SDN Project — Mininet + POX Controller (Data-Center Emulation)

Course: Cloud Computing / SDN module
Controller: POX (OpenFlow 1.0)
Topology: Clos/Fat-Tree on Mininet

Learning goals

- Emulate a small data-center network on Mininet and manage it with a custom POX controller.
- Implement shortest-path routing over a multi-rooted tree (Clos/fat-tree) with multiple equal-cost paths.
- Enforce tenant isolation via a packet-level firewall (ARP and IP).
- Support transparent host/VM migration with header rewrite at the edge.

Starter files (provided)

- `CloudNetController.py` — POX app skeleton with event handlers and helpers (place under `pox/ext/`).
- `firewall_policies.csv` — tenant allow-lists (place under `pox/ext/`).
- `migration_events.csv` — planned migrations `(delay_seconds, old_ip, new_ip)` (place under `pox/ext/`).
- `fattree_topo.py` — Mininet script that builds a k-ary fat-tree and connects to a remote controller.

  The CSVs are read at controller startup. Example rows appear later in this brief.

What you must deliver

1. **Working controller app** in POX that:
   a) discovers the topology,
   b) computes and stores all equal-length shortest paths between switches, and
   c) installs flow rules for new IP flows along a randomly selected shortest path.
2. **Firewall isolation**: only hosts in the same tenant may communicate (applies to ARP and IP).
3. **Host migration**: transparently reroute traffic from an old host IP to a new host IP after a scheduled migration delay, with header rewrites at the first hop so peers still “see” the old IP/MAC.
4. **Sanity checks** and short report: successful `pingall` and targeted pings under each mode (routing only, firewall on, migration on).

Environment & run order

1. Start POX from its repo root, loading your app and the discovery module. Examples:
2. Routing only:
   `./pox.py openflow.discovery CloudNetController --firewall_capability=False --migration_capability=False`
3. Firewall only:
   `./pox.py openflow.discovery CloudNetController --firewall_capability=True --migration_capability=False`
4. Migration only (or both):
   `./pox.py openflow.discovery CloudNetController --firewall_capability=False --migration_capability=True`
5. In another terminal, launch the Mininet topology and wait for discovery to converge, then run `pingall`.
   Example: `sudo python fattree_topo.py -k 8`

Part A — Topology, Shortest-Path Routing, Firewall

A.0 Read the skeleton code

- Inspect `fattree_topo.py` (builds a Clos/fat-tree and attaches to a remote controller) and `CloudNetController.py` (packet handlers, per-switch path storage, helpers for flow mods/packet outs). Fill the parts marked for you in the controller skeleton.

A.1 Build the topology

- Create a multi-rooted tree (Clos-style): core, aggregation, and edge layers. Edge switches connect hosts; agg switches connect to all cores; edges connect to all aggs in the pod. You may use the provided `fattree_topo.py` and parameter `-k`.

A.2 Shortest-path routing

- Maintain an adjacency map from `openflow.discovery` events.
- Build a directed graph and compute **all shortest simple paths** between every switch pair. Store per-destination path lists inside each `SwitchWithPaths` object.
- Upon each new IP `PacketIn`, if the destination host is known, pick one shortest path at random and **install rules from destination toward source**. For the current packet, send a `PacketOut` at the source switch so traffic starts immediately while rules propagate.
- Idle timeout 10 s is acceptable; hard timeout may remain infinite.

A.3 Firewall isolation (ARP + IP)

- CSV input format (example for 16 hosts):

```
1,10.0.0.1,10.0.0.3,10.0.0.5,10.0.0.7,10.0.0.9,10.0.0.11,10.0.0.13,10.0.0.15
2,10.0.0.2,10.0.0.4,10.0.0.6,10.0.0.8,10.0.0.10,10.0.0.12,10.0.0.14,10.0.0.16
```

- Map each IP to a tenant ID. When handling ARP/IP packets, allow only **same-tenant** communication; otherwise install a drop rule on the switch that raised the event. The illegal triggering packet can be ignored.

Sanity checks (A): - Routing-only run: `pingall` succeeds across all hosts.
- Firewall run: odd↔odd and even↔even succeed; odd↔even fail.

Part B — Transparent Host Migration

B.1 Input

- `migration_events.csv` rows: `delay_seconds,old_ip,new_ip`
  Example: `180,10.0.0.1,10.0.0.5`

B.2 Behavior

- At the scheduled time, delete existing rules toward the **old IP** to “force” new PacketIns.
- For traffic headed to the **old IP**, install a **forward migrated path** toward the new host and rewrite headers (dst MAC/IP) at the first switch so intermediates match the new headers.
- For reverse traffic sourced from the **new host**, install a **reverse migrated path** and rewrite headers (src MAC/IP) so peers believe they are still talking to the old host.
- Update the controller’s ARP map so future ARP resolutions for the old IP point to the new attachment.

Sanity checks (B): - Continuous ping to the old IP survives the migration with no packet loss beyond a brief convergence blip.
- Verify with tcpdump/wireshark if available.

Grading rubric (indicative)

- Topology build and discovery integration: **20%**
- Shortest-path computation and rule installation: **30%**
- Firewall (correct policy enforcement for ARP/IP): **25%**
- Migration (correct header rewrites and path installs): **25%**

Academic integrity

Submit your own implementation. Cite external libraries and provide a brief README with run instructions.