



Excelencia que trasciende

DELVALLE
GRUPO EDUCATIVO

Laboratorio 3

Redes

Hugo Rivas - 22500
Mauricio Lemus - 22461
Alexis Mesias -22562

Algoritmo Utilizado: Dijkstra comunicación vía sockets.

Descripción:

Componentes Core

- **Graph:** Representa topología como diccionario de adyacencias
- **DijkstraRouter:** Implementa algoritmo usando heap, calcula distancias mínimas y genera tabla de forwarding
- **DijkstraResult:** Encapsula resultados y determina next-hops

Algoritmo

1. Inicializar distancias (origen=0, resto=infinito)
2. Priority queue con nodos ordenados por distancia
3. Relajación de aristas: actualizar distancias si se encuentra ruta mejor
4. Construir tabla de ruteo con next-hop para cada destino

Versión con Sockets

- **Multihilo:** servidor TCP + manejadores de mensajes + interfaz usuario
- **Protocolo:** mensajes HELLO (descubrimiento), INFO (tablas), MESSAGE (datos)
- **Funcionalidad:** descubrimiento automático de vecinos y forwarding de mensajes

Salida

Genera tablas compatibles con protocolo del laboratorio en formato JSON y legible para debugging. Modular para uso en algoritmos más complejos.

Resultados:

```
PS C:\Users\Eduar\OneDrive\Documentos\REDES\Lab3_Redex> python dijkstra_sockets.py --node-id A --port 8000 --topo ejemplos/topo-simple.txt
Iniciando nodo A en puerto 8000
Tabla de ruteo calculada: 4 entradas
Servidor iniciado en localhost:8000
Mensaje recibido de B: hello
Vecino descubierto: B en puerto 8001
Tabla enviada a B
Nodo A iniciado correctamente

Tabla de ruteo para A:
-----
Destino  NextHop  Costo
-----
A        A        0
B        B        1
C        C        1
D        B        2
-----

Comandos disponibles:
  send <destino> <mensaje> - Enviar mensaje a otro nodo
  table                    - Mostrar tabla de ruteo
  quit                     - Salir

[A]> Mensaje recibido de C: hello
Vecino descubierto: C en puerto 8002
Mensaje recibido de D: hello
Vecino descubierto: D en puerto 8003
send C holaNodoC
Mensaje forwarded a C via C

[A]> []
```

```

PS C:\Users\Eduar\OneDrive\Documentos\REDES\Lab3_Redes> python dijkstra_sockets.py --node-id B --port 8001 --topo ejemplos/topo-simple.txt
Iniciando nodo B en puerto 8001
Tabla de ruteo calculada: 4 entradas
Servidor iniciado en localhost:8001
Mensaje recibido de A: info
Tabla de ruteo recibida de A: 4 entradas
Mensaje recibido de C: hello
Vecino descubierto: C en puerto 8002
Mensaje recibido de D: hello
Vecino descubierto: D en puerto 8003
Tabla enviada a C
Tabla enviada a D
Nodo B iniciado correctamente

Tabla de ruteo para B:
-----
Destino  NextHop  Costo
-----
A        A        1
B        B        0
C        A        2
D        D        1
-----

Comandos disponibles:
  send <destino> <mensaje> - Enviar mensaje a otro nodo
  table                    - Mostrar tabla de ruteo
  quit                     - Salir

[B]> send D HolaMundo
Mensaje forwarded a D via D

[B]> 

```

```

PS C:\Users\Eduar\OneDrive\Documentos\REDES\Lab3_Redes> python dijkstra_sockets.py --node-id C --port 8002 --topo ejemplos/topo-simple.txt
Iniciando nodo C en puerto 8002
Tabla de ruteo calculada: 4 entradas
Servidor iniciado en localhost:8002
Mensaje recibido de D: hello
Vecino descubierto: D en puerto 8003
Mensaje recibido de B: info
Tabla de ruteo recibida de B: 4 entradas
Tabla enviada a D
Nodo C iniciado correctamente

Tabla de ruteo para C:
-----
Destino  NextHop  Costo
-----
A        A        1
B        A        2
C        C        0
D        D        1
-----

Comandos disponibles:
  send <destino> <mensaje> - Enviar mensaje a otro nodo
  table                    - Mostrar tabla de ruteo
  quit                     - Salir

[C]> Mensaje recibido de A: message
MENSAJE PARA MI: holaNodoC

```

```

PS C:\Users\Eduar\OneDrive\Documentos\REDES\Lab3_Redes> python dijkstra_sockets.py --node-id D --port 8003 --topo ejemplos/topo-simple.txt
Iniciando nodo D en puerto 8003
Tabla de ruteo calculada: 4 entradas
Servidor iniciado en localhost:8003
Mensaje recibido de B: info
Tabla de ruteo recibida de B: 4 entradas
Nodo D iniciado correctamente

Tabla de ruteo para D:
-----
Destino  NextHop  Costo
-----
A        B        2
B        B        1
C        C        1
D        D        0
-----

Comandos disponibles:
  send <destino> <mensaje> - Enviar mensaje a otro nodo
  table                    - Mostrar tabla de ruteo
  quit                     - Salir

[D]> Mensaje recibido de C: info
Tabla de ruteo recibida de C: 4 entradas
Mensaje recibido de B: message
MENSAJE PARA MI: HolaMundo

```

```

PS C:\Users\Eduar\OneDrive\Documentos\REDES\Lab3_Redes> python dijkstra_sockets.py --node-id D --port 8003 --topo ejemplos/topo-simple.txt
Iniciando nodo D en puerto 8003
Tabla de ruteo calculada: 4 entradas
Servidor iniciado en localhost:8003
Mensaje recibido de B: info
Tabla de ruteo recibida de B: 4 entradas
Nodo D iniciado correctamente

Tabla de ruteo para D:
-----
Destino  NextHop  Costo
-----
A        B        2
B        B        1
C        C        1
D        D        0
-----

Comandos disponibles:
send <destino> <mensaje> - Enviar mensaje a otro nodo
table                    - Mostrar tabla de ruteo
quit                     - Salir

[D]> Mensaje recibido de C: info
Tabla de ruteo recibida de C: 4 entradas
Mensaje recibido de B: message
MENSAJE PARA MI: HolaMundo
send A Hola como estas
No se encontró ruta para A

[D]> 

```

Algoritmo Utilizado: Link State Routing comunicación vía sockets

Descripción:

- El algoritmo implementa el enrutamiento mediante difusión de estados de enlace y cálculo de rutas óptimas con Dijkstra. Cada nodo descubre a sus vecinos, genera un Link State Packet (LSP) con sus costos y lo propaga a toda la red por flooding. Con la información recibida de todos los nodos, reconstruye la topología global y calcula las rutas más cortas.

Componentes Core:

- **Graph:** representa la topología a partir de todos los LSP recolectados.
- **LSDB:** base de datos que almacena los estados de enlace recibidos (seqno, expiración, costos).
- **Dijkstra:** se ejecuta sobre la LSDB para generar la tabla de ruteo.
- **Forwarding:** reenvía paquetes DATA según el next-hop calculado.

Algoritmo:

- Enviar HELLO a vecinos para descubrir y medir costos.
- Construir y floodear LSP con enlaces y costos propios.
- Al recibir un LSP nuevo, almacenarlo en la LSDB y re-floodearlo.
- Ejecutar Dijkstra sobre la topología global para actualizar tabla de ruteo.
- Usar tabla para reenviar mensajes por la ruta más corta.

Versión con Sockets:

- **Multihilo:** un proceso gestiona Forwarding (entrada/salida de paquetes) y otro Routing (tablas y Dijkstra).
- Protocolo en JSON con mensajes HELLO, INFO/LSP y MESSAGE.

- Flooding confiable con control de duplicados para difundir LSPs.

Salida:

- Tablas de ruteo en JSON legibles (`{ 'C' : 'A' , 'B' : 'A' }` etc.).
- Entrega de mensajes extremos confirmada en destino.

Resultados:

- Los nodos calculan rutas óptimas a todos los destinos a partir de la LSDB.
- Se verificó entrega de mensajes en topología estrella (ej. B→C vía A).
- Al caer un nodo, la ruta se elimina tras expirar su LSP; al reincorporarse se recalcula automáticamente.

```

Windows PowerShell
Nodo B levantado en modo lsr. Vecinos: ['A']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
[DATA] A -> B: Hola B te saluda A

VS Code: run_node.py
PS C:\Users\Alexis Mesias\OneDrive\UNGB\O\REDES\Lab3> python run_node.py --node A --mode lsr --config-topo config/topo-exampl
e.txt --config-names config/names-example.txt --listen 127.0.0.1:5001 --peers 127.0.0.1:5002,127.0.0.1:5003
Nodo A levantado en modo lsr. Vecinos: ['B', 'C']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
send B Hola B te saluda A

```

```

Windows PowerShell
Nodo B levantado en modo lsr. Vecinos: ['A']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
[DATA] A -> B: Hola B te saluda A
send C Hola C te saluda B

Windows PowerShell
Nodo C levantado en modo lsr. Vecinos: ['A']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
[DATA] B -> C: Hola C te saluda B

```

```

VS Code: run_node.py
PS C:\Users\Alexis Mesias\OneDrive\UNGB\O\REDES\Lab3> python run_node.py --node A --mode lsr --config-topo config/topo-exampl
e.txt --config-names config/names-example.txt --listen 127.0.0.1:5001 --peers 127.0.0.1:5002,127.0.0.1:5003
Nodo A levantado en modo lsr. Vecinos: ['B', 'C']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
send B Hola B te saluda A
[DATA] C -> A: Hola A te saluda C

Windows PowerShell
Nodo C levantado en modo lsr. Vecinos: ['A']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
[DATA] B -> C: Hola C te saluda B
send A Hola A te saluda C
[DATA] A -> C: Hola C te saluda A

```

```

VS Code: run_node.py
PS C:\Users\Alexis Mesias\OneDrive\UNGB\O\REDES\Lab3> python run_node.py --node A --mode lsr --config-topo config/topo-exampl
e.txt --config-names config/names-example.txt --listen 127.0.0.1:5001 --peers 127.0.0.1:5002,127.0.0.1:5003
Nodo A levantado en modo lsr. Vecinos: ['B', 'C']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
send B Hola B te saluda A
[DATA] C -> A: Hola A te saluda C
send C Hola C te saluda A

Windows PowerShell
Nodo C levantado en modo lsr. Vecinos: ['A']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
[DATA] B -> C: Hola C te saluda B
send A Hola A te saluda C
[DATA] A -> C: Hola C te saluda A

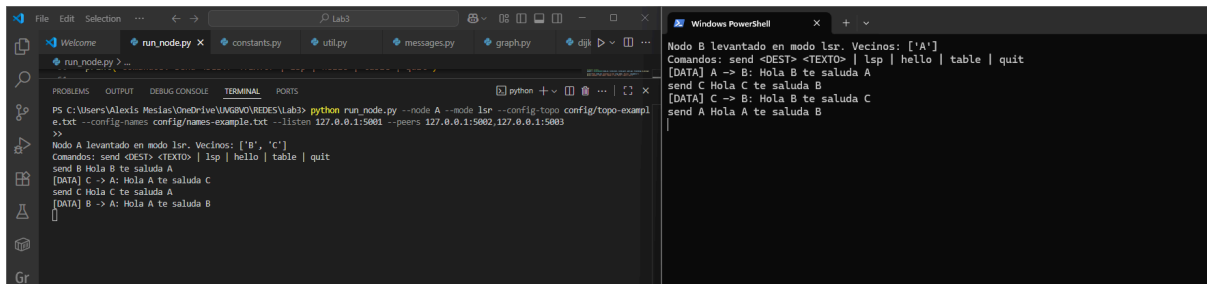
```

```

Windows PowerShell
Nodo B levantado en modo lsr. Vecinos: ['A']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
[DATA] A -> B: Hola B te saluda A
send C Hola C te saluda B
[DATA] C -> B: Hola B te saluda C

Windows PowerShell
Nodo C levantado en modo lsr. Vecinos: ['A']
Comandos: send <DEST> <TEXT0> | lsp | hello | table | quit
[DATA] B -> C: Hola C te saluda B
send A Hola A te saluda C
[DATA] A -> C: Hola C te saluda A
Send B Hola B te saluda C
Comando desconocido
send B Hola B te saluda C

```



The image shows a code editor with several files open: `run_node.py`, `constants.py`, `util.py`, `messages.py`, `graph.py`, and `dijkstra.py`. The `run_node.py` file is active, showing a Python script for a network node. The terminal output shows the execution of the script, displaying the node's neighbors and the messages it receives and sends. The PowerShell terminal on the right shows the execution of the `python run_node.py` command, displaying the node's neighbors and the messages it receives and sends.

Algoritmo Utilizado: Flooding con TTL, comunicación vía sockets UDP

Descripción:

Cada nodo reenvía cualquier mensaje que recibe a todos sus vecinos, disminuyendo un TTL hasta que llega a 0. El destino acepta el mensaje cuando el campo `to` coincide con su `node_id`. Para evitar tormentas de duplicados, cada nodo guarda una huella del mensaje ya visto. No calcula rutas: confía en la difusión ciega acotada por TTL.

Componentes Core:

- **Graph:** grafo no dirigido construido desde `topo.json`.
- **Endpoints:** mapa `{node: (host, port)}` desde `endpoints.json` para enviar/recibir UDP.
- **Envelope:** mensaje JSON con campos

```
{"proto": "flooding", "type": "message|echo", "from", "to", "ttl", "headers": [], "payload": ...}.
```

- **Duplicate filter :** set con una clave ligera por mensaje para evitar re-procesarlo.
- **Forwarder:** reenvía a todos los vecinos salvo `came_from`, decrementando `ttl`.
- **Listener:** hilo que recibe datagramas y despacha a `_handle`.

Algoritmo:

1. **Inicialización:** cada proceso liga su socket al `(host, port)` del nodo y arranca un hilo receptor.
2. **Recepción:**
 - Si la clave del mensaje ya está en `seen`, se descarta.
 - Si `type=="echo"` o `to==node_id`, se imprime un evento `recv` (y el `echo` no se reenvía).
 - Se decrementa `ttl`; si `ttl<=0`, se descarta.
 - Se re-floodea a todos los vecinos \neq `came_from`.
3. **Envío de datos :** el emisor crea el sobre y lo inunda a sus vecinos.
4. **Ping :** envía `echo` a todos los vecinos directos para verificar conectividad.

Versión con Sockets:

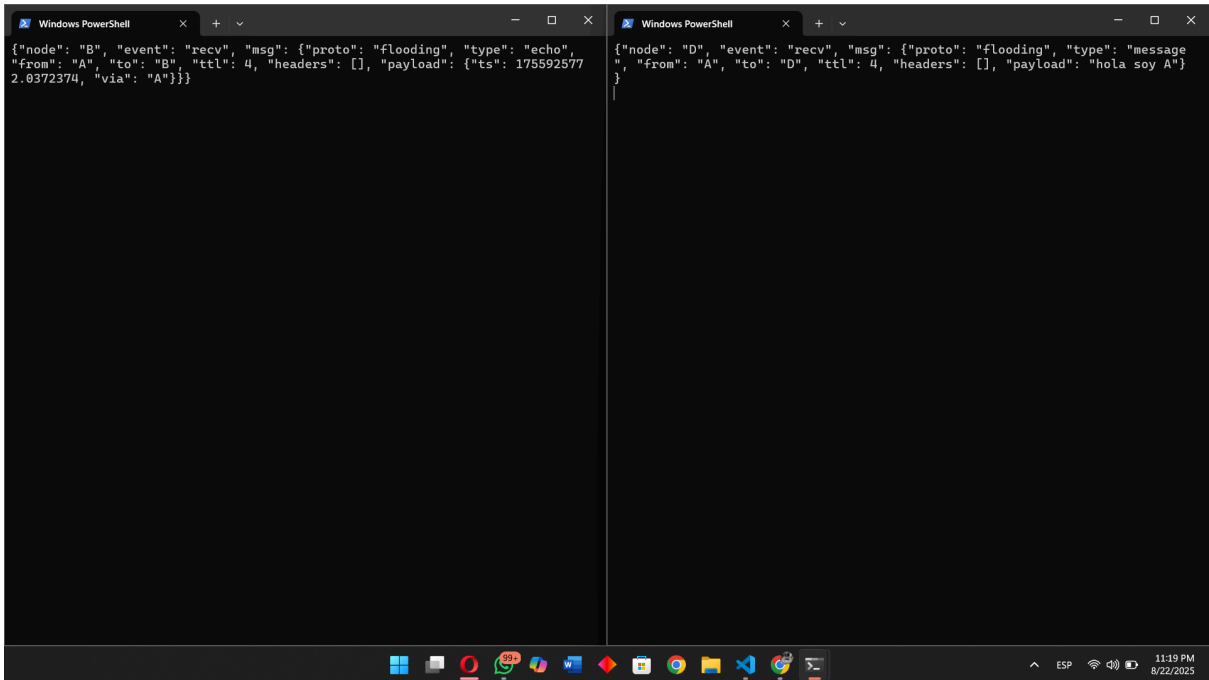
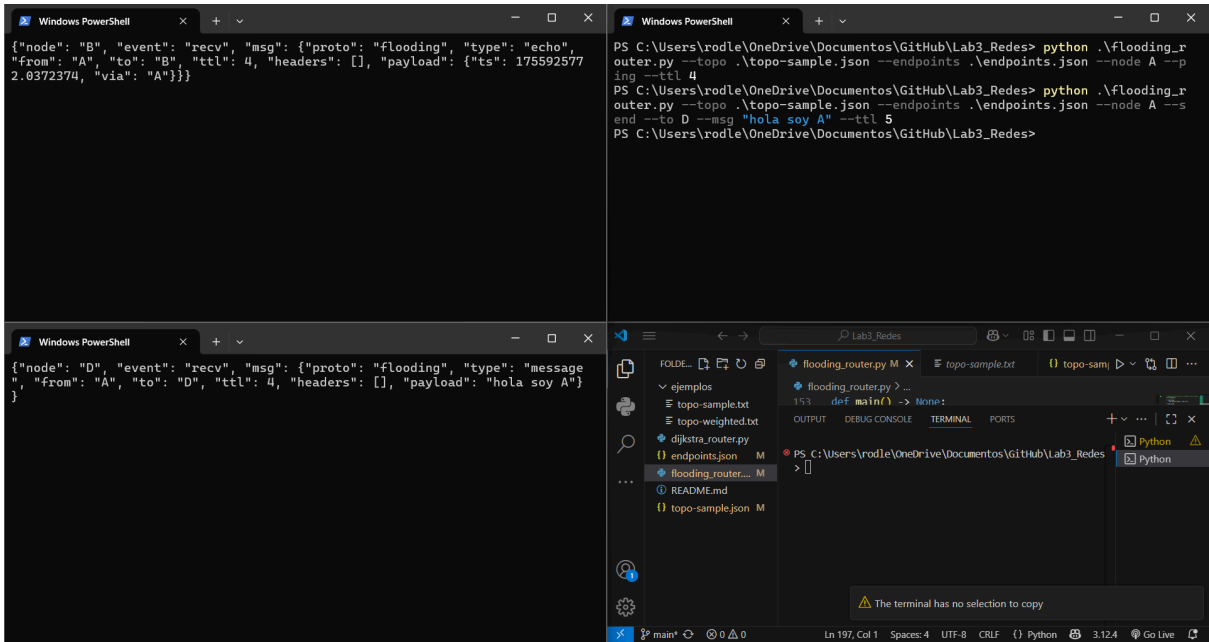
- **UDP puro en localhost.**
- **Multihilo:** un hilo receptor por proceso , el hilo principal permanece vivo.
- Proceso inyector sin `bind**:**` para `--send/--ping` se ejecuta un proceso que no hace `bind`.
- **Control de duplicados:** clave basada en `{from, to, payload}`; limita reenvíos redundantes, aunque no garantiza unicidad perfecta.
- **Fiabilidad:** no garantiza entrega ni orden; el control de la difusión es por TTL y filtro de duplicados.

Salida:

- **Eventos en JSON en cada nodo receptor:**

```
{"node": "D", "event": "recv", "msg": {"proto": "flooding", "type": "message", "from": "A", "to": "D", "ttl": 2, "headers": [], "payload": "hola"}}
```

- Para `--ping`, cada vecino imprime su `recv` con el payload `{"ts": ..., "via": "<origen>"}`.



Discusión:

Los algoritmos que se estuvieron probando fueron Dijkstra, Link State y Flooding. Cada uno tiene un estilo muy diferente de trabajar y por lo mismo nos deja aprendizajes complementarios. El flooding fue el más sencillo de implementar, pero también el menos “inteligente”, ya que no busca la ruta más corta ni optimiza el tráfico, solo se dedica a inundar mensajes hasta que lleguen al destino. Eso lo hace útil para propagar información rápidamente en redes pequeñas o desconocidas, pero poco escalable en redes grandes. Comparado con Dijkstra y Link State, se nota la diferencia: estos dos intentan calcular caminos eficientes, mientras que el flooding es más una garantía de alcance a costa de consumir ancho de banda. Al final, verlo funcionando junto a los otros algoritmos nos ayuda a valorar cuándo usar un método simple y cuándo uno más sofisticado.

Conclusiones:

- El flooding con TTL es una estrategia fácil de implementar y asegura la entrega de mensajes mientras el TTL sea suficiente, aunque genera sobrecarga de tráfico.
- A diferencia de Dijkstra y Link State, no construye tablas de ruteo ni busca rutas óptimas, lo que lo hace poco eficiente en redes grandes.
- Es un buen punto de partida para comprender cómo se difunde información en la red y sirve como base para algoritmos más avanzados que sí optimizan el camino.