Networking Foundations Course

Flavio Copes

Updated August 8, 2026

Contents

Learn how local networks and the Internet move data through links, IP addresses, routes, TCP, UDP, ports, NAT, firewalls, and practical diagnostics.

What you'll learn: Trace one connection from an application through the local network and Internet, inspect each layer, and identify where a failed connection stops.

How networks fit together

Understand endpoints, links, interfaces, layers, packets, encapsulation, and the complete path.

What a network is

Identify endpoints, interfaces, links, and intermediary devices in a small computer network.

A computer network lets separate devices exchange data. Phones, laptops, servers, printers, and sensors are **endpoints**. They are where data starts and where it ends up. Switches and routers are intermediary devices that move data between them without being the final destination.

A device connects through a network **interface**. An interface may use an Ethernet cable, Wi-Fi radio, virtual adapter, or another link technology. One computer can have several interfaces and several addresses at the same time.

You can see your own interfaces right now. On Linux:

```
ip -brief link
```

```
lo               UNKNOWN        00:00:00:00:00:00
eth0             UP             3c:22:fb:9a:12:40
wlan0            DOWN           a4:83:e7:2c:88:01
```

`lo` is the loopback interface, which only talks to the machine itself. `eth0` is a wired Ethernet interface, currently `UP`. `wlan0` is a Wi-Fi interface that's not connected. Each hardware interface has its own link-layer address.

On macOS, run `ifconfig` and look for `en0`, which is usually your main interface. The `status: active` line tells you it's connected.

Links join interfaces

A **link** connects interfaces that can communicate directly, with no router in between. Your laptop's Wi-Fi interface and your home router's Wi-Fi interface share one link. A network can contain many links joined by routers.

The Internet is the largest example: many independently operated networks agreeing to exchange IP packets. Your home network is one network. Your Internet provider runs another. The server hosting this page sits in yet another. Data crosses all of them.

You can prove your machine knows about at least one other device — the router that connects your link to everything else:

```
ip route | grep default
# default via 192.168.1.1 dev eth0
```

That output says: "anything I can't deliver locally goes to the device at `192.168.1.1`, through my `eth0` interface." That device is a router, your first intermediary.

Why this vocabulary matters

Start every network diagram with devices, interfaces, and links. When something breaks, "the network is down" tells you nothing. "My `eth0` interface has no address" or "the link to the router is down" points at one specific thing to fix.

Protocol names make more sense once you know which two things are communicating at each step. The rest of this course builds on exactly these three words: endpoint, interface, link.

LANs, WANs, and the Internet

Separate a local network from routed networks and understand how the Internet joins independent networks.

A **local area network**, or LAN, connects nearby devices through one organization's links. A home Ethernet and Wi-Fi network is a common example. Everything behind your home router — laptop, phone, printer, TV — sits on one LAN.

A router joins IP networks. Traffic for a device on the same local network stays local. Traffic for another network is sent to a router, which chooses the next hop.

You can see this boundary on your own machine:

```
ip route
```

```
default via 192.168.1.1 dev wlan0
192.168.1.0/24 dev wlan0 scope link
```

The second line is your LAN: addresses in `192.168.1.0/24` are reachable directly, on the local link. The first line says everything else goes to the router at `192.168.1.1`. That single `default` line is the edge of your local network. On macOS, `netstat -rn` shows the same two facts.

Where the WAN starts

A **wide area network**, or WAN, connects networks across a larger area. Your Internet provider operates a WAN that joins thousands of homes and offices to its own infrastructure.

But the Internet is not one company's WAN. It is a network of networks using shared Internet protocols. Your provider hands packets to other providers, who hand them to data centers, until they reach the destination network.

Watch a packet leave your LAN and cross those networks:

```
traceroute -n 1.1.1.1
```

```
 1  192.168.1.1     2.1 ms
 2  100.64.12.1     8.4 ms
 3  62.101.93.45   12.9 ms
```

...

Hop 1 is your router — the last device on your LAN. Hop 2 is already inside your provider's network. Every later hop belongs to some network you don't control. On macOS the command is the same; on some Linux systems install it or use `tracepath`.

Same packet, different scopes

These labels describe scope and ownership, not a different packet format. The same IP packet can cross your LAN, your provider, and several other networks before reaching a server. Nothing about the packet changes because it moved from a "local" to a "wide" network.

This matters when you diagnose problems. A failure at hop 1 is your problem: cable, Wi-Fi, router. A failure at hop 2 or beyond belongs to your provider or someone further away, and no amount of local configuration will fix it.

Layers and protocols

Use a compact four-layer model to place application, transport, Internet, and link protocols.

Networking is easier to reason about in layers. Each layer solves a narrower problem and provides a service to the layer above it.

- **Application:** HTTP, DNS, SSH, and the rules applications understand
- **Transport:** TCP or UDP communication between processes
- **Internet:** IP addressing and routing between networks
- **Link:** delivery across one local link, such as Ethernet or Wi-Fi

The model is a tool, not a physical stack inside a cable. Implementations sometimes cross layer boundaries, but the separation still helps you ask precise questions.

Watch the layers in one request

You can see the layers take turns in a single `curl` call:

```
curl -v https://flaviocopes.com/ -o /dev/null
```

Read the output top to bottom:

```
* Host flaviocopes.com:443 was resolved.
* IPv4: 104.21.4.157, 172.67.132.90
*   Trying 104.21.4.157:443...
* Connected to flaviocopes.com (104.21.4.157) port 443
* SSL connection using TLSv1.3
> GET / HTTP/2
< HTTP/2 200
```

The first two lines are DNS, an application-layer protocol, turning a name into IP addresses. `Trying ... Connected` is the transport layer opening a TCP connection to port 443, carried by IP packets across many links you never see. The TLS line secures that connection. Only then does HTTP — the application protocol you actually wanted — send `GET /` and receive a `200`.

One command, four layers, each doing one job in sequence.

Why layers matter for debugging

When a website fails, "the network is broken" is too vague. The failure might be name resolution, routing, a TCP port, TLS, or the HTTP application. Layers give each possibility a name.

Compare these two failures:

```
curl https://doesnotexist.flaviocopes.com/
# curl: (6) Could not resolve host

curl https://flaviocopes.com:9999/ --connect-timeout 3
# curl: (28) Connection timed out
```

The first error is an application-layer DNS failure: the name has no address. The second is a transport failure: the name resolved fine, but nothing answered on TCP port 9999. Two different layers, two completely different fixes.

My advice is to name the layer before touching any configuration. If DNS failed, don't restart your router. If TCP timed out, don't edit `/etc/hosts`. The model exists so you can point at the right problem.

Encapsulation and one complete path

Follow application data as transport, IP, and link headers are added and removed along a path.

Suppose a browser sends an HTTP request. The request itself is just bytes of application data. Before those bytes reach the wire, each layer wraps them with its own header. TCP adds transport information. IP adds source and destination addresses. The local link adds the information needed for the next local hop.

> **Diagram: Network encapsulation and decapsulation.** The sender wraps HTTP data in a TCP segment, IP packet, and link frame. The receiver removes those layers in reverse order. This wrapping is called **encapsulation**. Think of it as envelopes inside envelopes: the HTTP request sits inside a TCP segment, which sits inside an IP packet, which sits inside an Ethernet or Wi-Fi frame.

The receiving side processes the headers in the opposite direction. The link layer strips the frame, IP strips its header, TCP strips its header, and each layer passes its payload upward until the web server receives the exact HTTP request the browser wrote.

See the layers around real data

You can watch encapsulation on your own machine. Start a capture of local web traffic, then make a request:

```
sudo tcpdump -ni any -c 4 port 443
```

```
IP 192.168.1.20.54312 > 104.21.4.157.443: Flags [S], seq 1187624
IP 104.21.4.157.443 > 192.168.1.20.54312: Flags [S.], ack 1187625
```

Every line describes one packet, and every field belongs to a different layer. `IP 192.168.1.20 > 104.21.4.157` is the Internet layer. The `.54312` and `.443` port numbers belong to TCP. Add `-e` to the command and tcpdump also prints the MAC addresses from the link frame around it all. Several protocols, one piece of data.

What changes along the path, and what doesn't

Routers normally replace the link framing at every hop while forwarding the IP packet. The frame that leaves your laptop is addressed to your router. The router strips it, reads the IP destination, and builds a fresh frame for the next link.

The end-to-end transport conversation belongs to the endpoints; each link frame belongs only to one local hop. The TCP header your browser wrote arrives at the server untouched. The Ethernet frame around it was rebuilt a dozen times along the way.

This is why a packet capture can show several protocols around the same piece of application data, and why the same capture looks slightly different depending on which link you take it from.

Check your understanding: network models

Check endpoints, interfaces, LANs, routers, layers, encapsulation, and the scope of a link frame.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Local networks

Follow frames through Ethernet or Wi-Fi using MAC addresses, switches, ARP, and Neighbor Discovery.

Frames and MAC addresses

Understand how an Ethernet frame and MAC addresses deliver data across one local network segment.

Ethernet carries data in **frames**. A frame has source and destination link-layer addresses, a field identifying the carried protocol, payload data, and error-detection information. The frame is the unit of delivery on one local network segment.

Ethernet link-layer addresses are commonly called **MAC addresses**. They are 48-bit identifiers usually written as six hexadecimal pairs. They identify interfaces for delivery on the local Ethernet network. They are not Internet-wide routes.

Every interface you own has one. Look at yours:

```
ip link show eth0
# link/ether 3c:22:fb:9a:12:40 brd ff:ff:ff:ff:ff:ff
```

On macOS:

```
ifconfig en0 | grep ether
# ether a4:83:e7:2c:88:01
```

The `3c:22:fb` half identifies the manufacturer of the network chip. The second half is unique to the interface. The `brd ff:ff:ff:ff:ff:ff` value is the broadcast address — a frame sent there is delivered to every interface on the segment.

Two addresses, two different jobs

Here's the part that confuses beginners. When a laptop sends an IP packet to its router, the Ethernet destination is the router interface's MAC address. The IP destination can still be a server on the other side of the world.

Check which MAC address your machine actually uses to reach the Internet:

```
ip neighbor
# 192.168.1.1 dev eth0 lladdr 9c:53:22:aa:04:1e REACHABLE
```

That entry maps your router's IP address to its MAC address. Every packet you send to any website is wrapped in a frame addressed to `9c:53:22:aa:04:1e` — the router — regardless of the IP destination inside. On macOS, `arp -an` shows the same table.

Keep the scopes separate: MAC addresses move a frame over one Ethernet link, while IP addresses help routers move a packet across networks. The MAC address changes at every routed hop. The IP addresses normally stay the same end to end.

A common mistake

People sometimes try to firewall or identify remote servers by MAC address. That can't work. You never see a remote machine's MAC address — only the MAC of the last local device that handed you the frame, usually your router. MAC addresses are local facts. Anything beyond your own link is IP territory.

How switches forward frames

Follow a frame through an Ethernet switch and understand learning, forwarding, flooding, and broadcast.

An Ethernet switch connects devices on a local network. Each device plugs into a port, and the switch's job is to deliver frames only where they need to go.

It does this by **learning**. As frames arrive, the switch reads the source MAC address of each one and remembers which port it came from. Over time it builds a table like this:

```
MAC address         Port
3c:22:fb:9a:12:40   1     (your laptop)
9c:53:22:aa:04:1e   4     (the router)
b8:27:eb:44:c1:09   7     (a Raspberry Pi)
```

For a known destination MAC address, the switch forwards the frame only through the matching port. Your laptop's traffic to the router goes out port 4 and nowhere else. Other devices never see it.

For an unknown destination, it initially floods the frame through the other relevant ports. Whichever device answers reveals its port, the switch learns the mapping, and flooding stops for that address.

Broadcast reaches everyone

A broadcast destination — `ff:ff:ff:ff:ff:ff` — is delivered throughout the **broadcast domain**: every port on the switch, and every switch connected to it. ARP requests are a familiar example. You can watch them arrive at your own machine:

```
sudo tcpdump -ni eth0 -c 3 broadcast
```

```
ARP, Request who-has 192.168.1.42 tell 192.168.1.1
ARP, Request who-has 192.168.1.17 tell 192.168.1.20
```

Your machine receives these even though none are addressed to it. That's broadcast: the switch copies the frame to every port. On a busy LAN you'll see a steady trickle of them.

Routers normally separate broadcast domains instead of forwarding Ethernet broadcasts between networks. That's why an ARP request in your home never reaches your neighbor's LAN, and why broadcast-heavy protocols stay local.

What a switch does not do

A switch makes local forwarding decisions based on MAC addresses. It does not read IP addresses and it does not replace the IP routing decision made by the sending host or a router.

This distinction matters when you debug. If two devices on the same switch can't talk, the problem is link-level: cabling, the switch itself, or the MAC table. If a device can reach its neighbors but not the Internet, the switch is doing its job fine — look at routing instead.

ARP resolves IPv4 neighbors

Use ARP to map a local IPv4 next-hop address to the Ethernet address needed for a frame.

Before sending an IPv4 packet over Ethernet, a host needs the MAC address of the next hop. The Address Resolution Protocol, or **ARP**, maps a local IPv4 address to an Ethernet address.

The sender broadcasts a request asking which interface owns an IPv4 address. The owner replies with its MAC address, and the sender stores the answer temporarily in its neighbor cache.

For a destination on the same subnet, the next hop is the destination itself. For a remote destination, the next hop is usually the default router. ARP resolves the router's local address, not the remote server's address.

A stale or missing neighbor entry can stop communication before the first packet leaves the LAN. Inspecting the neighbor cache is therefore a useful local diagnostic.

IPv6 Neighbor Discovery

Understand how IPv6 discovers neighbors, routers, prefixes, and link-layer addresses without ARP.

IPv6 uses **Neighbor Discovery** instead of ARP. It is built on ICMPv6 and handles several related jobs on a local link: finding link-layer addresses, finding routers, learning prefixes, and checking that neighbors are still alive.

Neighbor Solicitation and Neighbor Advertisement messages discover link-layer addresses and test whether neighbors remain reachable. They do the job ARP does for IPv4: ”who has this address, and what's your MAC?”

Router Advertisements tell hosts about routers and network prefixes. Your machine listens for them and can configure itself from what it hears, with no DHCP server involved.

Look at your neighbor cache

Your system keeps the results in a neighbor table, just like the IPv4 ARP cache:

```
ip -6 neighbor
```

```
fe80::9e53:22ff:feaa:41e dev eth0 lladdr 9c:53:22:aa:04:1e router REACHABLE
2a01:e0a:4f2:1::1 dev eth0 lladdr 9c:53:22:aa:04:1e STALE
```

`REACHABLE` means the entry was confirmed recently. `STALE` means it's remembered but unverified — the next packet will trigger a fresh check. The `router` flag marks a device that sent Router Advertisements. On macOS, `ndp -an` shows the same table.

You can populate the cache by pinging the all-nodes multicast address on your link:

```
ping6 -c 2 ff02::1%eth0
```

Every IPv6 device on the link may answer. On macOS use your interface name, such as `ff02::1%en0`. The `%eth0` part is the zone identifier: link-local addresses exist on every link, so you must say which one you mean.

Multicast, not broadcast

IPv6 uses multicast for these exchanges rather than Ethernet-wide ARP broadcasts. A Neighbor Solicitation goes to a small computed multicast group, so most machines on the LAN never process it. This is one of the quiet efficiency wins of IPv6.

A host also performs Duplicate Address Detection before relying on a new address: it solicits its own tentative address, and silence means the address is free to use.

Don't firewall ICMPv6 away

Do not block all ICMPv6 as a generic security measure. IPv6 depends on specific ICMPv6 messages for normal local operation and path feedback. A machine that can't send or receive Neighbor Discovery messages can't find its router or its neighbors — the result looks like a dead network with a cable that's clearly plugged in. If IPv6 works for a minute and then dies, check whether a firewall rule is eating Neighbor Discovery or Router Advertisements.

Check your understanding: local networks

Check Ethernet frames, MAC address scope, switches, broadcasts, ARP, next hops, and IPv6 Neighbor Discovery.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

IP addresses and subnets

Read IPv4 and IPv6 addresses, prefixes, subnets, and special address ranges.

Read an IPv4 address and prefix

Interpret an IPv4 address together with its CIDR prefix length instead of treating the address alone as a network.

An IPv4 address contains 32 bits, usually written as four decimal numbers such as `192.0.2.34`. Each number is one byte, 0 through 255. An interface also needs a **prefix length**, such as `/24`, and the address means very little without it.

The prefix length says how many leading bits identify the network. The remaining bits identify addresses inside that network. `192.0.2.34/24` belongs to the prefix `192.0.2.0/24`: the first 24 bits (the first three numbers) name the network, and the last 8 bits name this particular host.

Look at your own interface and you'll always find the two together:

```
ip -4 address show
```

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP>
    inet 192.168.1.20/24 brd 192.168.1.255 scope global dynamic eth0
```

The `inet` line is the fact that matters: this interface is `192.168.1.20`, inside the network `192.168.1.0/24`. On macOS, `ifconfig en0` shows the same information as an address plus `netmask 0xffffff00` — that hexadecimal mask is `/24` written the old way, 24 one-bits followed by 8 zero-bits.

The prefix decides who's local

Two addresses that look similar are not necessarily local neighbors. The prefix length determines whether the sender considers a destination on-link or sends it to a router.

Ask your kernel directly how it would reach two destinations:

```
ip route get 192.168.1.50
# 192.168.1.50 dev eth0 src 192.168.1.20

ip route get 192.168.2.50
# 192.168.2.50 via 192.168.1.1 dev eth0 src 192.168.1.20
```

`192.168.1.50` matches your `/24`, so the answer has no `via`: deliver directly on the link. `192.168.2.50` differs in the third byte, falls outside the prefix, and gets a `via 192.168.1.1` — hand it to the router. One byte of difference, a completely different delivery path.

Record both, always

Always write down an interface as address plus prefix. An address without its prefix is incomplete configuration information — you can't tell which destinations are local, what the broadcast address is, or whether two machines should see each other directly.

A classic misconfiguration: two machines on the same cable, one set to `/24` and the other to `/16`. Each computes a different idea of what's local. Traffic flows one way but not the other, and everything half-works. Whenever direct communication between neighbors fails, compare prefixes first.

Calculate a subnet boundary

Find the network range represented by a CIDR prefix and see how a longer prefix creates a smaller subnet.

A CIDR prefix is a block of addresses sharing the same leading bits. A `/24` fixes 24 bits and leaves 8 bits variable, so it contains 256 IPv4 addresses. Every bit you add to the prefix cuts the block in half:

```
/24: 256 addresses
/25: 128 addresses
```

```
/26:  64 addresses
/27:  32 addresses
```

A `/26` fixes two more bits than a `/24` and contains 64 addresses. So which 64 addresses does `192.0.2.70/26` sit among? That's the calculation this lesson is about.

Do the math once by hand

The last byte of `192.0.2.70` is 70. A `/26` means blocks of 64 addresses, so the boundaries in that byte are 0, 64, 128, and 192. The value 70 falls between 64 and 128.

That means `192.0.2.70/26` falls in the block from `192.0.2.64` through `192.0.2.127`. The network address is `192.0.2.64/26`, and `192.0.2.127` is the last address in the block.

The general recipe: work out the block size (2 raised to the number of host bits), find the largest multiple of that size that fits below your address, and that multiple is the start of the subnet.

Then let the computer check you

Python ships an `ipaddress` module that does exactly this:

```
python3 -c "import ipaddress; n = ipaddress.ip_network('192.0.2.70/26', strict=False); print(
```

```
192.0.2.64/26 192.0.2.64 192.0.2.127
```

`strict=False` tells Python you're giving it a host address, not a network address, and it should find the containing network. The output confirms the hand calculation: network `192.0.2.64/26`, first address `.64`, last address `.127`.

Try a few of your own. What block contains `10.1.7.200/27`? Compute it by hand, then verify. The boundaries in the last byte for a `/27` are multiples of 32, so the answer is `10.1.7.192` through `10.1.7.223`.

Why this matters in practice

Subnetting is binary boundary calculation, not guesswork based on decimal dots. Modern routing uses prefixes instead of the old class A, B, and C assumptions, so nothing about the address itself tells you where its network starts.

The classic mistake is assuming two addresses are in the same subnet because three dots' worth of digits match. `192.0.2.60/26` and `192.0.2.70/26` look close, but `.60` lives in the `192.0.2.0`–`.63` block and `.70` lives in the next one. They need a router to talk. When neighbors mysteriously can't reach each other, calculate the boundaries before you trust your eyes.

Recognize special IPv4 ranges

Recognize private, loopback, and link-local IPv4 addresses and understand where each one works.

Some IPv4 ranges are reserved for special jobs. You'll meet three of them constantly, and recognizing them on sight tells you a lot about a network before you run a single diagnostic.

Private ranges

Private IPv4 ranges are `10.0.0.0/8`, `172.16.0.0/12`, and `192.168.0.0/16`. They can be reused inside separate private networks and are not globally routed on the public Internet. Your home

network and your office network can both use `192.168.1.0/24` without conflict, because those addresses never appear on the public Internet directly.

Check what you have right now:

```
ip -4 address show
# inet 192.168.1.20/24 ... scope global dynamic wlan0
```

An address starting with `192.168.`, `10.`, or `172.16` through `172.31` means you're on a private network, and something — almost always NAT on your router — stands between you and the Internet.

Loopback

The `127.0.0.0/8` block is loopback. Traffic to it remains inside the local host and never touches any network hardware. `127.0.0.1` is the address you will see most often in local development.

```
ping -c 2 127.0.0.1
```

```
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.041 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.039 ms
```

Those sub-0.1ms times are the giveaway: no cable, no radio, just the kernel talking to itself. If `ping 127.0.0.1` fails, your machine's network stack itself is broken — an extremely rare and serious condition.

Link-local

The `169.254.0.0/16` range is IPv4 link-local. A host may choose one of these addresses itself when normal configuration is unavailable. It works only on the local link and no router will forward it.

These addresses are not interchangeable. Seeing `169.254.x.x` when you expected DHCP is often evidence of a configuration problem:

```
ip -4 address show eth0
# inet 169.254.83.107/16 scope link eth0
```

Translation: ”I asked for an address and nobody answered, so I made one up.” The machine isn't broken — the DHCP conversation failed. Check the cable, the Wi-Fi association, or the DHCP server before touching anything else.

My advice is to memorize these ranges cold. Reading `10.0.4.7` and thinking ”private, probably a cloud or office network” or reading `169.254.12.9` and thinking ”DHCP failed” turns raw output into an explanation instantly.

Read an IPv6 address and prefix

Read compressed IPv6 notation, recognize a prefix, and distinguish global, link-local, and loopback scope.

An IPv6 address contains 128 bits and is written in hexadecimal groups of 16 bits, separated by colons. Full addresses are long, so the notation has two shortcuts: leading zeros in a group can be dropped, and one run of zero groups can be compressed with `::`. That makes `2001:db8::20` a complete address — it expands to `2001:0db8:0000:0000:0000:0000:0000:0020`.

The `::` trick is allowed once per address. If it appeared twice, you couldn't tell how many zero groups each one hides.

IPv6 interfaces also use prefix lengths, and they work exactly like IPv4 CIDR: the prefix names the network, the rest identifies the interface. A `/64` leaves 64 bits for the interface identifier and is the usual subnet size for normal LANs. You'll rarely see anything else on an end-user network.

Read your own addresses

```
ip -6 address show
```

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP>
    inet6 2a01:e0a:4f2:1:8c3a:11ff:fe2b:940c/64 scope global dynamic
    inet6 fe80::8c3a:11ff:fe2b:940c/64 scope link
```

On macOS, `ifconfig en0 | grep inet6` shows the same thing. Notice you have more than one IPv6 address, and that's normal. The `scope` labels tell you what each is for.

Three scopes to recognize

Addresses beginning with `fe80::/10` are link-local and always have local-link scope. Every IPv6 interface gives itself one automatically. It works only on the directly attached link and is never routed anywhere.

`::1` is loopback, the IPv6 equivalent of `127.0.0.1`:

```
ping6 -c 1 ::1
# 64 bytes from ::1: icmp_seq=1 ttl=64 time=0.045 ms
```

Global unicast addresses — in practice, addresses starting with `2` or `3` — can be routed beyond the local network. The `2a01:...` address above is what remote servers see when this machine connects over IPv6.

The zone identifier

When a link-local destination could exist on more than one interface, commands may require a **zone identifier** such as `fe80::1%en0`. The reason: `fe80::` addresses exist per link, so `fe80::1` on your Wi-Fi and `fe80::1` on your Ethernet are different machines. The `%en0` suffix says which interface you mean.

```
ping6 -c 1 fe80::1%en0
# 64 bytes from fe80::1%en0: icmp_seq=1 ttl=64 time=2.1 ms
```

Without the zone, the same command typically fails with an `Invalid argument` style error, because the system can't pick an interface on its own. If an IPv6 command fails with an odd error on a `fe80::` address, a missing zone identifier is the first thing to check.

Check your understanding: IP addressing

Check IPv4 prefixes, subnet sizes, private and special ranges, IPv6 notation, prefixes, and address scope.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Configuration and routing

Learn how hosts get configuration, choose routes, cross routers, and receive ICMP feedback.

Get network configuration

Understand how a host obtains addresses, prefixes, routers, and DNS server information.

A usable host configuration normally includes four pieces: an address, a prefix, routes, and DNS server information. The exact source can be static configuration or an automatic protocol, but a host needs all four to be genuinely online.

How IPv4 gets configured: DHCP

DHCP commonly supplies IPv4 configuration through a discover, offer, request, and acknowledgment exchange. Your machine broadcasts "anyone have an address for me?", a DHCP server offers one, your machine formally requests it, and the server acknowledges. A **lease** gives the configuration a limited lifetime, after which the host must renew it.

On macOS you can read the actual lease your machine received:

```
ipconfig getpacket en0
```

```
yiaddr = 192.168.1.20
subnet_mask (ip): 255.255.255.0
router (ip_mult): {192.168.1.1}
domain_name_server (ip_mult): {192.168.1.1}
lease_time (uint32): 0x15180
```

There are the four pieces in one packet: your address (`yiaddr`), the prefix (as a subnet mask), the router, and the DNS server. The lease time `0x15180` is 86400 seconds — one day. On Linux, `journalctl -u NetworkManager | grep -i dhcp` or your distribution's equivalent shows the same negotiation in logs.

How IPv6 gets configured: Router Advertisements

IPv6 hosts can learn prefixes and default routers from Router Advertisements. Stateless Address Autoconfiguration (SLAAC) lets a host form its own address for an advertised prefix — no server keeps a list of who has what.

Check the results:

```
ip -6 address show eth0
# inet6 2a01:e0a:4f2:1:8c3a:11ff:fe2b:940c/64 scope global dynamic
ip -6 route | grep default
# default via fe80::9e53:22ff:feaa:41e dev eth0 proto ra
```

The `proto ra` tag on the route means it was learned from a Router Advertisement, not typed in by anyone.

Verify the pieces agree

Configuration succeeded only when the pieces agree. An address without a suitable route may reach the local link but nowhere else. A working IP path without DNS can reach addresses but not resolve names.

A quick three-part check:

```
ip address        # do I have an address and prefix?
ip route          # do I have a default route?
dig example.com   # can I resolve names?
```

When someone says "the network is down", run these three. Address present but no default route: local misconfiguration or DHCP handed out something incomplete. Address and route fine but `dig` times out: it's DNS, not the network. Each failure pattern names its own culprit.

Read the routing table

Read destination prefixes, next hops, and interfaces, then identify the most specific matching route.

A host uses its routing table to choose where an IP packet goes next. A route includes a destination prefix and an outgoing interface, and may include a next-hop router.

The most specific matching prefix wins. A route for `192.0.2.0/24` is preferred over a default route for `0.0.0.0/0` when both match the destination.

Inspect routes on Linux or macOS with:

```
ip route
netstat -rn
```

An on-link route sends directly to the destination after neighbor discovery. A gateway route sends the frame to a router. The default route is only the fallback when no more specific route matches.

Inspect the routes your machine will really use:

```
ip route
ip route get 1.1.1.1
ip route get 192.168.1.1
```

The first command shows the table. The next commands ask the kernel to choose a path for one destination. Record the selected interface, gateway, and source address. If your system uses route -n or netstat -rn instead, find the same four facts rather than memorizing one operating-system command.

Follow routers and hop limits

See what a router changes when forwarding a packet and how TTL or Hop Limit prevents endless loops.

A router receives a frame, removes the link header, examines the IP destination, chooses a route, and sends the packet inside a new frame for the next link. It repeats this for every packet, at every hop, all the way to the destination network.

The link-layer addresses change at each routed hop: every new frame carries the MAC addresses of that specific link. The source and destination IP addresses normally remain the same end to end, except when a device deliberately translates them, as NAT does.

The loop-prevention counter

Routing tables are written by humans and protocols, and both make mistakes. If router A sends packets to router B, and B sends them back to A, a packet could circulate forever, and enough of them would melt the network.

The fix is a counter in every IP header. IPv4 calls its loop-prevention field **Time to Live**, while IPv6 calls it **Hop Limit**. Each router reduces the value by one. If it reaches zero, the router discards the packet and normally sends an ICMP "Time Exceeded" error back to the source.

You can trigger this on purpose. Send a ping that's only allowed one hop:

```
ping -c 1 -t 1 1.1.1.1        # Linux: -t sets the TTL
```

```
From 192.168.1.1 icmp_seq=1 Time to live exceeded
```

On macOS the flag is `-m`:

```
ping -c 1 -m 1 1.1.1.1
```

The reply doesn't come from `1.1.1.1`. It comes from `192.168.1.1` — your router, the first hop — telling you it discarded your packet because the TTL ran out there. Raise the limit to 2 and the discard message comes from the next router instead.

Count the hops on a real path

A normal reply also reveals distance. Ping a server and look at the `ttl` in the response:

```
ping -c 1 1.1.1.1
# 64 bytes from 1.1.1.1: icmp_seq=1 ttl=57 time=9.8 ms
```

The server sent that reply with a starting TTL, commonly 64. It arrived with 57, so the return path crossed 7 routers.

This mechanism prevents a routing loop from circulating a packet forever. It also gives traceroute a way to reveal successive routers along a path — traceroute is nothing more than the `-t 1`, `-t 2`, `-t 3` trick performed systematically, collecting the address of whoever sends each Time Exceeded error. You just did one round of it by hand.

Use ICMP, ping, and traceroute

Use ICMP feedback without treating ping or traceroute as a complete test of application health.

ICMP carries control and error information for IP. It's not a protocol applications use to move data; it's how the network itself reports what happened. Echo Request and Echo Reply messages power `ping`. Time Exceeded messages help `traceroute` reveal hops.

```
ping 192.0.2.1
traceroute 203.0.113.10
```

Reading ping output

Try a real destination and let it run a few rounds:

```
ping -c 4 1.1.1.1
```

```
64 bytes from 1.1.1.1: icmp_seq=1 ttl=57 time=9.6 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=57 time=10.2 ms
64 bytes from 1.1.1.1: icmp_seq=3 ttl=57 time=9.8 ms
64 bytes from 1.1.1.1: icmp_seq=4 ttl=57 time=48.1 ms

4 packets transmitted, 4 received, 0% packet loss
```

Three fields matter. `icmp_seq` numbers the probes — gaps mean lost packets. `time` is the round trip; steady values mean a healthy path, and a jump like that last `48.1 ms` means queuing or congestion somewhere. The summary line gives you the loss percentage, the single most useful number for "is this path reliable?"

What ping does and doesn't prove

A successful ping shows that a particular ICMP exchange worked. It does not prove that a TCP application port is open — a server can answer ping perfectly while its web server is crashed.

A failed ping does not prove the host is down either, because a firewall may filter echo messages. Plenty of production servers deliberately ignore ping. Absence of a reply is weak evidence; treat it as "ICMP is filtered or the host is down", not as a verdict.

Reading traceroute output

```
traceroute -n 1.1.1.1
```

```
 1  192.168.1.1     2.0 ms   1.8 ms   1.9 ms
 2  100.64.12.1     8.9 ms   9.1 ms   8.7 ms
 3  * * *
 4  141.101.67.40   10.3 ms  10.1 ms  10.4 ms
```

Each line is a router that returned a Time Exceeded error, with three timing probes. The `-n` flag skips reverse DNS so the output is faster and cleaner.

Line 3 is the classic trap: `* * *` means that router didn't answer traceroute probes, not that the path is broken there. Traffic clearly continued, because hop 4 answered. Traceroute output can contain missing hops or changing routes — paths can even differ between packets.

Treat it as path evidence, not a perfect map. Combine it with routes, port tests, and application-specific checks before drawing conclusions.

Check your understanding: configuration and routing

Check DHCP, IPv6 autoconfiguration, route selection, default routes, router forwarding, hop limits, ping, and traceroute.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

TCP, UDP, ports, and sockets

Connect application processes through reliable streams or independent datagrams.

Choose TCP or UDP

Compare TCP streams with UDP datagrams and choose transport behavior based on application needs.

TCP gives applications a reliable, ordered byte stream between two endpoints. It detects loss, retransmits data, puts bytes back in order, and controls how quickly data is sent. You write bytes

in, the other side reads the same bytes out, and TCP handles everything that can go wrong in between.

UDP sends independent datagrams. It preserves message boundaries — one send is one message — but does not itself guarantee delivery, ordering, duplicate removal, or congestion response. What you get is an address, a port, and a fire-and-forget message.

Feel the difference with nc

You can experience both with `netcat`. Open two terminals. In the first, listen on TCP:

```
nc -l 8080
```

In the second, connect and type:

```
nc localhost 8080
hello over tcp
```

The connection exists before any data flows — kill the listener and the client notices immediately. Now repeat with UDP, adding `-u` to both sides:

```
nc -u -l 8080     # terminal 1
nc -u localhost 8080     # terminal 2
```

Messages still arrive, but stop the listener and the sender keeps typing happily into the void. Nothing tells it the other side is gone. That's the UDP contract: no connection, no delivery confirmation, no feedback.

What each one is used for

You use TCP every day: HTTP, SSH, databases — anywhere every byte must arrive, in order. DNS queries typically ride UDP because a query is one small message, and if the answer doesn't come back, asking again is cheaper than maintaining a connection. DHCP also uses UDP while a machine is still discovering its network configuration.

Video calls and games use UDP because a late packet is worthless; better to drop it and move on than stall the stream waiting for a retransmission. HTTP/3 runs over QUIC, which uses UDP as its base and implements reliability, encryption, and congestion control above it.

The choice is about behavior, not speed

Neither transport makes an application correct. An application using UDP may add acknowledgments and retries — at which point it has rebuilt part of TCP, hopefully for a good reason. An application using TCP must still define its own message boundaries inside the byte stream, because TCP deliberately doesn't preserve them.

Choose from the behavior the application needs. TCP is common when every byte must arrive in order. UDP is useful when independent messages, low overhead, multicast, or application-controlled recovery matter.

Be suspicious of ”UDP is faster” as a blanket claim. UDP has less overhead per packet, but TCP's perceived slowness usually comes from doing work your application would otherwise have to do itself. Skipping the work is only faster when you genuinely didn't need it.

Ports and sockets

Use port numbers and socket endpoints to identify the communicating processes behind an IP connection.

An IP address identifies an interface, but several applications can share that address. Your laptop runs a browser, a mail client, and a dozen background services, all using the same IP at once. TCP and UDP **port numbers** help deliver traffic to the correct process.

A **socket** endpoint combines an address, transport protocol, and port. A TCP connection is commonly identified by four values: source address, source port, destination address, and destination port. That four-tuple is why thousands of browser tabs can talk to the same web server without their data getting mixed up — each connection differs in at least one value.

See it live

Start a tiny web server and inspect it:

```
python3 -m http.server 8000 &
ss -lntp | grep 8000
```

```
LISTEN  0  5  0.0.0.0:8000  0.0.0.0:*  users:(("python3",pid=41285,fd=3))
```

The server is listening: address `0.0.0.0` (all interfaces), TCP port 8000, owned by process 41285. On macOS, use `lsof -nP -iTCP:8000` for the same answer.

Now connect to it from another terminal and look at the established connection:

```
curl -s localhost:8000 > /dev/null &
ss -tnp | grep 8000
# ESTAB  127.0.0.1:8000  127.0.0.1:52814
```

There's the four-tuple in action. Servers listen on known or configured ports — here, 8000. Clients usually choose temporary ports: `52814` was picked by the kernel from the ephemeral range and will be different next time. A web server might listen on TCP port 443 while each browser connection uses a different client port.

Stop the test server with `kill %1` when you're done.

”Is the port open?” is the wrong question

A port is not a physical hole and it is not globally open or closed. It matters together with an address, protocol, interface binding, listening process, and firewall path.

A frequent real-world bug: a service binds to `127.0.0.1:8000` instead of `0.0.0.0:8000`. From the machine itself everything works. From any other machine, connections are refused — nothing is listening on the externally reachable address. The `ss -lntp` output makes this visible instantly: check the local address column, not just the port number.

So when something can't connect, don't ask ”is the port open?” Ask: which process is listening, on which address, for which protocol, and does the path from the client actually reach it?

Follow a TCP connection

Follow connection setup, sequence numbers, acknowledgments, retransmission, flow control, and closure.

TCP begins a normal connection with a three-way handshake: SYN, SYN-ACK, and ACK. The client asks to talk, the server agrees, and the client confirms. The exchange establishes state and initial sequence numbers on both endpoints.

You can watch a handshake happen. In one terminal, capture packets for one connection:

```
sudo tcpdump -ni any -c 3 "port 443 and host 1.1.1.1"
```

In another, trigger it with `curl https://1.1.1.1/ -so /dev/null`. The capture shows:

```
IP 192.168.1.20.55102 > 1.1.1.1.443: Flags [S], seq 3402118443
IP 1.1.1.1.443 > 192.168.1.20.55102: Flags [S.], ack 3402118444
IP 192.168.1.20.55102 > 1.1.1.1.443: Flags [.], ack 1
```

`[S]` is the SYN, `[S.]` is the SYN-ACK, and the final `[.]` with an ack is the closing step of the handshake. Note the server acknowledged `3402118444` — exactly one more than the client's starting sequence number.

Sequence numbers and reliability

Sequence numbers describe positions in the byte stream. Acknowledgments tell the sender which bytes arrived. Missing acknowledgments can trigger retransmission — the sender's timer expires, and it sends the unacknowledged bytes again. This is the machinery that turns unreliable IP packets into a reliable stream.

Your kernel exposes live statistics for its connections:

```
ss -ti dst 1.1.1.1
```

```
ESTAB 0 0 192.168.1.20:55102 1.1.1.1:443
    cubic rtt:9.7/4.3 cwnd:10 bytes_acked:517 retrans:0/0
```

`rtt` is the measured round-trip time. `retrans:0/0` means nothing needed retransmitting. On a bad Wi-Fi link or congested path you'll see that counter climb — the connection still works, but every retransmission costs at least one round trip of waiting.

Flow control, congestion control, and closing

Flow control protects the receiver from too much queued data, while congestion control protects the network path — the `cwnd` (congestion window) value above is the sender pacing itself. These mechanisms affect speed without changing the application protocol.

A graceful close uses FIN and ACK exchanges: each side says ”I'm done sending” and confirms the other's goodbye. A reset, or RST, ends the connection immediately with no goodbye. Seeing RST where you expected data usually means no process was listening, a firewall rejected the connection, or a server gave up on you.

A successful handshake still does not guarantee a valid application response. TCP delivered the bytes; whether the application on top sends something sensible is a separate question, and a separate layer to debug.

Send a UDP datagram

Understand one UDP message, its ports and checksum, and the responsibilities left to the application.

A UDP datagram contains source and destination ports, a length, a checksum, and application data. One send produces one datagram, and a receive returns one complete datagram when the buffer is large enough.

UDP has no handshake. A sender can transmit without first proving that the destination application exists. An ICMP error may come back, or there may be no response at all.

DNS often uses UDP for ordinary queries. Real-time applications also use UDP when late data can be less useful than missing data. These applications define any needed retries, ordering, and rate control themselves.

"UDP is faster" is too simple. It offers fewer built-in services; whether that helps depends on the protocol designed above it and the network conditions.

Check your understanding: transport and ports

Check TCP streams, UDP datagrams, ports, socket endpoints, handshakes, acknowledgments, closure, and application responsibilities.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

NAT, firewalls, and packet size

Understand translated addresses, stateful filtering, MTU, fragmentation, and path boundaries.

Understand NAT and port translation

Follow a private IPv4 connection through address and port translation without confusing NAT with a firewall.

Network Address Translation, or NAT, rewrites address information as packets cross a boundary. It exists because IPv4 ran out of addresses: your provider gives your home one public address, but your home contains a dozen devices. Home routers commonly translate many private IPv4 clients to one public IPv4 address.

You can see both sides of your own NAT in ten seconds:

```
ip -4 address show wlan0 | grep inet
# inet 192.168.1.20/24 ...

curl https://ifconfig.me
# 93.41.227.140
```

Your machine believes it is `192.168.1.20`. The rest of the Internet sees `93.41.227.140`. Those are different because your router rewrote the source address of every packet on the way out. Everyone in your house gets the same answer from `ifconfig.me`.

Port translation keeps connections apart

If five devices share one public address, how do replies find the right device? Port translation keeps the connections distinct. The router records a mapping between an internal address and port and an external address and port, then reverses the mapping for reply traffic:

```
inside                      outside
192.168.1.20:55102   <->    93.41.227.140:40311
192.168.1.31:49220   <->    93.41.227.140:40312
```

When a reply arrives at `93.41.227.140:40311`, the router looks up the table, rewrites the destination back to `192.168.1.20:55102`, and forwards it inward. The server at the far end never learns your private address existed.

Consequences you'll run into

This state explains why a new inbound connection usually has no destination mapping unless you configure port forwarding. Someone connecting to `93.41.227.140:8080` from outside hits a router with no table entry for that port — there's nowhere to send the packet. Port forwarding is you manually adding the missing row: "port 8080 always goes to `192.168.1.20`."

It also explains why idle mappings can expire. The table isn't infinite, so mappings for quiet connections get dropped. Long-lived idle connections — SSH sessions, database connections — die mysteriously after minutes of silence for exactly this reason. Keepalive packets exist to refresh the mapping before it expires.

NAT is not a firewall

NAT changes addresses; a firewall enforces policy. They often run on the same device, and unsolicited inbound traffic does fail by default behind NAT, which feels like protection. But NAT is not a substitute for explicit firewall rules or host security. A forwarded port, a UPnP request from any device on your LAN, or a compromised machine inside the network all bypass that accidental protection. Treat NAT as address plumbing and keep real security controls in place.

Firewalls and connection state

Read a firewall decision in terms of direction, protocol, addresses, ports, and existing connection state.

A firewall permits or drops traffic according to policy. Rules may consider direction, interface, IP protocol, source and destination addresses, ports, and connection state. Reading a firewall decision means checking each of those dimensions against the packet in question.

The most important concept is **state**. A stateful firewall tracks connections. It can allow reply packets for a connection that began from the trusted side while rejecting unrelated new inbound traffic. That's how your laptop browses the web freely while accepting no unsolicited connections: outbound requests create state, and replies match it. This tracking is separate from TCP state maintained by the endpoints — the firewall keeps its own table.

Look at a real policy

On Ubuntu and similar systems, `ufw` shows a readable policy:

```
sudo ufw status verbose
```

```
Status: active
Default: deny (incoming), allow (outgoing)

To         Action      From
--         ------      ----
```

```
22/tcp     ALLOW IN    Anywhere
```

Read it as: everything outgoing is allowed, everything incoming is denied, except new connections to TCP port 22. Replies to outgoing connections are permitted by connection tracking, which is why the browsing works without any explicit ”allow” rule for it.

On macOS the application firewall reports its state differently:

```
/usr/libexec/ApplicationFirewall/socketfilterfw --getglobalstate
# Firewall is enabled. (State = 1)
```

Firewalls stack

Firewalls exist at several boundaries: on the host, home router, cloud network, or service edge. A packet from the Internet to a cloud VM might cross a provider security group, a subnet ACL, and the VM's own `ufw` — three separate policies. One allowed layer does not override a deny at another layer. The packet gets through only if every layer says yes.

This is why ”but I opened the port!” so often fails: you opened it in one place, and a different layer is still dropping the traffic.

Diagnose without weakening anything

Do not diagnose by disabling every firewall. Turning protection off to see if a problem disappears leaves you exposed and teaches you almost nothing about which rule mattered.

Instead: identify the exact flow — source, destination, protocol, port, direction. Inspect the relevant rule and log at each layer; most firewalls can log dropped packets, and one log line names the offender precisely. Make the smallest authorized change, such as one allow rule for one port from one source. Then retest the flow and confirm the log shows it passing.

MTU, fragmentation, and Path MTU

Understand packet-size limits, IPv4 and IPv6 fragmentation, and failures caused by broken Path MTU discovery.

A link's Maximum Transmission Unit, or **MTU**, is the largest IP packet it can carry without link-specific fragmentation. Ethernet's standard MTU is 1500 bytes. Check your own interfaces:

```
ip link show eth0
# 2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ...
```

A VPN or tunnel interface on the same machine will show something smaller, like `mtu 1420`, because the tunnel's own headers eat into the budget. Different links along one path can have different limits, and the smallest one wins — that smallest value is the **path MTU**.

Who fragments, and who doesn't

IPv4 routers may fragment some oversized packets, splitting them into pieces the next link accepts. In IPv6, routers never fragment; the source must send packets that fit, using feedback to choose a suitable size. Modern systems try to discover the path MTU in both versions, because fragmentation is slow and fragile even where it's allowed.

The discovery mechanism relies on ICMP feedback: a router that can't forward a too-large packet reports back that a packet is too large for the next link. You can probe a path yourself by sending pings that refuse fragmentation:

```
ping -c 1 -M do -s 1472 1.1.1.1     # Linux: -M do sets Don't Fragment
# 64 bytes from 1.1.1.1: icmp_seq=1 ttl=57 time=10.1 ms

ping -c 1 -M do -s 1473 1.1.1.1
# ping: local error: message too long, mtu=1500
```

1472 bytes of payload plus 28 bytes of ICMP and IP headers is exactly 1500 — it fits. One byte more fails. On macOS the equivalent is `ping -D -s 1472 1.1.1.1`. Walk the size down until it succeeds and you've measured the path MTU by hand. On a VPN, expect the boundary near 1392 rather than 1472.

The classic failure: PMTUD black holes

Blocking that ICMP feedback can create a path where small messages work but larger transfers stall. This produces one of networking's most confusing symptoms:

```
- ping works
- curl of a small page works
- large downloads or uploads hang forever
- git push freezes mid-transfer
```

Small packets fit everywhere. Large packets get dropped at the narrow link, and the "too big" message never reaches the sender because a firewall somewhere eats ICMP. The sender retransmits the same oversized packet forever.

When that symptom appears, inspect tunnels, VPN overhead, interface MTUs, and ICMP filtering before blaming the application. The `ping -M do` probe above is your measuring tool: if the working size is well below what your interface MTU suggests, something in the path is narrower than your system believes.

Separate address, route, port, and protocol

Classify a connection failure before changing configuration at the wrong networking layer.

Four questions narrow many failures quickly: Did the name resolve to the intended address? Is there a route to that address? Is the correct transport port reachable? Does the application speak the expected protocol?

Each question can fail independently. An IP address can be correct while the route is wrong. A route can work while a firewall blocks TCP port 443. TCP can connect while the server sends an invalid HTTP response. Fixing the wrong one wastes time and often breaks something that was fine.

Error messages already classify the failure

The good news: most tools tell you which question failed, if you read the error instead of skimming it.

```
Could not resolve host              -> address (DNS)
No route to host                    -> route
Connection refused                  -> port (reached the host; nothing listening)
Connection timed out                -> route or a silent firewall
certificate verify failed           -> protocol (TLS, above transport)
502 Bad Gateway                     -> protocol (HTTP reached an application)
```

`Connection refused` is oddly reassuring: the address was right, the route worked, and the destination actively answered "no process here". You're one layer from success. A timeout is murkier — packets vanished, either because routing is broken or a firewall drops silently.

Run one deliberate failure so you recognize the shape:

```
curl --connect-timeout 3 https://flaviocopes.com:9999/
# curl: (28) Failed to connect ... after 3002 ms: Timeout was reached
```

Name resolved, route existed, but port 9999 answered nothing. Address: fine. Route: fine. Port: failed. Protocol: never got a chance. That's a classification, and it tells you where to look next — a firewall or the server's listening configuration, not DNS.

Layers above transport

TLS and authentication add more boundaries above the transport connection. A certificate error is not evidence that IP routing failed; it means the conversation reached a later stage. The same logic applies to a `401 Unauthorized` or a `500`: those are application answers, delivered over a perfectly working network.

The order matters. A failure at one stage makes all later stages unreachable, so the first failing stage is the only one worth investigating.

Write down the first failing stage

Write down the first stage that fails, in one sentence: "DNS returns the right address, TCP to port 443 times out." This keeps a precise transport failure from becoming an uncontrolled rewrite of DNS, routing, and firewall configuration.

The people who fix network problems fast aren't running more commands than you. They classify first, then aim. Changing configuration before classifying is how a one-line firewall fix turns into an evening of self-inflicted outages.

Check your understanding: network boundaries

Check NAT mappings, port translation, firewall state, MTU, fragmentation, ICMP feedback, and layered failure classification.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Diagnose the complete path

Inspect local state, test each layer, capture packets, and explain a failure with evidence.

Inspect local network state

Inspect interfaces, addresses, routes, neighbors, DNS, and listening sockets before changing anything.

Begin a diagnosis by recording the current state. You want facts about interfaces, addresses, routes, neighbors, name resolution, and listening processes — before you change anything, and before the situation changes on its own.

On Linux, four commands cover it:

```
ip address
ip route
ip neighbor
ss -lntup
```

Here's what to look for in each. `ip address` should show your main interface `UP` with an `inet` line carrying a sensible address and prefix:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    inet 192.168.1.20/24 brd 192.168.1.255 scope global dynamic
```

No `inet` line, or a `169.254.x.x` address, means configuration failed — you can stop looking further out.

`ip route` must include a `default via ...` line, or only local destinations are reachable. `ip neighbor` shows whether your router's entry is `REACHABLE` or stuck at `FAILED` — a failed neighbor entry means frames aren't even crossing your own link.

`ss -lntup` lists listening sockets with their owning processes:

```
tcp LISTEN 0 4096 127.0.0.1:5432 users:(("postgres",pid=812,fd=6))
tcp LISTEN 0 511  0.0.0.0:80    users:(("nginx",pid=1401,fd=8))
```

Read the address column carefully. That PostgreSQL bound to `127.0.0.1` is invisible from other machines; nginx on `0.0.0.0` accepts from anywhere. A service "not responding" remotely while bound only to loopback is one of the most common findings this command produces.

The macOS equivalents

On macOS, start with `ifconfig`, `netstat -rn`, `arp -an`, `ndp -an`, and `lsof -nP -i`. They map one-to-one: interfaces, routes, IPv4 neighbors, IPv6 neighbors, and sockets with processes. Use `dig` on either system to inspect DNS answers:

```
dig +short flaviocopes.com
# 104.21.4.157
# 172.67.132.90
```

Facts first, changes later

Run only the commands you need and preserve their output — paste it into a note or redirect it to a file. Diagnosis is comparison: state now versus state when it worked, or this machine versus a healthy one. Output you didn't save can't be compared.

A missing address, unexpected default route, incomplete neighbor, or service bound only to loopback may already explain the symptom. My advice is to resist the itch to restart things until you've collected all four views. A restart destroys the evidence, and when the problem comes back next week, you'll be starting from zero.

Test one layer at a time

Build a short sequence of tests that isolates local service, DNS, route, transport, TLS, and application behavior.

Start close to the failing program and move outward. If you control the server, first confirm that

the process is running and listening on the expected address, protocol, and port — no network test means anything if the service itself is down.

Then work through the layers with one tool per question:

```
dig example.com
nc -vz example.com 443
curl -v https://example.com/
```

What each test proves

Resolve the name with `dig` and check two things in the answer:

```
dig +short example.com
# 93.184.215.14
```

Did an address come back at all, and is it the address you expected? A stale DNS record pointing at a decommissioned server passes every other test on the wrong machine.

Test the exact TCP port with `nc -vz host port`:

```
nc -vz example.com 443
# Connection to example.com port 443 [tcp/https] succeeded!
```

`succeeded!` means the TCP handshake completed: route works, port reachable, something is listening. `Connection refused` means you reached the host but nothing listens there. A hang means packets are vanishing — routing or a silent firewall.

For HTTP, use `curl -v` so DNS, connection, TLS, and HTTP events are visible separately:

```
curl -v https://example.com/ -o /dev/null
```

```
* Connected to example.com (93.184.215.14) port 443
* SSL connection using TLSv1.3
> GET / HTTP/2
< HTTP/2 200
```

Each `*` line is a layer succeeding in real time. Wherever the output stops is your failing layer — after `Connected` but before the TLS line means a certificate or TLS problem, not a network problem.

Inspect the selected route with `ip route get <address>` if the connection test hangs; it tells you which interface and gateway your machine actually chose.

One test, one question

A test should answer one question. Ping is not a port test, and a TCP connection is not an HTTP health check. When a test bundles several layers — like opening the URL in a browser — a failure tells you almost nothing about where the problem lives.

Stop when you find the first failed boundary. Everything past it is guaranteed broken for the same reason, so testing further just accumulates noise. The classic mistake runs the sequence backwards: the browser fails, so you clear caches, restart the router, and edit configuration — before ever learning whether DNS, TCP, TLS, or HTTP was the failing step. Three commands, run in order, replace an hour of guessing.

Capture and read packets

Use an authorized packet capture to confirm DNS exchanges, handshakes, retransmissions, resets, and application bytes.

A packet capture shows what crossed an interface. Capture only traffic you own or are authorized to inspect, because packets may contain private application data.

A narrow filter keeps the evidence readable:

```
sudo tcpdump -ni any host 192.0.2.20 and port 443
```

Look for a DNS query and reply, TCP SYN and SYN-ACK, repeated retransmissions, resets, TLS alerts, or application responses. Missing packets are evidence too when compared with captures from both endpoints.

Wireshark can decode the same layers visually. Remember that capture offloading and encryption can affect what you see. Correlate packets with application and firewall logs.

Capture a small, bounded sample instead of recording everything:

```
sudo tcpdump -ni any -c 20 'icmp or port 53'
```

In another terminal, run one ping and one DNS lookup. Match each request with its reply. Notice the source, destination, protocol, and timing. Do not capture production traffic casually: packets can contain private addresses, names, tokens, and unencrypted application data.

Diagnose one complete connection

Trace a browser request from name resolution to the application response and record the first failed boundary.

For a final exercise, choose a web service you control. Write down its hostname, expected addresses, transport protocol, port, and URL before testing.

Resolve the hostname. Inspect the route to the chosen address. On the local link, inspect the neighbor entry for the destination or default router. Test the TCP port, then inspect TLS and HTTP with `curl -v`.

If needed, capture the flow and match packets to each stage: DNS exchange, local frame delivery, routed IP packets, TCP handshake, TLS handshake, HTTP request, and response.

Your result should be one precise sentence, such as: "DNS and routing work, but the server sends a TCP reset because no process listens on port 443." That sentence is more actionable than "the Internet is down."

Check your understanding: diagnose the path

Check inspection commands, focused tests, packet captures, authorization, evidence, and complete connection diagnosis.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/networking-

foundations/.