Universidad del Valle de Guatemala

Facultad de ingeniería

Redes

Catedrático: Jorge Yass

Laboratorio #2 - Algoritmos de Enrutamiento

Nelson Eduardo García Bravatti 22434

Joaquín André Puente 22296

Flavio André Galán Donis 22386

Guatemala, 22 agosto de 2025

Descripción de la Práctica

El Laboratorio 3 consiste en la implementación y análisis de algoritmos de enrutamiento utilizados en redes de comunicación para comprender el funcionamiento de las tablas de enrutamiento dinámicas. La práctica busca simular cómo los nodos de una red construyen sus tablas de

enrutamiento conociendo únicamente a sus vecinos directos, y cómo estas tablas se adaptan a cambios en la infraestructura de red.

El laboratorio se estructura en dos fases principales. La primera fase se enfoca en la implementación de los algoritmos utilizando sockets TCP en el entorno local, permitiendo el desarrollo y prueba de la lógica específica de cada algoritmo. Los algoritmos requeridos incluyen Dijkstra, Flooding, Link State Routing y Distance Vector Routing

La segunda fase escala la implementación hacia un entorno distribuido utilizando un servidor XMPP o similar como medio de comunicación, donde cada nodo corresponde a un usuario en el servidor. Se establece un protocolo de comunicación estandarizado basado en mensajes JSON que incluye campos como protocolo utilizado, tipo de mensaje, origen, destino, TTL y payload.

Algoritmos Implementados

1. Dijkstra

Dijkstra es un algoritmo link-state que calcula el menor costo desde un nodo fuente u hacia todos los demás nodos de una red con pesos no negativos.
Mantiene tres estructuras:

N′: conjunto de nodos cuyo costo mínimo desde u ya es definitivo.

D(v): costo conocido más bajo de u a v en la iteración actual.

p(v): predecesor inmediato de v en el camino más corto actual.

Inicialización: N′ = {u}; para cada vecino v de u, D(v)=c(u,v) y p(v)=u; el resto D(v)=∞.
Iteración: mientras falten nodos por fijar, elegir el nodo w fuera de N′ con D(w) mínimo, añadirlo a N′ y relajar a sus vecinos: D(v) = min(D(v), D(w) + c(w,v)) (actualizando p(v) cuando mejora). Al terminar, los p(v) permiten reconstruir rutas y la tabla de reenvío (el next-hop es el primer salto desde u hacia v).

Arquitectura general (LSR + Dijkstra)

Cada Node (link_state_routing/node.go) mantiene:

neighbors con costos directos.

routingTable (next-hops).

topologyDB con los LSA recibidos (link-state de todos).

packetQueue (canal) y un ticker para generar LSAs periódicos.

Los LSA se generan con GenerateLSA() y se inundan con FloodLSA(); cada nodo que recibe un LSA nuevo (processLSA) lo re-propaga (excepto al emisor) y recalcula su tabla.

Cálculo de caminos mínimos

RunDijkstra() (link_state_routing/dijkstra.go) implementa el algoritmo:

Crea distances y previous, inicializa todos a MaxInt32 y la fuente a 0.

Repite: elige el no visitado con distancia mínima, lo marca visitado y relaja a cada vecino tomando los costos del TopologyDB (a través del LSA del nodo actual).

Devuelve Distances y Previous para reconstruir rutas.

calculateRoutingTable() toma ese resultado y, con findNextHop(), recorre previous hacia atrás hasta encontrar el primer salto después del nodo local; con eso llama routingTable.AddRoute(dest, nextHop, cost).

2. Flooding
3. Link state routing

Para el Algoritmo de Link State Routing (LSR) se implementaron varias abstracciones:

- Nodos

```
type Node struct {
ID           string
neighbors    map[string]int // neighbor -> cost
routingTable *RoutingTable
topologyDB   *TopologyDB
sequenceNum  int
packetQueue  chan *Packet
lsaInterval  time.Duration
simulation   *Simulation
}
```

Esta “clase” se encarga de guardar toda la información del nodo como sus vecinos, la tabla de ruteo que lleva calculada hasta ahora y la cola de paquetes que falta por procesar.

Un paquete de la red se ve de la siguiente forma:

```
type Packet struct {
Type        PacketType
Source      string
Destination string
Data        any
TTL         int
```

```
SequenceNum int
}
```

El tipo del campo data es cualquier cosa debido a que puede ser una string con data o una tabla de LSA. La tabla de LSA se ve de la siguiente forma:

```
type LSAData struct {
OriginRouter string
Neighbors    map[string]int // neighbor -> cost
SequenceNum  int
Timestamp    time.Time
}
```

Por último una simulación está compuesta de nodos y links entre nodos:

```
type Simulation struct {
nodes map[string]*Node
links map[string]*Link
}
```

Un link representa la conexión entre dos nodos:

```
type Link struct {
NodeA string
NodeB string
Cost  int
Up    bool
}
```

Al construir una simulación, todos los nodos se conectan de la siguiente manera:

```
nodes := []string{"A", "B", "C", "D", "E"}
for _, nodeID := range nodes {
sim.AddNode(nodeID)
}

// Add links to create a network topology
sim.AddLink("A", "B", 1)
sim.AddLink("A", "C", 4)
sim.AddLink("B", "C", 2)
sim.AddLink("B", "D", 5)
sim.AddLink("C", "D", 1)
sim.AddLink("C", "E", 3)
sim.AddLink("D", "E", 2)
```

Lo que nos permite después desconectar dos nodos entre sí:
sim.RemoveLink("C", "D")

// Trigger LSA updates
for _, node := range sim.nodes {
lsa := node.GenerateLSA()
node.topologyDB.UpdateLSA(lsa)
node.FloodLSA(lsa)
}

En la sección de resultados se puede apreciar el output de correr este programa.

Resultados

A nivel funcional y de desempeño, los tres algoritmos cumplieron su objetivo dentro de la simulación y se complementaron bien. A continuación el resumen por algoritmo y su "rol" dentro del sistema:

Flooding

Función: difusión ciega de paquetes. En LSR se usa para propagar LSAs; también sirve como mecanismo de prueba/descubrimiento.

Resultado: los mensajes alcanzaron a todos los nodos conectados respetando el TTL y sin bucles, gracias al filtro de duplicados (SeenPackets).

Desempeño: latencia baja en topologías pequeñas, pero sobre-carga de tráfico proporcional al número de enlaces; no escala para el plano de datos, sí para bootstrap/difusión ocasional (p.| ej., LSAs).

Dijkstra (SPF)

Función: cálculo centralizado del árbol de menor costo desde un origen hacia todos los destinos.

Resultado: para la topología de prueba, los costos y caminos obtenidos coincidieron con los valores óptimos esperados; la tabla de reenvío se construyó a partir de los predecesores (previous) sin inconsistencias.

Desempeño: ejecución rápida en nuestro tamaño de red (implementación $O(V^2)$, suficiente para el laboratorio). Se observaron rutas estables y deterministas; ante empates de costo, el comportamiento fue consistente.

Link-State Routing (LSR)

Función: protocolo distribuido que combina flooding de LSAs + Dijkstra local para que cada nodo mantenga una visión coherente de la topología y su tabla de ruteo.

Resultado: tras el arranque, los nodos convergieron a tablas idénticas (misma métrica/next-hop por destino).

Al simular una falla de enlace, los nodos generaron LSAs con número de secuencia mayor; cada uno re-ejecutó Dijkstra y eligió rutas alternativas (se mantuvo la conectividad cuando existía camino).

Al restaurar el enlace, la red volvió a las rutas mínimas originales. No se observaron bucles ni "flapping" en el plano de datos.

Desempeño: la convergencia ocurrió en el orden de segundos (acorde al intervalo de LSAs y a la difusión). El overhead quedó limitado a LSAs periódicos y a LSAs por evento; el forwarding de datos se mantuvo eficiente (un solo next-hop por destino).

Discusión

Definitivamente el algoritmo más complicado de implementar fue el LSR, ya que necesita conocer más estados y además actualizarse en vivo según el estatus de sus vecinos. Su principal desventaja es su complejidad ya que en tiempos de convergencia es uno de los mejores.

El algoritmo de Flooding es uno muy sencillo que realmente su única complejidad es tener que reducir el TTL del paquete para asegurarnos que, aunque se tenga un paquete en loop no llene la red por tiempo indefinido.

Por último, el algoritmo de DIJKSTRA por sí solo tiene la gran desventaja que no se puede adaptar a topologías dinámicas en donde nodos entran y salen, al menos no lo suficientemente rápido. Es justamente en respuesta a esta carencia que se crea LSR aunque sea más complicado.

Conclusiones

- **El algoritmo de Link State Routing (LSR) demostró ser el más robusto y eficiente en redes dinámicas**, ya que permitió la convergencia de todos los nodos hacia tablas de enrutamiento coherentes y adaptables. Ante fallas o restauraciones de enlaces, el protocolo ajustó las rutas rápidamente mediante la difusión de LSAs y la reejecución de Dijkstra, manteniendo la conectividad sin bucles ni inconsistencias.
- **Flooding cumplió un rol fundamental como mecanismo de difusión inicial y de propagación de LSAs**, garantizando que la información de topología llegara a todos los nodos de la red. Aunque no es escalable como protocolo de enrutamiento de datos por su

sobrecarga de tráfico, resultó útil para descubrimiento y distribución ocasional de información de estado.

- **Dijkstra confirmó su eficacia como algoritmo base para el cálculo de rutas óptimas**, generando tablas de enrutamiento deterministas y sin errores en topologías estáticas. Sin embargo, su limitación para adaptarse por sí solo a cambios en la infraestructura evidencia la necesidad de protocolos distribuidos como LSR, que integran su capacidad de optimización con mecanismos de actualización dinámica.