

How Linux Decides Where Packets Go

Lesson 02 · Linux Networking · ~10 min read

The question Linux asks for every packet

Every time your machine generates an outbound packet — whether it comes from a browser, a `curl` command, or a container — the kernel faces the same two-part question: *Which interface should carry this packet out?* and *Should I forward it through a gateway, or is the destination directly reachable on this link?* This decision is routing, and it happens entirely inside the kernel, long before anything touches the wire.

Routing is separate from filtering. In Lesson 1 we saw that netfilter hooks can accept or drop packets; routing happens a step earlier, at the `PREROUTING` and `OUTPUT` hooks, before the `FORWARD` vs `INPUT` split. If the kernel cannot find any route for a destination, the packet is silently discarded with an ICMP "Network unreachable" returned to the sender. No firewall rule is even consulted — the packet never made it that far.

Understanding routing is therefore foundational: a misconfigured route can look exactly like a firewall block, and the fastest way to distinguish them is `ip route get <dst>`, which asks the kernel directly what it would do. We will use that command throughout this lesson.

Reading the routing table

Run `ip route show` on almost any Linux host and you will see output like this:

```
default via 192.168.1.1 dev eth0
10.0.5.0/24 via 10.0.5.1 dev veth-a
10.0.0.0/16 via 10.0.0.1 dev eth1
192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.50
```

Each line is a route. A route says: "For packets whose destination matches this network prefix, send them out of this device (`dev`), optionally through this next-hop gateway

(`via`)." The kernel compares the destination IP of every outbound packet against every prefix in the table and picks the *most specific* match — the one with the longest prefix length. A `/24` beats a `/16` beats a `/0` (default). This algorithm is called **longest-prefix match**.

The simulator below lets you type any destination IP and watch the kernel's decision unfold. Each matching route turns green; the winner — the longest-prefix match — is outlined in amber.

Destination:

default via 192.168.1.1 dev eth0

10.0.5.0/24 via 10.0.5.1 dev veth-a

10.0.0.0/16 via 10.0.0.1 dev eth1

192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.50

Winner: 10.0.5.0/24 via 10.0.5.1 dev veth-a (longest-prefix /24)

Default, local, and link routes

Three flavors of route appear in almost every Linux routing table, and it is worth understanding what each one means before moving on to more advanced topics.

A **link route** (also called a "connected" route) is added automatically by the kernel whenever you assign an IP address to an interface. If you give `eth0` the address `192.168.1.50/24`, the kernel immediately inserts a route for `192.168.1.0/24 dev eth0 scope link`. This tells the kernel that all addresses in that subnet are directly reachable — no gateway needed, just ARP for the MAC address and send the frame. You will recognize link routes in `ip route show` by the absence of a `via` keyword and the presence of `scope link`.

A **local route** lives in a separate, special routing table (table 255, called `local`) and covers your own IP addresses. When a packet arrives destined for `192.168.1.50` — your own address — the kernel matches the local table first and delivers the packet to a local socket rather than forwarding it. You rarely need to touch this table manually.

A **default route** is simply the route for `0.0.0.0/0` — the prefix of length zero that matches every IP address. Because longest-prefix match always prefers more specific routes, the default route only wins when nothing else matches. It says "if you have no idea where to send this, forward it to my gateway." A host without a default route can only reach networks it has an explicit route for; internet access requires a default route pointing at an upstream gateway.

When you run `ip route add` or `ip route del`, you are directly manipulating the kernel's Forwarding Information Base (FIB). The change takes effect immediately — there is no service to restart, no cache to flush.

Multiple routing tables — and how to pick one

Linux does not limit you to one routing table. The kernel supports up to 252 user-defined tables (numbered 1–252, plus the three built-in tables: `local`, `main`, and `default`). By default, every route you add goes into the `main` table (table 254), and the kernel only consults that one table. But you can configure it to consult different tables based on the source IP, the incoming interface, or a firewall mark — this is called **policy routing**.

The mechanism is `ip rule`. Rules are evaluated in priority order (lower number = higher priority). Each rule says: "if this packet matches this selector, look it up in this routing table." For example:

```
ip rule add from 10.10.0.0/24 lookup 100
ip rule add from 10.10.0.201/29 lookup 100
```

A common use case is a host with two uplinks (say, a corporate VPN and a direct internet connection) where traffic from the management subnet must always exit via the VPN, while regular traffic takes the direct route. You put the VPN default in table 100, the direct default in table 200, and use `ip rule` to steer traffic from each source subnet to the right table. Without policy routing this would be impossible to express in a single routing table.

The widget below visualizes a three-table setup. Drag the slider to change the source IP's last octet and watch the kernel pick the appropriate routing table.



Network namespaces — building a topology on one host

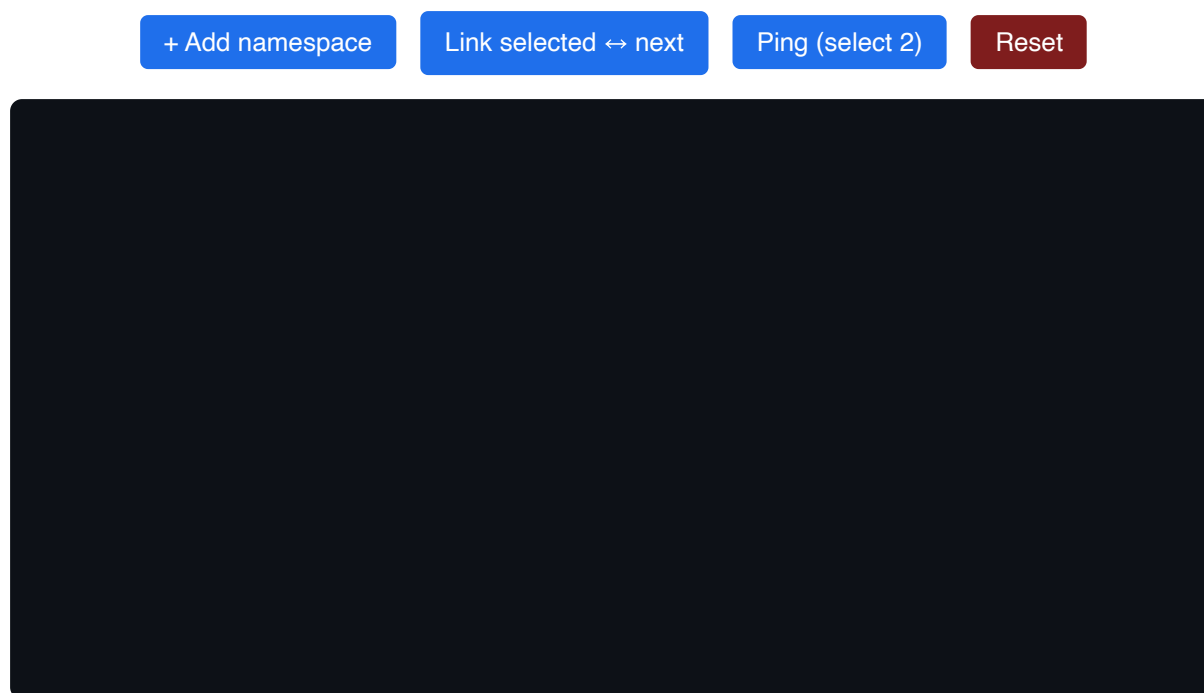
A network namespace is a complete, isolated copy of the Linux networking stack: its own interfaces, its own routing table, its own iptables rules, its own sockets. Processes inside one namespace cannot see or affect the network state of another namespace. The default namespace is simply the one your shell runs in; everything you have seen so far lives there.

Namespaces are not just for containers. They are the standard tool for testing routing topologies on a single developer laptop. Want to simulate a two-router setup? Create three namespaces (client, router, server), connect them with virtual Ethernet pairs (`veth`), assign IPs, enable IP forwarding in the router namespace, and you have a fully functional topology — all without any virtualization overhead, and all gone on reboot.

The essential commands are simple. `ip netns add red` creates a namespace called `red`. `ip netns exec red <cmd>` runs a command inside it. A `veth` pair is a pair of connected virtual interfaces: anything sent into one end comes out the other, as if they were wired together. You create the pair in the default namespace and then move each end into the appropriate namespace with `ip link set <iface> netns <name>`.

One subtle but important point: IP forwarding is per-namespace. Even if the host has `net.ipv4.ip_forward=1` set, a namespace that is meant to act as a router must have forwarding enabled *inside that namespace* via `ip netns exec <router-ns> sysctl -w net.ipv4.ip_forward=1`. Forgetting this is the most common reason a three-namespace topology fails silently.

Use the topology builder below to add namespaces, link them together, and simulate a ping. Namespaces without a veth link between them have no route to each other — the ping will fail with a red cross.



Click "Add namespace" to start.

Diagnostic toolkit

When a packet seems to be going nowhere, the first three tools to reach for are `ip route get`, `traceroute`, and `mtr`. Each answers a different question.

`ip route get <dst>` asks the kernel "what would you actually do with a packet bound for this destination?" It respects policy routing rules and all routing tables, returning the exact interface, gateway, and source address the kernel would use. It is instantaneous and requires no network traffic. If `ip route get` fails or returns the wrong gateway, the problem is in your routing configuration — no firewall investigation needed yet.

`tracert` (or `tracert6` for IPv6) sends probes with incrementing TTL values and listens for ICMP Time Exceeded responses, building a hop-by-hop map from your host to the destination. It is invaluable for pinpointing where in the network path traffic is being dropped or misdirected. Be aware that some routers silently discard ICMP; a missing hop in the output does not always mean the path is broken — only that a particular router chose not to respond.

`mtr` combines `tracert` and `ping` into a live, continuously updating display. It shows per-hop packet loss and round-trip time in real time, making it far easier to spot intermittent problems that a single `tracert` run would miss. Run `mtr --report <dst>` to produce a one-shot summary suitable for pasting into a bug report.

If a packet seems to be going nowhere, run `ip route get <dst>` first. It answers exactly what the kernel would do — saving you from guessing.

Common gotchas

A few mistakes appear so often in routing configurations that they are worth naming explicitly.

Asymmetric routing occurs when the path from A to B takes different interfaces than the path from B to A. This is legal in IP but can break stateful inspection: a firewall that only saw the outbound half of a TCP connection will reject the inbound reply as "INVALID" in conntrack terms. The fix is either to make routing symmetric or to tune conntrack not to track certain flows.

Missing return route is the single most common reason a ping "works one way." If host A can reach host B but B has no route back to A's subnet, B's ICMP echo-reply has nowhere to go and is dropped. Always check both directions. A quick test: after you confirm A can reach B, run `ip route get <A's IP>` from B's perspective (or from within B's namespace).

Reverse path filtering (`rp_filter`) is a kernel feature that drops packets arriving on an interface if the kernel's routing table says the source IP should have arrived on a different interface. This prevents certain spoofing attacks but can cause confusion in asymmetric or multi-homed setups. Check the value with `sysctl net.ipv4.conf.all.rp_filter`; set to `0` to disable or `2` for loose mode.

Metric collisions happen when two routes to the same prefix have the same metric (cost). The kernel picks one — which one is implementation-defined. If you add a failover route, give it a higher metric than the primary so the primary always wins when both are present.

Recap

- Linux looks up an interface + gateway for every outbound packet.
- The routing table is consulted by longest-prefix match.
- Multiple routing tables exist; `ip rule` chooses between them.
- Network namespaces let you build multi-router topologies on a single host.
- `ip route get` answers "what would the kernel do?" — your first diagnostic.