CSCI-4220 — Assignment 3: Prefix-Based Router with Longest-Prefix Match (Mini-RIP)

Overview

In this homework you will implement a **prefix-based router** that:

- runs a **Distance-Vector (DV)** routing protocol on prefixes (mini-RIP), and
- forwards packets using **Longest-Prefix Match (LPM)** over a simulated data plane.

Routers are separate processes on localhost and communicate via **UDP**:

- **Control port** (`listen_port`) for DV updates
- **Data port** (`listen_port + 1000`) for forwarding "IP-like" packets

Background:

RIP(Routing Information Protocol) is one of the oldest and simplest Distance-Vector routing protocols used in IP networks. Each router maintains a table of destinations and their distance (cost) — usually the number of hops. A direct neighbor = 1 hop, two routers away = 2 hops, etc. Periodically, routers exchange these tables with their neighbors and update their own entries using the Bellman–Ford algorithm.

Bellman–Ford algorithm

(Read Wikipedia https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)

It lets each router find the shortest-cost path to every destination, based only on information from its neighbors. Each router maintains a routing table with: `(destination network, mask, next_hop, cost)` Every neighbor also sends you its own table (a distance vector). For each destination in a neighbor's vector:

```
new_cost = link_cost_to_neighbor + neighbor_cost_to_destination
```

If this new_cost is smaller than your current cost, you update your table to use that neighbor as the new next hop.

Learning Objectives

- Represent routing entries as **(prefix, mask, next-hop, cost)**.
- Implement **Bellman–Ford** on prefix routes (like RIP).
- Perform **Longest-Prefix Match (LPM)** to forward packets.
- Use `select()` for concurrent sockets and periodic timers.

System Model

Each router loads a config file describing its identity, addresses, static connected routes, and neighbors.

- Control messages periodically advertise the sender's current prefix routes (net, mask, cost).
- On reception, routers apply **Bellman–Ford**:

  ```
  new_cost(prefix) = min( new_cost(prefix), cost_to_neighbor + neighbor_cost(prefix) )
  ```

- Forwarding uses **LPM** over the local routing table to choose the next hop.

In this lab, every "router" is simply a separate user process running on the same local machine. Each router listens on a unique UDP port number (for example, R1 = 12001, R2 = 12002, etc.), but all share the same host.

Therefore, all routing messages (Distance Vector updates and data packets) are actually exchanged locally via the loopback interface.

Example topology (three routers)

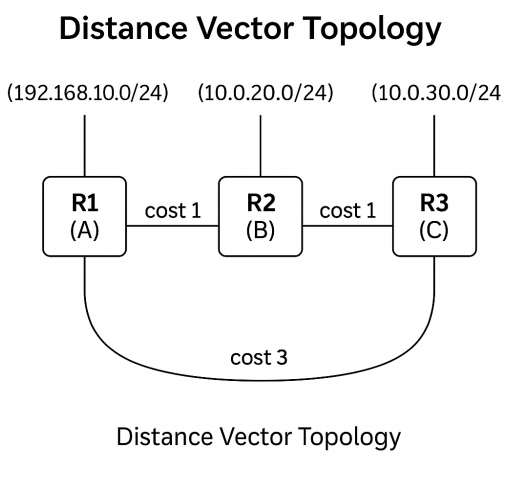

Figure 1: example

Configuration File Format

Example `r1.conf`:

```
router_id 1
self_ip 127.0.1.1
listen_port 12001

routes
  192.168.10.0 255.255.255.0 0.0.0.0 eth0

neighbors
  127.0.1.2 12002 1
  127.0.1.3 12003 3
```

- `self_ip` is the router's identity address (not used for real UDP dest; UDP still uses loopback ports).
- `routes` list **connected** prefixes. 192.168.10.0 255.255.255.0 specifies the network address and subnet mask, which together are equivalent to the CIDR notation 192.168.10.0/24 ; `next_hop=0.0.0.0` indicates directly attached. The cost from the router to this network is 0 because it is reachable without going through any other router.
- `eth0` represents the network interface through which the router is connected to that network. Because we are doing simulation and it does not correspond to a real network card, it is a symbolic placeholder for phyiscal connection. You do not need to use or process the eth0 field in your implementation — just read and store it if you like for printing or debugging.
- `neighbors` entries specify `neighbor_ip control_port link_cost`.

R2 (`r2.conf`):

```
router_id 2
self_ip 127.0.1.2
```

```
listen_port 12002

routes
  10.0.20.0 255.255.255.0 0.0.0.0 eth0

neighbors
  127.0.1.1 12001 1
  127.0.1.3 12003 1
```

R3 (`r3.conf`):

```
router_id 3
self_ip 127.0.1.3
listen_port 12003

routes
  10.0.30.0 255.255.255.0 0.0.0.0 eth0

neighbors
  127.0.1.1 12001 3
  127.0.1.2 12002 1
```

Control Plane (DV on prefixes)

Message format (binary struct):

```c
struct dv_msg {
    uint8_t  type;     // MSG_DV
    uint16_t sender_id;
    uint16_t num;
    struct {
        uint32_t net;   // network in network byte order
        uint32_t mask;  // mask in network byte order
        uint16_t cost;
    } e[MAX_DEST];
};
```

Behavior: - Every **5 seconds** send your route vector to all alive neighbors. - On receive, compute `new_cost = cost_to_neighbor + neighbor_cost` and update if better. - If any entry changes, print table and optionally trigger an immediate re-advert.

Failure detection: - If no control message from a neighbor for **15 s**, mark neighbor dead and poison routes through it (explained below).

When a router detects that a neighbor has failed (for example, it hasn't received any Distance Vector (DV) updates from that neighbor for 15 seconds), it must update its routing table to reflect that the paths going through that neighbor are no longer valid. Instead of silently deleting those routes, the router "poisons" them — that is, it keeps them in the table but marks their cost as infinity (INF_COST) and advertises that to other routers.

Data Plane (LPM Forwarding)

Data packet struct:

```c
struct data_msg {
    uint8_t  type;     // MSG_DATA
    uint8_t  ttl;
    uint32_t src_ip;   // network byte order
    uint32_t dst_ip;   // network byte order
    uint16_t payload_len;
    char     payload[128];
};
```

LPM lookup selects the route entry with the **longest mask** such that:

`(dst_ip & mask) == (dest_net & mask)`

To see if a mask is longer than another, you can compare their numeric values. (Example: 255.255.255.0 = 0xFFFFFF00 is greater than 255.255.0.0 = 0xFFFF0000 in host byte order.)

Forwarding decision: 1. If `dst_ip` belongs to `self_ip` (or any local attached prefix), **deliver** (print). 2. Else if `ttl == 0`, **drop**. 3. Else decrement TTL and **forward** to: - `next_hop` if set (non-zero); otherwise **directly** to `dst_ip` (for connected routes).

Routers send forwarded packets to the **neighbor's data port** (derived inside the router using the neighbor's control port + 1000).

Routing Stability: Split Horizon and Poison Reverse

Notes for Students

Distance Vector (DV) routing protocols can suffer from **routing loops**, where routers repeatedly advertise routes back to each other, creating oscillations and unstable costs.

To prevent this, our router needs to implement a technique called **Split Horizon with Poison Reverse**.

Split Horizon.

- **Rule:** Do **not** advertise a route back to the neighbor from which it was learned.
- **Purpose:** Prevents a router from saying "I can reach network X through you" — which would immediately form a loop.

Poison Reverse.

- **Rule:** If a route was learned from a neighbor, still advertise it back, but with an **infinite cost (INF_COST)** instead of omitting it.
- **Purpose:** Makes the route explicitly unreachable, helping neighbors invalidate it faster.

Example. Suppose R1 learns about `10.0.30.0/24` from R2:
R1 will advertise that network back to R2, but with **cost = ∞ (65535)**.
This tells R2: "Do not use me as a path to reach that network."

Code Reference.

- You need to implement this logic in send_dv() function to ensure your routers **avoid routing loops** and **stabilize quickly** after link cost changes.

Provided Starter Code

Files Included

File	Description
`common.h`	Contains shared data structures (`router_t`, `route_entry_t`, `neighbor_t`) and helper functions for managing and printing the routing table. You may read this file, but you do **not** need to modify it.
`router.c`	The main router program. It parses the configuration file, initializes the routing and neighbor tables, and includes an outline of the main loop. You will fill in the missing logic for routing updates, data forwarding, and UDP communication.
`sendpkt.c`	A small utility that simulates a host sending a packet into the network, used to test your router's forwarding behavior.
`Makefile`	Compiles all programs. Run `make` to build your router and testing tools.
Configuration files (`r1.conf`, `r2.conf`, `r3.conf`)	Define each router's ID, IP, ports, connected networks, and neighbor information. Each router reads one config file at startup.

Please read through the code and comments to understand what's already implemented and what you will implement. For sendpkt,

```
./sendpkt <data_port> <src_ip> <dst_ip> <ttl> "<payload>"
```

It generates and sends one data message (MSG_DATA) using UDP to a router's data port, with TTL and IP addresses of the simulated source and destination hosts. "" is the text string carried as the message payload.

Example run

Terminals:

```
make
```

```
./router configs/r1.conf
./router configs/r2.conf
./router configs/r3.conf
```

Inject a packet at R1 by sending a packet to R1's data port (13001) — by passing its control port (12001) to sendpkt:

```
./sendpkt 12001 192.168.10.10 10.0.30.55 8 "hello LPM world"
```

Expected logs (after convergence) from three routers:

```
[R1] FWD dst=10.0.30.55 via next_hop=127.0.1.2 mask=255.255.255.0 cost=2 ttl=7
[R2] FWD dst=10.0.30.55 via next_hop=127.0.1.3 mask=255.255.255.0 cost=1 ttl=6
[R3] DELIVER src=192.168.10.10 ttl=6 payload="hello LPM world"
```

Deliverables

- Source (`router.c`, `common.h`, `Makefile`)
- README: Log that demonstrates:
  - DV convergence

  * Example:

```
[R1] ROUTES (dv-update):
network         mask            next_hop        cost
192.168.10.0 255.255.255.0        0.0.0.0   0
10.0.20.0     255.255.255.0       127.0.1.2   1
10.0.30.0     255.255.255.0       127.0.1.2   2
```

- LPM forwarding
  * See the previous Example Run .
- Neighbor timeout behavior
  * How to demonstrate:
    1. Start all routers normally.
    2. Kill one router (e.g., pkill -f "router configs/r2.conf").
    3. Wait ~15 seconds (the DEAD_INTERVAL_SEC threshold).
    4. Check logs for:

```
[R1] ROUTES (neighbor-dead):
10.0.20.0 255.255.255.0 0.0.0.0 65535
10.0.30.0 255.255.255.0 127.0.1.2 65535
```

- To deepen your understanding of routing stability, try **disabling** the Split Horizon and Poison Reverse logic in `send_dv()` and observe what happens. What you observed after removing the Split Horizon/Poison Reverse logic?

Academic Integrity and AI Use Policy

You must implement all networking, threading, and synchronization logic yourself.
All submissions will be automatically checked for **code similarity** and **AI-generated content**.
Using code from another student or online source is considered plagiarism.

If you use AI tools (such as ChatGPT or Copilot) to help debug or explain concepts, you must **document this clearly in your README**. Only submit code that reflects your **own understanding and work**.