

Policy-Based Routing

Lesson 03 · Linux Networking · ~10 min read

Why a second routing decision exists

In Lesson 02 you learned that every outbound packet goes through a longest-prefix match against the `main` routing table. That model works perfectly when you have one uplink and every host in every subnet should exit through the same gateway. The moment you have two ISPs, a management network that must always go through a VPN, or application traffic that needs to be steered by firewall mark rather than destination — the single-table model breaks down.

Consider a dual-homed host with one interface to ISP-A and one to ISP-B. You want traffic from your internal `10.10.0.0/24` subnet to exit ISP-A, and traffic from a second subnet `10.20.0.0/24` to exit ISP-B. You cannot express this in a single routing table because both subnets share the same default route slot (`0.0.0.0/0`), and only one default wins at a time.

Linux solves this with a two-step lookup. Before the kernel consults a routing table, it walks a priority-sorted list of **rules**. Each rule says "if this packet matches these selectors, go look in *this* routing table." Only when a rule matches does the kernel actually touch a routing table. The rules themselves contain no route information — they just map packets to table IDs. This two-step design means you can have arbitrarily many routing tables and express almost any forwarding policy using combinations of source address, firewall mark, incoming interface, TOS bits, and other metadata.

Anatomy of an ip rule

A rule has three parts: a **preference** (priority number), one or more **selectors**, and a **target table**. Lower preference numbers are evaluated first. The first rule that matches wins — the kernel stops walking the list and uses the named table for the actual route lookup.

```
ip rule add pref 100 from 10.10.0.0/24 lookup 200
```

This says: for any packet whose source address falls inside `10.10.0.0/24`, look up the route in table `200`. Table `200` is a user-defined table that you populate separately with `ip route add ... table 200`. If the source address does not match, the rule is skipped and the next one (by preference) is tried.

Every Linux system starts with three built-in rules that you cannot delete:

- **pref 0** — `from all lookup local`: handles your own IP addresses and broadcast; always runs first.
- **pref 32766** — `from all lookup main`: the fallback that causes normal routing to work by default.
- **pref 32767** — `from all lookup default`: a last-resort empty table; almost never used in practice.

Your custom rules go between preference 1 and 32765. The selectors available to you include: `from` (source CIDR), `to` (destination CIDR), `fwmark` (Netfilter mark), `iif` (incoming interface), `oif` (outgoing interface), and `tos` (IP TOS field). You can combine selectors in a single rule — all of them must match for the rule to fire.

The widget below lets you build a rule list and send a test packet through it. Add rules, set the packet's source IP, fwmark, and incoming interface, then click Send to see which rule matches first.

Add up to 5 rules. Send a packet. See which one matches first.

pref: 100

selector: from CIDR ▼

e.g. 10.10.0.0/24

→ table: main ▼

+ Add rule

No rules. Click "+ Add rule".

test packet — src: 10.10.0.5

fwmark: 0x42

iif: eth0

Send

Clear all

Add some rules, then click Send.

Three use cases you will actually meet

Source-based routing

The most common PBR scenario is a host or router with two uplinks where each source subnet must always exit via a specific gateway. You create one routing table per uplink, populate each table with a default route pointing at the appropriate gateway, then add `ip rule` entries that steer each source subnet to the right table.

```
# Table 200: traffic from 10.10.0.0/24 exits via ISP-A
ip route add default via 203.0.113.1 dev eth0 table 200
ip rule add pref 100 from 10.10.0.0/24 lookup 200

# Table 201: traffic from 10.20.0.0/24 exits via ISP-B
ip route add default via 198.51.100.1 dev eth1 table 201
ip rule add pref 101 from 10.20.0.0/24 lookup 201
```

Traffic from any other source still falls through to the default pref 32766 rule and uses the main table as normal. Source-based routing is the lightest form of PBR — no Netfilter involvement, no conntrack, just `ip rule` and `ip route`.

fwmark-driven routing

Sometimes you cannot classify traffic by source address alone. You want to route all DNS queries via one path, or steer SSH traffic through a VPN regardless of where it originates. In these cases you use Netfilter's `MARK` target to paint a numeric mark onto matching packets in the mangle table, then write an `ip rule` that matches on that mark.

```
# Mark all TCP traffic destined for port 22
iptables -t mangle -A OUTPUT -p tcp --dport 22 -j MARK --set-mark 0x42

# Route all marked packets through table 300 (VPN)
ip route add default via 10.99.0.1 dev tun0 table 300
ip rule add pref 90 fwmark 0x42 lookup 300
```

The mark only exists in kernel memory for the lifetime of the packet — it is not part of the IP header. Once the packet leaves the machine the mark is gone. For most use cases this is fine; for TCP flows spanning many packets you need CONNMARK, covered in the next section.

Multi-WAN with metrics

When you have two uplinks and want automatic failover rather than strict partitioning, you install two default routes in the main table with different metrics. The kernel always prefers the lower metric when both routes are present. When ISP-A's link drops, you remove or invalidate its route and the kernel automatically falls over to the higher-metric ISP-B route.

```
ip route add default via 203.0.113.1 dev eth0 metric 100    # primary
ip route add default via 198.51.100.1 dev eth1 metric 200   # backup
```

You can also overlay `ip rule` on top of this: add a rule that forces a specific source CIDR to a dedicated table, while leaving the two-metric default routes in main for everything else. This gives you both source-based partitioning and metric-based failover simultaneously.

CONNMARK and stateful PBR

The `MARK` target paints a single packet. The moment that packet clears the mangle table, the mark exists only as long as the packet is in flight. The next packet in the same TCP flow — say, the second segment of an HTTP response — arrives with mark zero. If your `ip rule fwmark` only sees marked packets it will route the first packet to table 300 via the VPN, and the second packet will fall through to the main table via the regular gateway. The result is an asymmetric flow where replies and retransmits take different paths, which almost always causes TCP to stall or reset.

The fix is **CONNMARK**. CONNMARK stores the mark in the conntrack entry for the flow, not just on the individual packet. Two iptables targets handle this:

- `CONNMARK --save-mark`: copies the packet's fwmark into the conntrack entry. Run this after `MARK --set-mark` on the first (NEW) packet.
- `CONNMARK --restore-mark`: copies the mark from the conntrack entry back onto every subsequent packet in the flow. Run this early in PREROUTING so the mark is present before `ip rule` is consulted.

A complete three-rule mangle chain for a PBR scenario looks like this:

```
iptables -t mangle -A PREROUTING -j CONNMARK --restore-mark
iptables -t mangle -A PREROUTING -m conntrack --ctstate NEW -j MARK --set-mark
```

```
iptables -t mangle -A PREROUTING -m mark ! --mark 0 -j CONNMARK --save-mark
```

Rule order matters. The restore runs first so that established packets inherit their saved mark immediately. Only packets in state NEW then go through the classifier. The save runs last to ensure the mark is written back to conntrack whenever a new mark is set. With this chain in place every packet in a flow — from SYN to FIN — carries the same mark and takes the same routing path.

Use the playground below to see what happens when you send a four-packet flow with and without CONNMARK rules enabled.

iptables -t mangle -A PREROUTING

☒ -m conntrack --ctstate NEW -j MARK --set-mark 0x42
☐ -j CONNMARK --save-mark
☐ -j CONNMARK --restore-mark

ip rule

☒ from all fwmark 0x42 lookup 200
ip route show table 200:
default via 10.99.0.1

conntrack entry

(empty — no flow yet)

P1 NEW P2 EST P3 EST P4 EST

P1

P2

P3

P4

Send 4-packet flow

Toggle CONNMARK rules. Then send a flow.

Multi-WAN in practice

Running two uplinks reliably is harder than it looks. The metric-based failover approach described above requires something external — a health-check daemon, [keepalived](#), or a custom script — to detect when a link dies and remove its route. Until the route is removed the kernel keeps sending packets out the dead interface, where they are silently lost.

A production multi-WAN setup typically combines three pieces: metric-based routes in the main table for automatic failover, ECMP (equal-cost multi-path) or weighted routing

for load-sharing when both links are healthy, and `ip rule` with source-based rules to pin specific services to a specific WAN regardless of metrics. For example, a VoIP server might always exit via the low-latency fibre link even if the cable modem temporarily has a lower metric because its monitor script hasn't run yet.

Stateful services — anything using TCP — need CONNMARK if you want all packets in a flow to take the same path. Without it, the first packet of a TCP handshake might go out ISP-A but the ACK from the remote server comes back via ISP-B's IP. The remote host sees a reply from an unexpected address and drops the connection.

The simulator below gives you two WANs with adjustable metrics. Kill either WAN, toggle policy mode, and send packets to see how the routing decision changes.

WAN-A metric: 100

WAN-B metric: 200

Kill WAN-A

Kill WAN-B

Policy: OFF

Send packet from 10.10.1.5

host (10.10.1.5)

WAN-A
metric 100

WAN-B
metric 200

Try killing a WAN or toggling policy mode, then send a packet.

The rp_filter trap

Reverse-path filtering (`rp_filter`) is a kernel anti-spoofing mechanism. When enabled, the kernel checks whether the source address of an arriving packet is reachable via the same interface the packet arrived on. If the routing table says the source IP should have come from a different interface, the packet is dropped silently. For a standard single-homed host this is a useful protection. For a host doing PBR it is a frequent source of mysterious connectivity failures.

The scenario: you have set up a rule that routes replies from table 200 out through `eth1`, but the packet arrived on `eth0`. The kernel's strict `rp_filter` check (value `1`) sees that the main table routes back to that source via `eth0`, but the packet came in on `eth1`, so it drops the packet before any routing decision is made. Your PBR configuration is correct but you see no traffic.

Symptom: ping replies vanish when using PBR with non-default interfaces.

Cure: `sysctl -w net.ipv4.conf.<iface>.rp_filter=2` (loose mode).

Loose mode (value `2`) only requires that the source address be reachable via *any* interface, not necessarily the one the packet arrived on. This is almost always the right setting for interfaces participating in PBR. You can also set `net.ipv4.conf.all.rp_filter=2` to apply it globally, but prefer per-interface settings to minimize the attack surface.

To make the change persistent across reboots, add it to `/etc/sysctl.d/`:

```
echo "net.ipv4.conf.eth1.rp_filter=2" | sudo tee /etc/sysctl.d/99-pbr.conf
sudo sysctl -p /etc/sysctl.d/99-pbr.conf
```

Under the hood

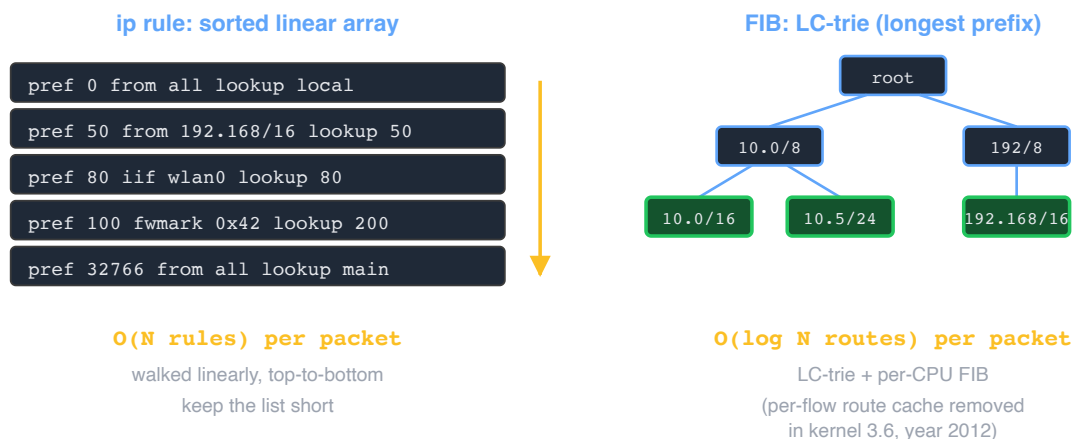
It is useful to understand what the kernel actually does when it evaluates rules and routes, because the performance characteristics differ between the two structures.

The rule list (`ip rule show`) is stored as a sorted linked list in the kernel. When a packet arrives, the kernel walks it from the lowest preference number upward, checking each rule's selectors against the packet's metadata. This is an $O(N)$ linear scan — if you have 200 rules, the worst-case packet may check all 200 before finding a match or falling through to main. Keep the rule list short. Dozens of rules are fine; thousands are not.

The routing table itself — the Forwarding Information Base (FIB) — is stored as an LC-trie (level-compressed trie), which allows longest-prefix lookups in $O(\log N)$ time. The kernel also maintains a per-CPU cache of recent lookups to amortize repeated lookups for the same flow. Note that the per-flow route cache that existed in older kernels was

removed in kernel 3.6 (2012); modern kernels do the trie walk on every packet, relying on CPU caches for speed rather than a software route cache.

The diagram below shows both structures side by side.



Rules are linear, FIB is a tree.
Order matters in ip rule. Specificity matters in ip route.

Recap

- Linux looks up an *ip rule* before consulting any routing table.
- Selectors include `from`, `to`, `fwmark`, `iif`, `oif`, `tos`.
- The first matching rule wins; lower `pref` = higher priority.
- CONNMARK `--save-mark` / `--restore-mark` persists a mark across every packet in a flow.
- Rule lists are walked linearly; keep them short. FIB is a trie — specificity matters.