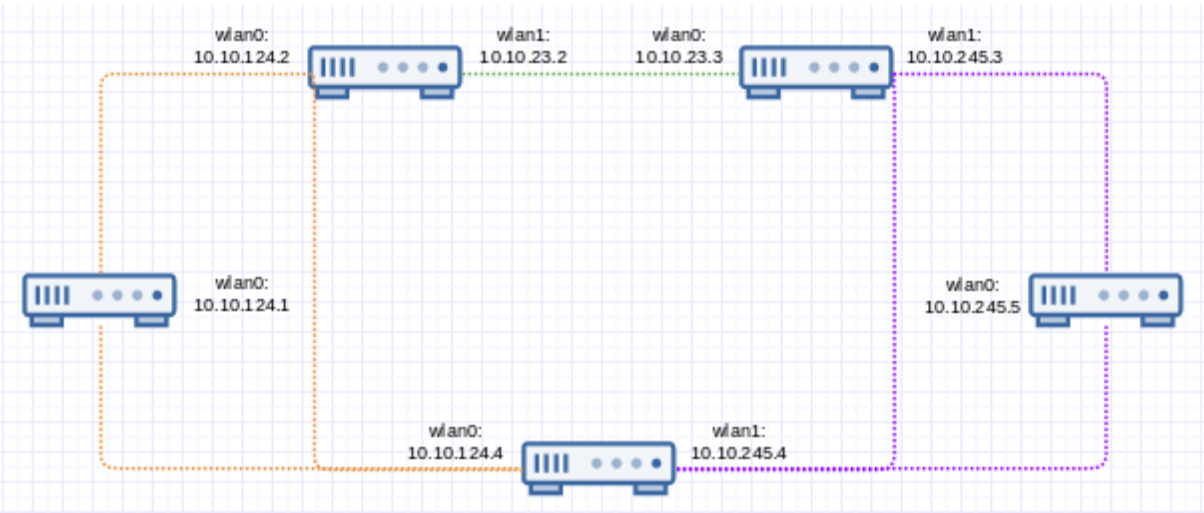

Figura 30: Diagrama de red de Raspberrys

La anterior estructura se armó con cinco Raspberry Pi, tres de ellas usando dos tarjetas de red para conectarse a dos diferentes redes. Hay tres redes, que se señalan en la gráfica anterior con tres diferentes colores, de tal manera que en dos de ellas hay tres nodos conectados y en la tercera hay dos. Cuando el nodo 1 requiera llegar al nodo 5, preferirá la ruta más corta, pasando por el nodo 4. Sin embargo, al caerse el nodo 4, la ruta se irá a través de los nodos 2 y 3.

Figura 31: Montaje en rack de las 5 Raspberry Pi (Leds rojos) en red

Figura 31: Raspberry Pi con tarjeta de red externa

Con esta configuración de hardware, se procede a realizar nuevamente una prueba de accesibilidad a todos los nodos.

Tabla de enrutamiento:

	Destino	Próximo salto	Saltos
◉	10.10.245.5	10.10.124.4	2
◉	10.10.245.4	10.10.124.4	1
◉	10.10.245.3	10.10.124.4	2
◉	10.10.23.3	10.10.124.2	2
◉	10.10.23.2	10.10.124.2	1
◉	10.10.124.4	10.10.124.1	1
◉	10.10.124.2	10.10.124.1	1
◉	10.10.124.1	10.10.124.1	0

Figura 32: Accesibilidad a todos los nodos de la red conectados

La interconexión de los nodos funciona adecuadamente, como se ve en la figura. Los nodos 1, 2, y 4 están conectados entre sí con una única red Ad-Hoc. Los nodos 3, 4, y 5 también están conectados entre sí usando otra red Ad-Hoc. Finalmente, los nodos 2 y 3 forman la tercera red, la cual servirá como ruta secundaria en caso de caerse el nodo 4. Se procede entonces a desactivar las redes del nodo 4, para simular una caída de dicho nodo.

```
root@RACK04-SW:/home/pi# ifdown wlan0
root@RACK04-SW:/home/pi# ifdown wlan1
root@RACK04-SW:/home/pi#
```

Figura 33: Desactivación de redes del nodo 4

Una vez hecho esto, se procede a probar nuevamente con el objetivo de que el algoritmo recalcule la ruta al nodo destino. Como se ve en la *Figura 34*, las direcciones IP del nodo 4 son inaccesibles, y, lo más importante de la demostración, la ruta de acceso al nodo 5 cambia su próximo salto, usando el nodo 2, y aumentando en uno la cantidad de saltos que se requieren para llegar a él.

Tabla de enrutamiento:

	Destino	Próximo salto	Saltos
●	10.10.245.5	10.10.124.2	3
●	10.10.245.4	10.10.124.4	1
●	10.10.245.3	10.10.124.4	2
●	10.10.23.3	10.10.124.2	2
●	10.10.23.2	10.10.124.2	1
●	10.10.124.4	10.10.124.1	1
●	10.10.124.2	10.10.124.1	1
●	10.10.124.1	10.10.124.1	0

Figura 34: Accesibilidad con el nodo 4 caído

La tabla de enrutamiento del sistema operativo también se actualiza según el descubrimiento de rutas óptimas del algoritmo.

```
root@RACK05-SW:/home/pi/aodv# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         ACiscoK05       0.0.0.0         UG    0      0        0 eth0
default         ACiscoK05       0.0.0.0         UG    202    0        0 eth0
10.10.23.2      10.10.245.3     255.255.255.255 UGH   0      0        0 wlan0
10.10.23.3      10.10.245.3     255.255.255.255 UGH   0      0        0 wlan0
10.10.124.1     10.10.245.4     255.255.255.255 UGH   0      0        0 wlan0
10.10.124.2     10.10.245.4     255.255.255.255 UGH   0      0        0 wlan0
10.10.124.4     10.10.245.4     255.255.255.255 UGH   0      0        0 wlan0
10.10.245.0     *               255.255.255.0   U     0      0        0 wlan0
10.10.245.3     10.10.245.5     255.255.255.255 UGH   0      0        0 wlan0
10.10.245.4     10.10.245.5     255.255.255.255 UGH   0      0        0 wlan0
10.10.245.5     10.10.245.5     255.255.255.255 UGH   0      0        0 wlan0
```

Figura 35: Tabla de enrutamiento del nodo 5

5. CONCLUSIONES

El algoritmo AODV tiene grandes aplicaciones en dispositivos interconectados en una red de datos Ad-hoc. Estas redes, al funcionar sin infraestructura centralizada, no cuentan con un nodo central (enrutador, AP) que les comunique su información de direccionamiento IP, por lo que este algoritmo se encarga de descubrir y establecer las rutas adecuadas para la comunicación entre diferentes nodos.

Esta especificación implementada en Go se puede ejecutar en procesadores con arquitectura ARM, que son usados por la gran mayoría de dispositivos de microcómputo como RaspBerry, Intel Edison, entre otras, y por lo tanto son muy usados para proyectos de internet de las cosas de gran envergadura.

Un ejemplo de posible uso de este algoritmo es en los contadores de consumo eléctrico residencial con monitorización remota que están implementados en algunas ciudades de España y Jaṕón. Estos dispositivos podrían usar un algoritmo como AODV para comunicarse entre ellos y enviar la información de consumo a la central.

6. GLOSARIO

Ruta activa: Una ruta hacia un destino que está marcada como válida en la tabla de enrutamiento. Sólamente las rutas activas se pueden usar para enviar paquetes.

Broadcast: Transmisión de paquetes a todos los nodos en una red. AODV envía paquetes de solicitud de ruta (RREQ) mediante broadcast para obtener tantas respuestas como sea posible.

Destino: Lo mismo que “Nodo destino”. Una dirección IP a la que serán transmitidos los datos. Un nodo sabe que es el destino si su dirección IP aparece en cierto lugar del encabezado del paquete.

Nodo intermediario: Un nodo que acepta recibir y reenviar paquetes destinados a otro nodo, retransmitiéndolos al próximo salto en la ruta hacia el destino.

Ruta no válida: Una ruta hacia un destino que está marcada como no válida en la tabla de enrutamiento. Las rutas no válidas no pueden ser usadas para enviar paquetes de datos. Las rutas no válidas son mantenidas durante un largo tiempo para propósitos de información, para responder a solicitudes de rutas, y para verificar su posible reparación futura.

Nodo origen: Un nodo que envía un mensaje de descubrimiento de ruta, es el nodo que origina un mensaje RREQ.

Ruta inversa: Una ruta establecida para reenviar un paquete de respuesta (RREP) al nodo origen, o desde un nodo intermediario teniendo una ruta de destino.

Número de secuencia: Un número asociado a una ruta que se incrementa con cada cambio de estado de la ruta. Es usado por otros nodos para determinar si la información sobre la ruta ha sido actualizada o no.

Ruta válida: Ruta activa.

7. APLICACIONES

El protocolo de enrutamiento AODV está diseñado para operar en redes móviles sin infraestructura (MANET) con decenas a miles de nodos. AODV está diseñado para usarse en redes donde los nodos pueden confiar entre ellos, ya sea por uso de claves preconfiguradas, o porque se tiene la certeza de que no habrá nodos intrusos maliciosos. AODV reduce la diseminación de control de tráfico y mejora la escalabilidad y el rendimiento.

8. REFERENCIAS

[1] Ad hoc On-Demand Distance Vector (AODV) Routing
https://tools.ietf.org/html/rfc3561

[2] The Go Programming Language - Documentation
https://golang.org/doc/