Lab 3 — Algoritmos de Enrutamiento

Redes de Computadoras
Rodrigo Mansilla
**Nelson Escalante

Resumen

Se implementaron y probaron cuatro modos de encaminamiento sobre un router local multihilo: **Dijkstra** (estático), **Flooding**, **LSR** (link-state con LSPs y flooding confiable) y **DVR** (distance-vector con Split Horizon + Poisoned Reverse).

Se añadieron **paquetes HELLO/ECHO** para medir RTT con vecinos, **PING/PONG** para RTT extremo a extremo, **DATA** para mensajes de usuario, e **INFO** (LSP/DV) para intercambio de estado de red.
Se automatizó la ejecución con `run.py` que levanta 4 nodos sin abrir consolas extra, ofrece menú de pruebas y genera logs por nodo. Los experimentos confirman entrega de datos y convergencia de rutas en los cuatro modos, además de recuperación ante reinicio de nodos.

1. Descripción de la práctica (qué se pidió y qué se hizo)

Objetivo. Implementar y comparar algoritmos de enrutamiento (estático y dinámico), diseñar paquetes de control, y **automatizar** pruebas con captura de métricas.

Requerimientos clave cumplidos

- **No "hacer trampa" con config**: cada nodo solo usa `names/topo` para su propia IP/puerto y lista de vecinos; el descubrimiento de costos se hace vía **HELLO/ECHO (RTT)** excepto en Dijkstra puro (topología fija).
- **Timers y expiración**: se incorporó `last_seen` y expiro de DV/LSP; umbrales ajustables por CLI (`--hello-period` , `--dead-after`).
- **Paquetes**:
  - `HELLO/ECHO` : medición RTT con vecinos.
  - `DATA` : mensajes de usuario.
  - `PING/PONG` (como `DATA`): RTT extremo a extremo.
  - `INFO` : LSP (LSR) / Vector de distancias (DVR).
- **Automatización**: `run.py` levanta 4 nodos en *background*, menú de pruebas (DATA, PING, INFO, reinicios), y **logs por nodo** (`A_modo.log` , etc.).
- **Preparación a Fase 2 (XMPP)**: diseño de `send_cli.py` y `run.py` desacoplados del transporte; solo reemplazar transporte TCP local por XMPP manteniendo los mismos "tipos" de paquetes.

Entorno

- Python 3.13 en Windows, sockets TCP locales (127.0.0.1:5001–5004).
- Archivos: `config/nodes.json` (names), `config/topo.json` (pesos base).
- Ejecución: `python run.py` → seleccionar algoritmo y prueba; `run.py` gestiona PIDs y logs

2. Arquitectura de la solución

- **RouterNode** (por proceso):
  - **Hilos**: `forwarding_loop` (TCP server), `routing_loop` , `hello_loop` (HELLO/ECHO periódicos).
  - **Tabla de ruteo**: `{dest: {next_hop, cost}}` , protegida por lock.

- **Métricas de vecino**: `rtt_ms`, `last_seen`.
- **Paquetes**: `HELLO/ECHO/INFO/DATA` (+ `PING/PONG` sobre `DATA`).

- **Módulos**:
  - `flooding.py`: deduplicación por `mid` y `prev`, TTL, evita vecinos “caídos”.
  - `lsr.py`: LSDB con `seq`, `age/ts`, expiración, flooding confiable, reconstrucción de grafo y Dijkstra.
  - `dvr.py`: Bellman-Ford distribuido, Split Horizon + Poisoned Reverse, expiración de DV, anuncios periódicos y “triggered updates”.
  - `dijkstra.py`: Dijkstra clásico + tabla de next-hop.
  - **Mensajería**: `messages.py` unifica wire-format JSON.
- **Automatización**:
  - `run.py`: lanza nodos (sin abrir consolas), ofrece **menú** (DATA, PING, INFO, reinicios), y genera/lee **logs**.
  - `send_cli.py`: cliente para inyectar mensajes de prueba (reutilizado por `run.py`).

3. Descripción de los Algoritmos Utilizados y su Implementación

> En todos los modos, cada **RouterNode** corre 3 hilos:
> `forwarding_loop` (servidor TCP + parseo de mensajes),
> `routing_loop` (cómputo/advertise/expiración),
> `hello_loop` (HELLO/ECHO para medir RTT y *liveness*).
> Los paquetes son JSON con campos `{proto, type, from, to, ttl, headers, payload}`

3.1 Dijkstra (ruteo estático)

Computar las rutas de costo mínimo desde el nodo local hacia todos los destinos en un **grafo ponderado** conocido usando el algoritmo de Dijkstra.

Estructuras clave.

- `topology: Dict[str, Dict[str, float]]` (grafo dirigido/no dirigido con pesos).
- `PathResult = {dist, prev, next_hop}`.
- `routing_table[dst] = {next_hop, cost}`.

Implementación (archivo `dijkstra.py`).

- `dijkstra(topology, source)`: usa un **heap** para relajar aristas; guarda `dist[]` y `prev[]`.
- `_compute_next_hops(prev, source)`: sube por `prev` desde cada `dst` hasta `source` para fijar el **primer salto**.
- `build_routing_table(result, me)`: arma la tabla; para `me`, fuerza `{next_hop: me, cost: 0}`.

Complejidad $O(E \log V)$ con heap binario ($E = aristas, V = nodos$).

Integración.

- En `routing_loop`, si `mode == "dijkstra"`, se computa siempre desde el **grafo del archivo**.
- `forwarding_loop` reenvía con el `next_hop` de `routing_table`; si `to == me`, entrega al usuario.

Logs.

- `[A/FWD] FWD D vía B`,
- ECHOs solo informativos (no afectan costos en este modo).

3.2 Flooding (inundación con deduplicación)

Reenviar paquetes **DATA/INFO** a todos los vecinos , con **TTL** y **deduplicación** para evitar bucles y explosión.

Estructuras clave (archivo `flooding.py`).

- `self.seen: set[str]` para `mid` .
- Encabezados en `headers` :
  - `mid` : `<from>:<seq>` .
  - `prev` : último salto .

Implementación.

- `handle_data(node, msg)` : genera/lee `mid` ; si ya visto, descarta; si `to == me` , entrega; si no, llama `_flood` .
- `_flood(node, msg)` : copia `msg` , decrementa `ttl` , fija `prev = node_id` , y envía a cada vecino **vivo** (ver `nei_metrics.last_seen`) excepto `prev` .
- `handle_info` hace igual que `handle_data` pero para `INFO` .

Detalles

- **Vecinos vivos**: se filtran con `last_seen` (HELLO/ECHO) y el umbral `DEAD_AFTER` .
- **TTL**: corta inundaciones largas; se fija al crear el mensaje (`run.py` / `make_msg`).

Complejidad. Por mensaje, $O(E)$ *en el peor caso*, con alta redundancia.

Logs.

- `[A] FWD(flooding) → B (dst=D, mid=A:...)`
- En el destino: `DATA para mí de A: {...}` .

3.3 LSR — Link-State Routing (LSP + Dijkstra dinámico)

Cada nodo anuncia a toda la red su **estado de enlaces**. Todos mantienen una **LSDB** y corren Dijkstra localmente sobre la topología resultante.

Paquete LSP (en `payload.lsp`).

```
{
  "origin": "C",
  "seq": 12,
  "age": 0,
  "neighbors": [{"id": "A", "cost": 4.0}, {"id": "B", "cost": 2.0}]
}
```

- `mid` del INFO: `"origin:seq"` (estable) para **deduplicación** en flooding.

Estructuras clave (archivo `lsr.py`).

- `lsdb: {origin -> {"seq": int, "ts": float, "neighbors": {nbr: cost}}}`
- `last_local` : *snapshot* local de costos a vecinos vivos.
- Flags y timers: `seq` , `last_adv[me]` , `changed` .

Ciclo de control.

1. **Medición de costos locales**: `hello_loop` actualiza `nei_metrics.rtt_ms` .

2. **should_advertise(node)**: si pasaron `min_interval` o hubo cambios `> threshold` o 10s para *refresh*.
3. **advertise(node)**: construye LSP con `seq++` y difunde por flooding .
4. **on_receive_info**: si el LSP es **nuevo** , **actualiza LSDB** y re-difunde.
5. **expire(max_age)**: borra LSPs antiguos .
6. **build_topology()**: grafo simétrico tomando el **menor** costo cuando hay dos reportes divergentes (A→B y B→A).
7. `routing_loop` : si `changed` , corre **Dijkstra** sobre `build_topology()` y publica nueva `routing_table` .

Complejidad.

- Anuncio: flooding $O(E)$.
- Recomposición + Dijkstra cuando cambia LSDB: $O(E \log V)$

Logs.

- `[C/LSR] ADV LSP seq=…` (si se habilita),
- recomputes reflejados como nuevos `FWD … vía …` en nodos.

3.4 DVR — Distance-Vector Routing (Bellman-Ford distribuido)

Cada nodo mantiene un **vector de distancias** (`dv_self[dst]`) y lo intercambia con sus vecinos. El costo hacia un destino es el **mínimo** entre el enlace al vecino + su costo al destino.
Se añaden **Split Horizon + Poisoned Reverse** para reducir *count-to-infinity*, **triggered updates** y manejo explícito de `INF` .

Estructuras clave (archivo `dvr.py`).

- `dv_self: {dst -> cost}` con `dv_self[me] = 0.0` .
- `next_hop: {dst -> nh}` (incluye `me->me`).
- `dv_from[n]: {dst -> cost}` y `dv_from_ts[n]` para *aging*.
- Timers/flags: `seq` , `last_adv` , `changed` .

Costos de enlace.

- `use_static_costs=True` : del `topo.json` .

Bellman-Ford local (núcleo).

Pseudocódigo resumido de `recompute(node)` :

```
alive = vecinos con last_seen ≤ dead_after
dests = {me} ∪ vecinos ∪ claves de dv_from[*]

new_dv[me] = 0; new_nh[me] = me
para cada dst != me:
  best_cost = INF, best_nh = None
  si dst ∈ alive:                 # enlace directo
    best_cost = cost(me→dst); best_nh = dst
  para cada n ∈ alive:            # vía vecinos
    via = cost(me→n) + dv_from[n].get(dst, INF)
    si via < best_cost: best_cost = via; best_nh = n
  new_dv[dst] = best_cost; new_nh[dst] = best_nh

changed = (dv_self, next_hop) difieren de (new_dv, new_nh)
dv_self = new_dv; next_hop = new_nh; self.changed = changed
```

Split Horizon + Poisoned Reverse.

- Al **anunciar a** `nei`, si `next_hop[dst] == nei` (y `dst != nei`), se anuncia `INF` para `dst`.
- Implementado en `_vector_for_neighbor(nei)`.

Triggered updates y periódico.

- `should_advertise(min_interval, refresh_every)`:
  - True si `seq==0` (primera vez),
  - o si pasó `min_interval` y (`changed` o `refresh_every`).
- **Triggered**: al recibir ECHO que **cambia costos** o DV que **modifica rutas**, `changed=True`. Desde `node._on_echo` se fuerza `self.dvr.changed = True` y `trigger_update()`.

Expiración / caída de vecino.

- `expire(dv_max_age)`: si no llega DV de un vecino o ya **no está** en `neighbors`, se borra su entrada → `recompute()`.

INF y estabilidad.

- `INF = 1e12`. Ante pérdida de vecino o DV que "envenenó" un destino, los costos se **propagan al alza** evitando rutas inválidas; SH+PR reduce *loops*.

Tabla de ruteo.

- `build_routing_table()`: vuelca `dv_self` y `next_hop` a `{dst: {next_hop, cost}}`.

Integración.

- `routing_loop`: `expire()`, `update_local_links()` → `recompute()`; si `changed`, publicar tabla y **luego** `advertise()` cuando `should_advertise()` sea True.
- `on_receive_info`: guarda `dv_from[src]`, marca timestamp y `recompute()`; el **RouterNode** publica la tabla **de inmediato** ("Tabla DVR actualizada (RX)").

Complejidad por recompute.

- Aproximadamente $O(|alive| \cdot |dests|)$ por iteración local. Con pocos nodos del laboratorio, es barato y se corre a cada evento/ciclo

3.5 Paquetes, Timers y Reglas Comunes

Tipos de paquetes (resumen).

- `HELLO` → `ECHO` (vecino-vecino): payload `{seq, ts}`; actualiza `last_seen` y `rtt_ms`.
- `DATA` (usuario): payload libre `{kind, text, …}`; TTL decrece en cada salto.
- `INFO` (control):
  - **LSR**: `payload.lsp = {origin, seq, neighbors:[{id, cost}], age}` + `headers.mid="origin:seq"`.
  - **DVR**: `payload.dv = {dst: cost, …}`.

Timers/umbrales (ajustables por CLI).

- `--hello-period` (default 5.0 s): período de HELLO a cada vecino.
- `--dead-after` (default 10.0 s): si `now − last_seen > dead_after`, el vecino se considera **caído** (filtrado en Flooding/LSR/DVR).
- **LSR**: `min_interval≈3 s`, `refresh≈10 s`, `max_age≈30 s`.
- **DVR**: `min_interval≈1–2 s`, `refresh≈8–10 s`, `dv_max_age≈20 s`.

Reenvío y control de TTL.

- Si `ttl <= 0` → descartar.
- **Flooding** añade `prev` y respeta `mid` para deduplicación.
- **Table-driven (Dijkstra/LSR/DVR)**: consulta `routing_table[to].next_hop`; si no existe, registra `Sin ruta a X`.

Descubrimiento/vecinos (sin "trampa").

- Cada nodo **solo** usa `names/topo` para: su **IP/puerto** y la **lista inicial de vecinos**.
- Costos *dinámicos* (cuando aplica) provienen de **HELLO/ECHO (RTT)**; **Dijkstra** permanece atado al **topo.json** (estático), tal como pide el enunciado.

Logging normalizado (con `--log-level`).

- Prefijos: `[A/START]`, `[A/ECHO]`, `[A/FWD]`, `[A/RECV]`, `[A/LSR]`, `[A/DVR]`.
- `run.py` redirige stdout/stderr de cada nodo a `A_modo.log`, etc., e imprime cabeceras `----- start <ISO8601> -----`.

4 .Resultados

Topología A–B–C–D del enunciado. Se ejecutaron pruebas por modo: `dijkstra`, `flooding`, `lsr` y `dvr`. En cada modo se envió un burst de 10 PINGs lógicos A→D (PONGs vistos en D y/o en A con RTT). Además se verificó el next-hop observado en los logs de cada nodo.

Métrica de entrega (A→D):

- `dijkstra`: 10/10 entregados (PONGs en A).
- `flooding`: 10/10 entregados (DATA en D).
- `lsr`: 10/10 entregados (PONGs en A).
- `dvr`: 10/10 entregados (PONGs en A).

RTT de PING A→D (ms, desde los PONG en A; n=10 por modo):

Modo	min	p50 (mediana)	p95	max	promedio
dijkstra	44.0	97.3	114.2	118.6	91.6
lsr	23.1	68.1	82.9	87.6	62.5
dvr	18.6	73.9	95.5	99.3	68.4

Rutas observadas (next-hop):

- `dijkstra`
  - A→D: A **via B** (A/FWD "via B").
  - B→D: B **via C**.
  - C→D: C **via D**.
  - D→A: D **via C** , coherente con el camino inverso D–C–B–A.
- `flooding`
  - Entrega por inundación con deduplicación: D recibe exactamente 10 DATA únicos de A. Se observa alto fan-out de reenvíos (A/B/C → múltiples vecinos) como es esperable en flooding.
- `lsr`
  - A→D: mayormente **via B**, con un ajuste puntual **via C** tras difusión de LSPs .
  - C→D: **via D** .
- `dvr`
  - A→D: arranca **via C** (coste 5) y converge a **via B** (coste 4).
  - B→D: **via C** (coste 3).

- D→A: **via C** (coste 4).
- Se observan varias “Tabla DVR actualizada (RX)” consecutivas hasta estabilizar.

5. Discusión

Dijkstra (estático).

- El plano de reenvío fue correcto y estable: las trazas muestran que A reenvía hacia D por B, B por C y C por D, exactamente el camino mínimo A–B–C–D según los pesos del archivo.
- El comportamiento es predecible y sin oscilaciones, es menos sensible a variaciones de calidad
- No ajusta rutas cuando cambian los RTT o cae temporalmente un vecino. Es el baseline para validar correctitud, con la contrapartida de no adaptarse.

LSR (estado de enlace).

- La difusión de LSPs por flooding con deduplicación propagó rápido los cambios de vecindad/RTT, y al recomputar Dijkstra sobre la LSDB reconstruida se observó coherencia en los next-hops y una convergencia ágil.
- Hubo un cambio puntual en A→D pasando a vía C cuando las métricas lo justificaron, lo que demuestra reacción al estado vivo de la red.
- En latencia entregó los mejores p50/p95 : menos retransmisiones, selección de enlaces más “frescos” y recalculo inmediato tras los anuncios.
- El overhead de control queda acotado a los LSPs , no a todos los DATA.

DVR (vector de distancias).

- Los anuncios de vecinos fueron reduciendo el costo percibido a D y el next-hop de A→D migró de C a B.
- El rendimiento quedó intermedio frente a LSR: converge bien y evita bucles transitorios gracias al envenenamiento, pero la información tarda un par de rondas en asentarse.
- Overhead es bajo y su implementación es simple; la convergencia no es tan inmediata como en LSR.

Flooding (referencia de plano de datos).

- Sirvió para validar alcance y deduplicación: cada DATA generó múltiples reenvíos, pero solo una entrega en destino.
- Es útil como mecanismo de transporte para LSR, aunque como algoritmo de encaminamiento puro tiene el mayor overhead y no es competitivo en eficiencia.

6. Conclusiones

- Se Implementarón LSR/Dijkstra (tipo OSPF), DVR (tipo RIP) y flooding. Se verificaron sus propiedades: LSR converge rápido con estado fresco; DVR es simple y ligero pero converge por oleadas; flooding solo como soporte/diagnóstico.
- Las tablas se actualizan con HELLO/ECHOy con INFO . Se observarón cambios de next-hop y costos coherentes.
- Los nodos corren con `run.py` , envían PING/DATA/INFO y registran eventos. La lógica de routing está desacoplada del transporte, así que migrar a XMPP implica solo reemplazar `_send/_recv` , manteniendo mensajes/TTL/deduplicación.
- LSR dio mejor latencia y reacción a cambios; DVR estabilizó rutas tras varias actualizaciones con bajo overhead; Dijkstra estático fue correcto y estable pero sin adaptarse a variaciones de calidad.

Referencias

- Kurose, J. & Ross, K. **Computer Networking: A Top-Down Approach**.
- RFC 1058 — **Routing Information Protocol (RIP)** (base para distance-vector).
- RFC 2328 — **OSPF Version 2** (link-state).

- Tanenbaum, A. & Wetherall, D. **Computer Networks**.