Networking Fundamentals

A Complete 11-Chapter Course on OSI, TCP/IP, Addressing & Diagnostics

Topics covered:

The OSI & TCP/IP models · IP addressing & subnetting

Routing basics & the default gateway · TCP vs. UDP

Ports & sockets · DNS resolution in depth

Real diagnostic tools: ping, traceroute, ss, tcpdump

Capstone: tracing a real request end to end

Exercises: 33 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · supplies the foundation ssh1/vpn1/rdp1/https1/host1 all assumed

Table of Contents

CHAPTER 1 OF 11

Why Networking Fundamentals Matter

Networking Fundamentals

Chapter 1 · Why Networking Fundamentals Matter

A handful of courses on this site have quietly assumed something no course ever actually taught: that you already know what an IP address, a port, or a routing table is. This chapter names that gap directly — and previews the journey the rest of this course is going to explain, piece by piece.

The Gap This Course Fills

A few concrete examples, not a vague complaint:

- `ssh1`'s own "How SSH Works" chapter talks about connecting to a "host" over a "port" without ever explaining what a port actually is, or how a connection gets established in the first place.
- `https1` walks through the TLS handshake in detail, but that handshake rides on top of a plainer, older handshake — TCP's own — that's never explained anywhere on this site.
- `vpn1` talks about routing traffic through a tunnel, assuming the reader already knows what routing does under normal, non-VPN conditions.
- `host1` covers managing DNS records at a registrar and at Cloudflare, without ever explaining what actually happens, step by step, when a browser resolves a domain name into an address.
- `cloud1-5`'s own Networking Fundamentals chapter covers VPCs, subnets, and security groups, assuming subnetting and routing were already understood coming in.

None of those courses were wrong to skip this — they each had their own scope. This course exists specifically to retroactively supply the foundation every one of them quietly assumed.

A First Preview — What Actually Happens When You Load a Webpage

A short, deliberately light preview — every term here gets its own full chapter later, so don't worry about fully understanding any of it yet. Typing a URL and hitting enter sets off: a **DNS lookup** (turning a name into an address), **routing** across a chain of networks your computer has never directly touched, a **TCP handshake** (a short back-and-forth that establishes a reliable connection), and finally an actual **HTTP request and response** carrying the page itself.

Chapters 2 through 10 unpack every one of those terms in full. The capstone, `net1-11`, walks through this exact same journey again — this time with nothing left unexplained.

The Two Models You'll See Everywhere — OSI and TCP/IP

Networking is almost always explained in "layers" — a way of separating concerns so a web developer never has to think about electrical signals on a cable, and an electrical engineer never has to think about HTTP headers. Two models describe those layers: **OSI**, a seven-layer conceptual reference model still used as shared vocabulary, and **TCP/IP**, the leaner four-or-five-layer model that describes what's actually running on real networks. `net1-2` covers OSI; `net1-3` covers TCP/IP and maps the two together.

Why "It Just Works" Usually Hides Real Complexity

This isn't just academic. When something breaks — a site won't load, an SSH connection times out, a VPN refuses to connect — knowing which *layer* the problem most likely lives in (DNS? routing? the application itself?) is what separates a systematic diagnosis from random guessing. `net1-10`'s own diagnostic tools, and `cloud2-2`'s own connectivity-diagnosis material, both lean directly on the foundation this course builds.

TERM USED UNEXPLAINED ELSEWHERE	WHAT IT ROUGHLY MEANS	FULLY EXPLAINED IN
IP address / subnet	A network-level address, and the range of addresses it belongs to	net1-4, net1-5
Routing / default gateway	How a packet finds its way to a destination on another network	net1-6
TCP handshake	The setup exchange that establishes a reliable connection	net1-7
Port	The specific "door" on a machine a connection is aimed at	net1-8
DNS resolution	Turning a domain name into an IP address	net1-9

THIS CHAPTER IS A MAP, NOT A TEST

Nothing here needs to be memorized. Every term introduced in this preview gets its own full, dedicated explanation in an upcoming chapter — this one exists purely to show where the course is headed.

THE INTERNET IS A NETWORK OF NETWORKS

There is no single, unified "the internet" network — the name is literally short for *inter*-network. It's an enormous collection of independently-run networks, all agreeing to speak the same protocols so they can pass traffic between each other. Almost every concept in this course — addressing, subnetting, routing — exists specifically to solve the problem of connecting many independent networks together, not to describe one single network.

Hands-On Exercises

EXERCISE 1

Pick three specific instances from this chapter's own "The Gap This Course Fills" section where a site course used networking terminology without explaining it. For each one, name which upcoming net1 chapter will explain it.

View solution

EXERCISE 2

Using this chapter's own "layers separate concerns" idea, explain why a web developer normally never needs to think about electrical signals on a cable, but a network engineer troubleshooting a broken connection sometimes does.

View solution

EXERCISE 3

A user reports "the website won't load." Using this chapter's own preview of the full request journey (DNS, routing, TCP handshake, HTTP), list at least three fundamentally different points in that journey where the actual failure could be occurring.

View solution

CHAPTER 1 QUICK REFERENCE

- This course retroactively supplies the networking foundation ssh1/https1/vpn1/host1/cloud1 all quietly assumed
- A full request journey: DNS lookup → routing → TCP handshake → HTTP request/response
- **OSI** — seven-layer conceptual reference model (net1-2)
- **TCP/IP** — the leaner model describing what actually runs (net1-3)
- Knowing which layer a problem lives in turns troubleshooting into diagnosis, not guessing
- The internet is a network of independently-run networks, not one single network
- The capstone (net1-11) retraces this exact journey once every piece has been explained

CHAPTER 2 OF 11

The OSI Model — Seven Layers

Networking Fundamentals

Chapter 2 · The OSI Model — Seven Layers

`net1-1` promised a full explanation of "layers" — this chapter is the first, most widely-referenced version of that idea: the seven-layer OSI model. It's a conceptual reference model, not something running anywhere directly, and understanding that distinction is half the point of this chapter.

The Seven Layers, Bottom to Top

1. **Physical** — the actual medium carrying raw bits: cables, radio waves, voltage levels.
2. **Data Link** — framing raw bits into discrete frames, MAC addresses, the layer switches operate at.
3. **Network** — logical addressing (IP addresses) and routing between networks, the layer routers operate at.
4. **Transport** — end-to-end reliability between two endpoints; TCP and UDP, and ports, live here.
5. **Session** — establishing, managing, and tearing down a communication session between applications.
6. **Presentation** — data format translation, encryption, and compression.
7. **Application** — the actual protocols applications use directly: HTTP, DNS, SSH.

A classic mnemonic, bottom to top: *Please Do Not Throw Sausage Pizza Away* — Physical, Data Link, Network, Transport, Session, Presentation, Application.

Why "Nothing Implements It in Pure Form"

OSI was originally designed in the late 1970s and 1980s as an actual protocol suite, not just a teaching reference — but TCP/IP won out as the real-world implementation the internet actually runs on (the full story is `net1-3`'s own). In practice, real protocols rarely map cleanly onto exactly one OSI layer. TLS, for example, handles concerns that sit somewhere between the session and presentation layers, but in everyday conversation it's usually just described as sitting "between transport and application" — nobody insists on pinning it to layer 5 or layer 6 specifically.

Despite this mismatch, OSI remains genuinely valuable as shared, precise vocabulary. "Layer 2 switch," "Layer 3 routing," "Layer 4 load balancer," "Layer 7 firewall" are all real, commonly used industry terms that only make sense with this model as the shared reference point — that's the actual reason it's still taught, even though nothing runs it directly.

A Worked Example — A Simple File Download

Downloading a file over Wi-Fi, layer by layer: **Physical** is the actual radio signal between your laptop and the router. **Data Link** is your network card framing data and addressing it to your router's MAC address. **Network** is the destination server's IP address. **Transport** is the TCP connection on port 443, making sure every byte of the file arrives, in order, with nothing missing. **Session/Presentation** is the TLS encryption wrapping the whole exchange. **Application** is the actual HTTP `GET` request asking for the file in the first place.

Where This Course's Own Terminology Will Land

A mental map for the rest of this course: `net1-4` / `net1-5` (IP addressing, subnetting) and `net1-6` (routing) all live at **Layer 3**. `net1-7` (TCP vs. UDP) and `net1-8` (ports) both live at **Layer 4**. `net1-9` (DNS) is an **application-layer (Layer 7)** protocol, despite being one of the very first things that happens in any request.

LAYER	NAME	REAL-WORLD EXAMPLE
7	Application	HTTP, DNS, SSH
6	Presentation	TLS encryption, data compression
5	Session	Establishing/tearing down a session
4	Transport	TCP, UDP, ports
3	Network	IP addresses, routers
2	Data Link	MAC addresses, switches
1	Physical	Cables, radio waves, voltage

THE NUMBERING IS REAL INDUSTRY SHORTHAND

When you hear "a Layer 3 switch" or "a Layer 7 load balancer" in real conversation, that's this exact numbering being used directly — not a loose analogy. Knowing the table above makes that shorthand immediately readable.

DON'T FORCE EVERY PROTOCOL INTO EXACTLY ONE LAYER

Session and Presentation (layers 5 and 6) in particular are often folded into "the application" in real TCP/IP-based systems, since protocols like HTTP and TLS handle those concerns themselves rather than as cleanly separate layers. Trying to rigidly assign every real-world protocol to exactly one OSI layer is a common source of confusion — OSI is a reference model for reasoning about networking, not a literal blueprint every protocol was built to match.

Hands-On Exercises

EXERCISE 1

A colleague says "our load balancer operates at Layer 4, and we're thinking about switching to a Layer 7 load balancer instead." Using this chapter's own table, explain the practical difference between what each type of load balancer can "see" and act on.

View solution

EXERCISE 2

Using this chapter's own worked file-download example, explain what would happen at the Physical layer specifically if you unplugged your ethernet cable mid-download, and why that's different from a failure at the Transport layer.

View solution

EXERCISE 3

A student insists that TLS must be assigned to exactly one OSI layer, either 5 or 6, and refuses to move on until it's settled. Using this chapter's own warn-box, explain why this is the wrong question to be stuck on.

View solution

CHAPTER 2 QUICK REFERENCE

- Seven layers, bottom to top: Physical, Data Link, Network, Transport, Session, Presentation, Application
- Mnemonic: Please Do Not Throw Sausage Pizza Away
- OSI is a reference model and shared vocabulary — nothing runs it in pure form
- "Layer 2 switch," "Layer 3 routing," "Layer 4 load balancer," "Layer 7 firewall" are real, precise industry terms
- Session and Presentation are often folded into "the application" in real TCP/IP-based systems
- net1-4/5/6 → Layer 3, net1-7/8 → Layer 4, net1-9 (DNS) → Layer 7

CHAPTER 3 OF 11

The TCP/IP Model — What's Actually Running

Networking Fundamentals

Chapter 3 · The TCP/IP Model — What's Actually Running

`net1-2` covered OSI, the conceptual seven-layer reference model. This chapter is the model actually running underneath every network you've ever used — leaner, four layers instead of seven, and, unlike OSI, genuinely implemented rather than just referenced.

Why TCP/IP Won

OSI was designed as an actual protocol suite by an international standards committee, meant to be deployed, not just referenced. TCP/IP came out of DARPA-funded ARPANET research and was deployed and battle-tested in the real world starting in the 1970s and 80s. By the time OSI's own protocol suite was anywhere near finished, TCP/IP was already running the growing internet — a real-world deployment winning out over a more theoretically complete design that arrived too late to matter.

The Four (or Five) TCP/IP Layers

The classic model (RFC 1122) has four layers: **Link**, **Internet**, **Transport**, **Application**. Some material splits Link into separate Physical and Data Link layers, giving five — matching OSI's own bottom two directly. Both conventions are genuinely in use; neither is more "correct" than the other.

Mapped onto OSI: TCP/IP's **Application** layer absorbs OSI's Application, Presentation, and Session (layers 5–7). **Transport** matches OSI's Transport (layer 4) directly. **Internet** matches OSI's Network (layer 3) directly. **Link** absorbs OSI's Data Link and Physical (layers 1–2).

What Actually Runs at Each Layer

- **Link** — Ethernet, Wi-Fi (802.11): framing and physical transmission.
- **Internet** — IP (IPv4/IPv6): addressing and routing; ICMP (what `ping` uses) lives here too.
- **Transport** — TCP and UDP, `net1-7`'s own subject.
- **Application** — HTTP, HTTPS, DNS, SSH, SMTP: every protocol a user-facing application actually speaks.

Revisiting the File Download Through TCP/IP's Lens

Same download from `net1-2`, now through TCP/IP's four layers instead of OSI's seven: **Link** (the Wi-Fi signal and frame addressing, collapsing OSI's layers 1–2), **Internet** (the destination IP address — unchanged from OSI's own layer 3), **Transport** (the TCP connection on port 443 — unchanged from OSI's own layer 4), **Application** (the HTTP `GET` request, with TLS handled inside this same layer rather than as a separate one).

That last point is the concrete resolution of `net1-2`'s own warn-box: TCP/IP doesn't have separate homes for "session" and "presentation" concerns the way OSI describes them — protocols like TLS and HTTP simply handle those responsibilities themselves, folded into one Application layer.

Encapsulation — How Data Actually Gets Wrapped

A genuinely new mechanism this chapter adds: as data moves down the stack, each layer **encapsulates** the data from the layer above by wrapping it in its own header. An HTTP request becomes a TCP segment (a header carrying port information gets added), which becomes an IP packet (a header carrying address information gets added), which becomes an Ethernet frame (a header carrying MAC address information gets added). At the receiving end, this happens in reverse — each layer strips off its own header and passes the remaining payload up to the layer above.

```
HTTP request
  → wrapped in a TCP segment  (adds: source/destination port)
    → wrapped in an IP packet   (adds: source/destination IP address)
      → wrapped in an Ethernet frame (adds: source/destination MAC address)
```

This is the actual technical mechanism behind layering — not just a conceptual convenience, but literally how the bytes on the wire are structured.

TCP/IP LAYER	MATCHING OSI LAYER(S)	REAL PROTOCOL EXAMPLE
Application	7, 6, 5 (Application, Presentation, Session)	HTTP, DNS, SSH
Transport	4 (Transport)	TCP, UDP
Internet	3 (Network)	IP, ICMP
Link	2, 1 (Data Link, Physical)	Ethernet, Wi-Fi

LAYERS 1-4 NUMBERING IS USED INTERCHANGEABLY IN PRACTICE

When someone in a firewall or security context says "Layer 3 and Layer 4 information," they mean IP addresses (layer 3) and ports (layer 4) — TCP/IP-flavored shorthand borrowing OSI's own numbering, since those four layers genuinely map cleanly between the two models.

YOU WON'T FIND A "SESSION LAYER HEADER" IN A REAL PACKET

TCP/IP's own encapsulation only really produces four distinct header types — link, internet, transport, and application. Looking for a separate "session layer" header while examining a captured packet (`net1-10`'s own `tcpdump` chapter) will come up empty — that concern, where it exists at all, is handled inside the application-layer data itself.

Hands-On Exercises

EXERCISE 1

Explain, in terms of real-world deployment rather than technical superiority, why TCP/IP became the internet's actual protocol suite instead of OSI's own.

View solution

EXERCISE 2

Using this chapter's own encapsulation example, list the four headers that would exist around an HTTP request by the time it's actually transmitted, and name what information each one adds.

View solution

EXERCISE 3

A colleague examining a packet capture asks where the "session layer header" is, expecting to find one distinct from the application data. Using this chapter's own warn-box, explain why they won't find one.

View solution

CHAPTER 3 QUICK REFERENCE

- TCP/IP: four layers — Link, Internet, Transport, Application (sometimes five, splitting Link)
- TCP/IP's Application layer absorbs OSI's Application, Presentation, and Session (layers 5-7)
- TCP/IP's Internet and Transport layers map directly onto OSI's Network (3) and Transport (4)
- TCP/IP won out through real-world deployment, not because OSI was technically inferior
- **Encapsulation** — each layer wraps the layer above's data in its own header on the way down, strips it on the way up
- Only four real header types exist on the wire: link, internet, transport, application

CHAPTER 4 OF 11

IP Addressing

Networking Fundamentals

Chapter 4 · IP Addressing

`net1-3` named IP as the protocol running at the Internet layer, handling addressing and routing. This chapter is that addressing, in full — the foundation `net1-5`'s own subnetting chapter builds directly on top of.

IPv4 Structure and Notation

An IPv4 address is 32 bits, written as four "octets" separated by dots — **dotted-decimal notation**, e.g. `192.168.1.1`. Each octet is 8 bits, giving a decimal range of 0–255 per octet. Underneath, `192` is really the binary `11000000` — the human-readable decimal form is purely a convenience over the actual 32-bit binary value.

32 bits means a total address space of 2^{32} — roughly 4.3 billion possible addresses. That number matters directly later in this chapter, and again in `net1-5`: it's finite, and it ran out.

Public vs. Private IP Ranges

Three ranges (RFC 1918) are reserved specifically for private networks and are never routable on the public internet: `10.0.0.0/8`, `172.16.0.0/12`, and `192.168.0.0/16`. Every other address is potentially a **public** address — globally unique, routable across the internet, assigned down through IANA and regional registries to ISPs and organizations.

A home router makes this concrete: it has one public IP address facing the internet, while every device behind it — laptops, phones, smart TVs — gets a private `192.168.x.x` (or similar) address that only means something inside that one household's network.

NAT — Network Address Translation

NAT is the mechanism that lets many devices with private addresses share one public address. When a device behind the router sends a packet outward, the router rewrites the packet's source IP and port to its own public IP, and tracks the mapping. When a response comes back, the router rewrites the destination back to the correct private IP and port and forwards it inward.

A direct, practical consequence: an unsolicited incoming connection generally *can't* reach a device behind a home router without extra configuration (port forwarding) — NAT only tracks mappings for connections initiated from the inside. This is incidentally security-relevant, though it isn't a security feature by design (see the warn-box below), and it's exactly why `vpn1` 's own material needs special handling for NAT traversal when establishing a VPN tunnel.

A First Look at IPv6

IPv4's roughly 4.3 billion addresses are not enough for a world with vastly more connected devices than that — regional registries genuinely ran out of new IPv4 address blocks to hand out years ago, and NAT (above) has functioned as the major real-world stopgap ever since, not just an incidental convenience.

IPv6 addresses are 128 bits, written as eight groups of four hex digits separated by colons: `2001:0db8:85a3:0000:0000:8a2e:0370:7334`. Shorthand rules exist to make this less unwieldy — leading zeros within a group can be dropped, and one run of consecutive all-zero groups can be collapsed to `::`.

Honestly: adoption remains uneven. Most home and office networks still run IPv4, usually behind NAT, as the primary protocol as of this course's writing. IPv6 adoption is genuinely growing, and "dual-stack" setups — running both IPv4 and IPv6 simultaneously during the long transition — are common.

A Worked Example — Reading a Real Address

Take `192.168.1.42`. Conceptually, part of this address identifies "which network" and part identifies "which specific device on that network" — but exactly where that split happens can't be determined from the address alone. It requires a **subnet mask** alongside it, which is exactly why `net1-5` exists as its own dedicated chapter.

	ADDRESS SIZE	NOTATION	EXHAUSTION / ADOPTION
IPv4	32 bits (~4.3 billion addresses)	Dotted-decimal, e.g. 192.168.1.1	Exhausted at the registry level; still the dominant protocol, usually behind NAT
IPv6	128 bits	Colon-separated hex, e.g. 2001:0db8::1	Effectively inexhaustible; adoption real but uneven, often dual-stacked

SEEING YOUR OWN PRIVATE AND PUBLIC IPS

`ip addr` on Linux or `ipconfig` on Windows shows a device's private IP address. Visiting a "what is my IP" site from the same device shows the public IP NAT is translating that traffic to — two genuinely different addresses for the same device, at the same moment.

"NOT ROUTABLE" IS NOT THE SAME AS "SECURE"

It's easy to assume private IP ranges are inherently safer simply because they can't be reached directly from the internet without NAT or port forwarding. That's true as far as it goes, but it isn't a security control by design — an attacker already on the local network, or malware already running on a device behind the router, has full, direct access regardless of the address being private. Don't confuse "not internet-routable" with "protected."

Hands-On Exercises

EXERCISE 1

Explain why a laptop's own private IP address (e.g. 192.168.1.42) and its public-facing IP address (as seen by a "what is my IP" website) are different, and what mechanism is responsible for that difference.

View solution

EXERCISE 2

Explain why an unsolicited connection from the internet generally can't reach a laptop sitting behind a home router, without any firewall being involved at all.

View solution

EXERCISE 3

A colleague argues that a database server is "safe" from attackers because it's only reachable via a private 10.x.x.x address, never directly from the internet. Using this chapter's own warn-box, explain the flaw in that reasoning.

View solution

CHAPTER 4 QUICK REFERENCE

- IPv4: 32 bits, dotted-decimal notation, ~4.3 billion total addresses
- Private ranges (never internet-routable): 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16
- **NAT** — rewrites source IP/port on the way out, tracks the mapping, rewrites destination on the way back
- NAT is why unsolicited inbound connections generally can't reach a device behind a home router
- IPv6: 128 bits, colon-separated hex notation, effectively inexhaustible, adoption still uneven
- Determining "network part" vs. "host part" of an address requires a subnet mask — net1-5's own subject
- "Not internet-routable" is not a security control — don't confuse the two

CHAPTER 5 OF 11

Subnetting

Networking Fundamentals

Chapter 5 · Subnetting

`net1-4` closed on an unanswered question: given an address like `192.168.1.42`, where does the "network" part end and the "host" part begin? This chapter answers it in full, with real math and real worked examples.

The Subnet Mask — Splitting an Address Into Network and Host

A subnet mask is a 32-bit value, written the same way an IP address is, that marks which bits of an address are "network" bits (1s) versus "host" bits (0s). `255.255.255.0` in binary is `11111111.11111111.11111111.00000000` — the first 24 bits are network, the last 8 are host.

Applied to `192.168.1.42` with mask `255.255.255.0`: the network portion is `192.168.1.0`, and the remaining bits identify device `42` specifically on that network.

CIDR Notation — A Shorthand for the Mask

CIDR (Classless Inter-Domain Routing) notation writes the mask as a slash followed by the number of network bits: `192.168.1.0/24` means the first 24 bits are network bits — exactly equivalent to a mask of `255.255.255.0`.

The "classless" part of the name is a real historical improvement worth naming honestly: CIDR replaced an older, rigid Class A/B/C system that only allowed fixed boundaries (/8, /16, /24 and nothing in between) with flexible, arbitrary prefix lengths — letting networks be sized to what's actually needed instead of forced into one of three fixed sizes.

Calculating Usable Host Ranges

For a `/n` prefix: host bits = `32 - n`, total addresses = `2^(32-n)`, usable hosts = `2^(32-n) - 2` — subtracting the network address and the broadcast address, neither of which can be assigned to a device.

```
# 192.168.1.0/24
host bits: 32 - 24 = 8
total addresses: 2^8 = 256
usable hosts: 256 - 2 = 254
range: 192.168.1.1 - 192.168.1.254
network address: 192.168.1.0
broadcast address: 192.168.1.255

# 192.168.1.0/28
host bits: 32 - 28 = 4
total addresses: 2^4 = 16
usable hosts: 16 - 2 = 14
range: 192.168.1.1 - 192.168.1.14
network address: 192.168.1.0
broadcast address: 192.168.1.15
```

Why Subnetting Matters in Practice

Beyond the arithmetic, real reasons to subnet: organizing a network into logical, isolated segments (separate subnets for servers, employee workstations, and guest Wi-Fi), containing broadcast traffic (broadcasts only travel within a subnet, never across the whole network), and creating real access-control boundaries (a firewall or router can apply different rules per subnet). This is exactly the concept `cloud1-5`'s own VPC/subnet material assumed as background — this chapter is the foundational math that material was quietly built on.

A Worked Example — Splitting a /24 Into Four Subnets

Given `192.168.1.0/24`, split it evenly into 4 subnets for 4 departments. Borrowing 2 more bits from the host portion (24 → 26) produces exactly 4 subnets of `/26` each, 64 addresses per subnet (62 usable):

```
192.168.1.0/26    # hosts 192.168.1.1   - 192.168.1.62
192.168.1.64/26   # hosts 192.168.1.65  - 192.168.1.126
192.168.1.128/26  # hosts 192.168.1.129 - 192.168.1.190
192.168.1.192/26  # hosts 192.168.1.193 - 192.168.1.254
```

CIDR	MASK	TOTAL ADDRESSES	USABLE HOSTS
/24	255.255.255.0	256	254
/25	255.255.255.128	128	126
/26	255.255.255.192	64	62
/27	255.255.255.224	32	30

CIDR	MASK	TOTAL ADDRESSES	USABLE HOSTS
/28	255.255.255.240	16	14
/30	255.255.255.252	4	2

A QUICK MENTAL FORMULA, AND TWO EDGE CASES

Usable hosts = $2^{(\text{host bits})} - 2$. Two genuine edge cases: `/31` is commonly used for point-to-point links with no network/broadcast subtraction at all (RFC 3021, both addresses usable), and `/32` identifies a single specific host — a "network" of exactly one address.

NEVER ASSIGN THE NETWORK OR BROADCAST ADDRESS TO A DEVICE

A very common mistake: assigning `192.168.1.0` or `192.168.1.255` directly to a device on a `/24` network. Both are reserved — the network address identifies the subnet itself, and the broadcast address is used to reach every device on the subnet at once. Assigning either to a single device breaks things in ways that can be genuinely confusing to diagnose later.

Hands-On Exercises

EXERCISE 1

For the network 10.0.5.0/27, calculate: total addresses, usable host range, the network address, and the broadcast address.

View solution

EXERCISE 2

A team needs to split 172.16.0.0/24 into 8 equal-sized subnets. Using this chapter's own bit-borrowing method, determine the new CIDR prefix and list all 8 resulting subnet addresses.

View solution

EXERCISE 3

An administrator manually assigns the IP address 192.168.10.255 to a new server on a 192.168.10.0/24 network, and the server immediately has connectivity problems. Using this chapter's own warn-box, explain what went wrong.

View solution

CHAPTER 5 QUICK REFERENCE

- Subnet mask: marks network bits (1) vs. host bits (0), 32 bits total, same notation as an IP address
- CIDR: `/n` shorthand for a mask with n network bits — e.g. /24 = 255.255.255.0
- Total addresses = 2^(32−n); usable hosts = 2^(32−n) − 2
- The network address (all host bits 0) and broadcast address (all host bits 1) are never assignable
- Splitting a network into equal subnets = borrowing bits from the host portion
- /31 (point-to-point) and /32 (single host) are genuine edge cases to the usual −2 rule
- Subnetting creates real broadcast-containment and access-control boundaries — cloud1-5's own VPC/subnet material assumed this math

CHAPTER 6 OF 11

Routing Basics

Networking Fundamentals

Chapter 6 · Routing Basics

`net1-5` covered how to tell whether an address is on your own subnet. This chapter covers what happens the moment it isn't — how a packet actually finds its way to a network your device has never directly touched.

The Core Problem Routing Solves

A device already knows how to deliver a packet locally — same subnet, direct Data Link-layer delivery (`net1-2` / `net1-3`). Routing exists specifically for the moment the destination *isn't* local: something has to decide where that packet goes next.

The Default Gateway — What It Actually Does

When a device wants to reach an address outside its own subnet, it doesn't guess a path — it hands the packet to its **default gateway**, almost always a router, which takes on the job of figuring out the next step.

Concretely: the device compares the destination IP against its own subnet mask (`net1-5`'s own math). If the destination falls outside the local subnet, the packet gets addressed at the Data Link layer to the default gateway's own MAC address — but the destination IP address itself, at the Network layer, stays completely unchanged. The gateway is only the next physical hop, not the final destination.

This is the formal answer to the term left unexplained since `net1-1`, and it's exactly what `net1-4`'s own home router example was doing all along — the router is the default gateway for every device behind it.

Routing Tables — How a Router Decides Where Next

A routing table is a list of rules of the form "to reach network X, send packets to next-hop Y." Every router consults its own table for every packet it forwards.

```
Destination        Next hop         Interface
192.168.1.0/24     directly connected  eth0
192.168.2.0/24     10.0.0.2           eth1
0.0.0.0/0          10.0.0.1           eth1   # the default route — "everything else"
```

When more than one route could match a destination, the most specific one wins — this is called **longest prefix match**: the entry with the largest CIDR number (the most specific subnet) is preferred over a broader, more general route.

Static vs. Dynamic Routing

Static routing means someone manually configures each route ahead of time — simple and predictable, but it doesn't adapt automatically to network changes or failures. Reasonable for a small network with few, stable paths.

Dynamic routing means routers exchange information with each other using routing protocols (RIP, OSPF, and BGP are the real, named examples — none of them deep-dived here, that's genuinely out of this course's scope) to automatically learn the network's topology and reroute around failures. BGP specifically deserves a brief, honest mention: it's the literal protocol that routes traffic across the entire internet, between the independently-run networks (Autonomous Systems) that make up the "network of networks" `net1-1` opened with.

A Worked Example — Tracing a Packet Across Two Subnets

Device A (`192.168.1.42/24`) wants to reach Device B (`192.168.2.10/24`) — a genuinely different subnet. Device A checks whether `192.168.2.10` falls within its own `192.168.1.0/24` subnet — it doesn't, so the packet is sent to Device A's default gateway. The gateway checks its own routing table, finds a route to `192.168.2.0/24`, and forwards the packet out the correct interface toward Device B's subnet, where local delivery takes over again.

	SETUP EFFORT	ADAPTS TO FAILURE?	TYPICAL USE CASE
Static	Manual, per route	No — requires manual reconfiguration	Small, stable networks with few paths
Dynamic	Automatic once configured	Yes — routers exchange updates	Large or changing networks; the internet itself (BGP)

SEEING YOUR OWN DEVICE'S ROUTING TABLE

`ip route` on Linux or `route print` on Windows shows a device's real routing table, including its own default gateway entry — the exact mechanism this chapter describes, made visible. `net1-10` revisits this alongside other diagnostic tools.

THE DEFAULT GATEWAY IS A SINGLE POINT OF FAILURE FOR EVERYTHING OFF-SUBNET

A genuinely common, confusing real-world scenario: "I can reach my neighbor's device on the same subnet, but not google.com." If the default gateway is misconfigured or down, every off-subnet destination becomes unreachable —

while local, same-subnet traffic keeps working perfectly fine, since it never needed the gateway in the first place. The symptom (some things work, others don't) can look mysterious until the gateway itself is checked directly.

Hands-On Exercises

EXERCISE 1

A device on 10.0.1.0/24 wants to reach 10.0.9.5. Using this chapter's own step-by-step logic, explain exactly what the device does before the packet ever leaves it.

View solution

EXERCISE 2

A router's routing table contains both a route to 192.168.2.0/24 and a default route (0.0.0.0/0). A packet destined for 192.168.2.55 arrives. Using this chapter's own longest-prefix-match rule, explain which route wins and why.

View solution

EXERCISE 3

A user reports they can access every device on their own home network but can't reach any website. Using this chapter's own warn-box, explain the most likely cause and why local devices still work fine.

View solution

CHAPTER 6 QUICK REFERENCE

- Routing only matters once a destination is off the local subnet — local delivery needs no routing at all
- **Default gateway** — the router a device hands off-subnet traffic to; only the next hop, never the final destination
- **Routing table** — "to reach network X, send to next-hop Y," consulted for every packet
- **Longest prefix match** — the most specific matching route (largest CIDR number) always wins
- **Static routing** — manual, predictable, doesn't adapt to failure
- **Dynamic routing** — routers exchange updates automatically; BGP specifically routes the entire internet between Autonomous Systems
- A dead or misconfigured default gateway breaks everything off-subnet while local traffic keeps working — a classic, confusing symptom pattern

CHAPTER 7 OF 11

TCP vs. UDP

Networking Fundamentals

Chapter 7 · TCP vs. UDP

`net1-6` got a packet to the right network. This chapter is about the two dominant ways applications actually structure communication once it arrives — TCP and UDP, the Transport-layer protocols named back in `net1-3`.

Connection-Oriented vs. Connectionless

TCP (Transmission Control Protocol) is connection-oriented — it establishes a formal connection before any real data moves, and guarantees delivery, ordering, and error-checking. **UDP** (User Datagram Protocol) is connectionless — it just sends packets ("datagrams") with no setup, no delivery guarantee, and no ordering guarantee, in exchange for dramatically lower overhead.

TCP is a phone call — both sides confirm the line is open before speaking, and know immediately if it drops. UDP is a stack of mailed postcards — fast and cheap to send, with no confirmation they ever arrive, or in what order.

The Three-Way Handshake, In Full

This is the exact handshake `web-sockets1-2` mentioned only briefly — here it is in full, for the first time on this site.

1. **SYN** — the client sends a SYN (synchronize) packet carrying an initial sequence number.
2. **SYN-ACK** — the server responds with its own SYN, plus an ACK acknowledging the client's sequence number.
3. **ACK** — the client sends a final ACK, acknowledging the server's own sequence number.

After these three steps, both sides have a working, mutually-acknowledged connection and can begin exchanging real data. This is also the direct answer to the gap `net1-1` named in `https1`: TLS's own handshake happens *after* this one, layered on top of an already-established TCP connection — two separate handshakes, not one.

Sequencing and Reliability — What TCP Actually Guarantees

Every byte TCP sends carries a sequence number. The receiver acknowledges data by sequence number; if an expected acknowledgment doesn't arrive in time, TCP retransmits. This is the actual mechanism behind three separate guarantees: data arrives (via retransmission), arrives in the correct order (sequence numbers let out-of-order data be reordered), and isn't duplicated (sequence numbers also identify duplicates directly).

None of this exists in UDP at all. A lost, dropped, or out-of-order UDP datagram stays lost, dropped, or out-of-order — UDP itself does nothing about it. Fixing it, if it matters, is left entirely to the application.

When Each Is Actually Used

TCP: web browsing (HTTP/HTTPS), email, file transfer, SSH — anything where complete, correct, in-order delivery genuinely matters more than raw speed. **UDP**: DNS queries (`net1-9`'s own upcoming subject — a quick request/response where retry logic, if needed, happens at the application level), video/voice calls (a late frame is often worse than a missing one — better to skip it than stall waiting for a retransmission), online gaming (same logic), and DHCP.

DNS specifically mostly runs over UDP for speed, falling back to TCP only for unusually large responses — `net1-9` picks this up directly.

A Real Application Choosing Both — WebSockets

`web-sockets1-2`'s own material is worth revisiting here explicitly: a WebSocket connection starts life as an ordinary HTTP request — over TCP, naturally — that gets "upgraded" into a persistent, full-duplex TCP connection. WebSockets are built entirely on TCP, not UDP, specifically because ordered, reliable delivery genuinely matters for most real-time application data (chat messages, live updates), even though "real-time" might suggest reaching for UDP instead.

	CONNECTION SETUP	RELIABILITY / ORDERING	OVERHEAD	TYPICAL USE
TCP	Three-way handshake required	Guaranteed — retransmission, sequencing	Higher	HTTP/HTTPS, SSH, email, file transfer
UDP	None — send immediately	None — application's own responsibility	Lower	DNS, voice/video, gaming, DHCP

WATCHING A REAL HANDSHAKE HAPPEN

`ss` or `netstat` (`net1-10`'s own tools) show active TCP connections and their exact state — `SYN_SENT`, `ESTABLISHED`, and others — a direct, visible trace of where a real connection sits inside, or stuck inside, this exact handshake process.

A DROPPED HANDSHAKE LOOKS EXACTLY LIKE "THE SERVER IS SLOW"

A firewall or NAT device silently dropping any part of the three-way handshake — most often the SYN-ACK on the way back — causes a connection attempt to hang or time out in a way that's genuinely indistinguishable, from the user's side, from a server that's simply slow to respond. The real problem is that the handshake never completed at all, not that a working connection is just taking its time — a common, confusing source of misdiagnosed connectivity issues.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own handshake steps, why https1's TLS handshake can't begin before the TCP handshake finishes.

View solution

EXERCISE 2

A team is building a live video call feature and is deciding between TCP and UDP for the actual video stream. Using this chapter's own reasoning, recommend one and explain why the other's guarantees would actually hurt the experience.

View solution

EXERCISE 3

A user reports a web request "hangs forever" rather than failing outright. Using this chapter's own warn-box, explain a likely cause that has nothing to do with the server actually being slow.

View solution

CHAPTER 7 QUICK REFERENCE

- **TCP** — connection-oriented, guarantees delivery/ordering/no-duplication via sequence numbers and retransmission
- **UDP** — connectionless, no guarantees at all, much lower overhead
- Three-way handshake: SYN → SYN-ACK → ACK, establishing a connection before real data moves
- TLS's own handshake (https1) happens after, and on top of, the TCP handshake — two separate steps
- TCP for correctness-critical traffic; UDP for latency-critical traffic where stale data beats missing data
- WebSockets run entirely over TCP, upgraded from an ordinary HTTP request
- A dropped SYN-ACK causes a hang indistinguishable from "the server is slow" — a real, common misdiagnosis

CHAPTER 8 OF 11

Ports & Sockets

Networking Fundamentals

Chapter 8 · Ports & Sockets

`net1-7` covered TCP and UDP as the two ways to structure communication. This chapter answers a question left open since `net1-1`: how does one single IP address handle many different applications, and many simultaneous connections, all at once? Ports are the mechanism.

What a Port Actually Is

A port is a 16-bit number, 0 through 65535, that — combined with an IP address — identifies a specific endpoint for a specific connection on a device. It isn't a physical thing; it's a purely logical addressing concept the operating system's own networking stack manages.

The real addressable unit is a **socket**: IP address + port + protocol (TCP or UDP), together. `192.168.1.42:443` (TCP) is a distinct socket from `192.168.1.42:80` (TCP), which is distinct again from `192.168.1.42:53` (UDP).

Well-Known Ports — Formally Defining What's Been Used Unexplained

Ports 0–1023 are "well-known," assigned by IANA to specific standard services. This is the formal answer `net1-1` promised for terms used without explanation across several site courses:

PORT	PROTOCOL	SERVICE	USED UNEXPLAINED IN
22	TCP	SSH	ssh1
25	TCP	SMTP	—
53	TCP/UDP	DNS	host1, net1-9
80	TCP	HTTP	host1
443	TCP	HTTPS	https1

Ephemeral Ports — The Other Half of Every Connection

Every connection has two ports, not one: the well-known port on the **server** side (e.g. 443), and a temporary, dynamically assigned **ephemeral** port on the **client** side — typically somewhere in the 49152–65535 range.

This is the actual mechanism behind "how does one IP serve many simultaneous connections": a client can have many simultaneous connections to the same server on the same well-known port — many browser tabs all hitting the same website on port 443 — because each connection uses a *different* ephemeral port on the client side, making every full pairing unique even though the server-side port never changes.

The Full Picture — A Socket Pair

A single TCP connection is uniquely identified by four values together: source IP, source port, destination IP, and destination port. This four-value combination — not any single one of them — is what distinguishes one connection from every other connection happening at the same moment, even between the exact same two machines.

```
# Two browser tabs, same laptop, same website — two distinct connections
Tab 1: 192.168.1.42:51000  →  203.0.113.10:443
Tab 2: 192.168.1.42:51001  →  203.0.113.10:443
```

Identical source IP, identical destination IP, identical destination port — but two genuinely distinct connections, because the ephemeral source port differs.

Ports and Firewalls

A firewall rule like "allow inbound on port 22" is specifically about which well-known, *listening* ports are allowed to accept new incoming connections. It generally doesn't need to, and shouldn't try to, restrict ephemeral/outbound ports individually — those are dynamically assigned per-connection and would be impractical to enumerate ahead of time.

RANGE	NAME	TYPICAL USE
0–1023	Well-known	Standard services (SSH, HTTP, HTTPS, DNS)
1024–49151	Registered	Application-specific services (e.g. 3306 MySQL)
49152–65535	Ephemeral / dynamic	Temporary client-side ports, one per outgoing connection

SEEING REAL LISTENING PORTS AND THEIR OWNERS

`ss -tulpn` on Linux shows every port actively listening for connections, and which process owns it — a direct, practical view of the sockets this chapter describes. `net1-10` covers this alongside other diagnostic tools.

THE SAME PORT NUMBER CAN MEAN TWO DIFFERENT THINGS AT ONCE

TCP and UDP each maintain their own completely independent port space — DNS genuinely uses port 53 for both TCP and UDP simultaneously, and they don't conflict, because protocol is part of a socket's own identity, per this chapter's own definition. Seeing "port 53" alone doesn't tell you which protocol is meant — a genuinely common point of confusion.

Hands-On Exercises

EXERCISE 1

A laptop opens three separate SSH sessions to the same server. Using this chapter's own four-value model, explain how the server tells these three connections apart, given that the destination IP and destination port are identical for all three.

View solution

EXERCISE 2

A junior admin proposes a firewall rule blocking all outbound traffic except from a specific hardcoded list of source ports, believing this improves security. Using this chapter's own ephemeral-port material, explain why this approach doesn't make practical sense.

View solution

EXERCISE 3

A colleague says "port 53 is open" without specifying TCP or UDP, and assumes this is a complete statement. Using this chapter's own warn-box, explain what's actually still ambiguous about that claim.

View solution

CHAPTER 8 QUICK REFERENCE

- A port is a 16-bit number (0-65535); a socket is IP + port + protocol combined
- Well-known ports (0-1023): SSH=22, SMTP=25, DNS=53, HTTP=80, HTTPS=443
- Ephemeral ports (typically 49152-65535): temporary, client-side, one per outgoing connection
- A connection's real identity is the full 4-tuple: source IP, source port, destination IP, destination port
- Firewalls typically restrict inbound listening ports, not outbound ephemeral ports

- TCP and UDP maintain entirely separate port spaces — the same number can mean two different things

CHAPTER 9 OF 11

DNS In Depth

Networking Fundamentals

Chapter 9 · DNS In Depth

`host1` already covers DNS practically — adding records at a registrar, moving DNS to Cloudflare. This chapter doesn't repeat that; it goes underneath it, to the actual protocol-level mechanics `host1` never needed to explain.

What host1 Already Covered (And What This Chapter Adds)

`host1` is about managing DNS records — creating an A record, pointing a domain at a new host. This chapter assumes that part is already familiar and instead explains what actually happens, step by step, the moment a DNS query gets made — a resolution process `host1` never walked through.

The DNS Hierarchy — Root, TLD, Authoritative

DNS is a hierarchical, distributed database, not one central lookup table anywhere. Three tiers matter: **root servers** (13 logical root server clusters worldwide, knowing only where to find TLD servers), **TLD servers** (one set per top-level domain — `.com`, `.org`, `.net` — knowing where to find authoritative servers for domains under that TLD), and **authoritative servers** (holding the actual records for one specific domain — e.g. the nameservers a domain's own registrar or Cloudflare setup in `host1` points to).

The Resolution Process, Step by Step

Resolving `www.example.com` from scratch, assuming nothing is cached anywhere:

1. The application asks the operating system's own stub resolver.
2. The stub resolver asks a recursive resolver — the ISP's own DNS server, or a public one like `8.8.8.8` or `1.1.1.1`.
3. The recursive resolver asks a root server: "who handles `.com`?"
4. The root server responds with the address of a `.com` TLD server.
5. The recursive resolver asks that TLD server: "who is authoritative for `example.com`?"
6. The TLD server responds with `example.com`'s own authoritative nameservers.

7. The recursive resolver asks that authoritative nameserver directly: "what is the A record for `www.example.com`?"
8. The authoritative server responds with the actual IP address.
9. The recursive resolver caches the answer and returns it up the chain to the stub resolver, and finally to the application.

This full chain usually takes only milliseconds, and is almost always shortened dramatically by caching at multiple levels along the way — covered next.

Caching and TTL

Every DNS record carries a TTL (time to live) — how long a resolver is allowed to cache the answer before it has to ask again. This is exactly why a DNS change (updating an A record after moving hosts, per `host1`'s own material) doesn't take effect everywhere instantly — cached copies at resolvers scattered around the world keep serving the old answer until their own individual TTL expires. This is the concrete mechanism behind a genuinely common source of confusion: "I changed my DNS and nothing happened."

Record Types, Beyond A

RECORD	PURPOSE	EXAMPLE
A	Maps a name to an IPv4 address	example.com → 203.0.113.10
AAAA	Maps a name to an IPv6 address	example.com → 2001:db8::1
CNAME	Alias pointing to another name	www.example.com → example.com
MX	Identifies the mail server for a domain	example.com → mail.example.com
TXT	Arbitrary text, often for verification/SPF/DKIM	"v=spf1 include:_spf.example.com ~all"
NS	Delegates a domain to specific nameservers	example.com → ns1.registrar.com

`host1` already covers using several of these practically — this table is the formal, precise reference definition underneath that practical usage.

Why DNS Mostly Uses UDP

Revisiting `net1-7`'s own closing note directly: a typical DNS query and response is small, and UDP's complete lack of handshake overhead (no three-way handshake, per `net1-7`) makes a single query dramatically faster than establishing a full TCP connection just to ask one small question. TCP fallback happens specifically when a response is too large for a single UDP packet (historically capped at 512 bytes, extended by EDNS0) or during zone transfers between authoritative servers themselves.

TRACING RESOLUTION MANUALLY

`dig example.com` or `nslookup example.com` trace a real DNS query directly, showing which server actually answered and what TTL came back with it — `net1-10` covers both alongside the rest of this course's diagnostic toolkit.

LOWERING A TTL BEFORE A MIGRATION ONLY WORKS IF DONE EARLY ENOUGH

Setting a low TTL right before a planned migration genuinely helps minimize the caching-related propagation delay described above — but the TTL *change itself* is subject to the OLD TTL that was in effect before it. Lowering the TTL at the same time as making the actual change is too late; it needs to happen well in advance, giving the old, longer TTL time to fully expire everywhere before the real change goes out.

Hands-On Exercises

EXERCISE 1

Using this chapter's own step-by-step resolution process, explain what a root server does and does not know about www.example.com specifically.

View solution

EXERCISE 2

A site owner updates their A record to point at a new server, then reports the site is still loading from the old server 20 minutes later. Using this chapter's own TTL/caching material, explain the most likely cause.

View solution

EXERCISE 3

A team lowers their DNS record's TTL to 60 seconds on the same day they plan to migrate servers, assuming this guarantees a fast, clean cutover. Using this chapter's own warn-box, explain the flaw in their timing.

View solution

CHAPTER 9 QUICK REFERENCE

- DNS is hierarchical: root → TLD → authoritative servers, not one central database
- Full resolution: stub resolver → recursive resolver → root → TLD → authoritative → answer cached back up the chain
- **TTL** — how long a resolver may cache an answer before asking again; the real reason DNS changes propagate gradually

- A, AAAA, CNAME, MX, TXT, NS — the core record types, formally defined here beyond host1's own practical usage
- DNS mostly runs over UDP for speed (net1-7); TCP fallback for large responses or zone transfers
- Lower a TTL well before a migration, not at the same time — the change itself is bound by the old TTL

CHAPTER 10 OF 11

Diagnosing the Network

Networking Fundamentals

Chapter 10 · Diagnosing the Network

Every prior chapter built conceptual understanding. This chapter is the payoff — four real tools that make each of those concepts directly visible on a live system, and the workflow for combining them into an actual diagnosis rather than a guess.

ping — Testing Basic Reachability

`ping <host>` sends ICMP Echo Request packets and waits for an Echo Reply — a simple, direct test of reachability at the Network layer, using ICMP, which travels alongside IP itself (`net1-3`). Real output reports round-trip time (RTT) and packet loss percentage.

An honest limitation worth naming directly: ping confirms Network-layer reachability, nothing more. A "yes" from ping doesn't guarantee the actual service you care about — a web server on port 80, say — is reachable at all; many servers and firewalls block ICMP entirely while the real service works perfectly fine, and the reverse is also possible.

traceroute — Seeing the Actual Path

`traceroute <host>` (Linux/Mac) or `tracert` (Windows) reveals every router hop along the path to a destination — a direct, visible trace of `net1-6`'s own routing chapter in action.

The mechanism is genuinely clever: it sends packets with increasing TTL (time-to-live) values, and each router along the path that decrements a packet's TTL to zero sends back an ICMP "time exceeded" message, revealing itself as one hop in the chain. The real diagnostic value: seeing exactly where along a path traffic stops, or slows dramatically — directly answering the "is this a routing problem or something else" question `net1-6`'s own warn-box raised.

ss / netstat — Seeing Active Connections and Listening Ports

`ss -tulpn` on modern Linux (or the older `netstat -tulpn`) shows every listening port (`net1-8`'s own subject) and every active connection, including its exact TCP state — `ESTABLISHED`, `SYN_SENT`, and the rest of `net1-7`'s own handshake vocabulary.

Real diagnostic value: confirming whether a service is actually listening on the port expected, or whether a connection is genuinely stuck mid-handshake — directly revisiting `net1-7`'s own warn-box about a silently dropped SYN-ACK.

tcpdump — Seeing the Actual Packets

`tcpdump` captures raw packets directly off a network interface — the deepest level of visibility available, showing the real encapsulation (`net1-3`'s own headers) exactly as they exist on the wire.

```
tcpdump -i eth0 port 443
```

A captured line shows source and destination IP:port, and TCP flags — `SYN`, `ACK`, and the rest — directly tying back to `net1-7`'s own three-way handshake, now visible as literal packets instead of a diagram. This is the same underlying capability Wireshark offers through a GUI — `tcpdump` is the always-available, CLI-only version.

Putting It Together — A Real Troubleshooting Workflow

In order: **ping** first — is it reachable at all? **traceroute** next — if not, where does the path actually break down? **ss** — if this is the local machine, is the expected service even listening? **tcpdump** last — if everything above looks fine but something is still wrong, what's actually happening at the packet level?

This is exactly the layer-by-layer troubleshooting instinct `net1-1` opened this entire course with, now made concrete and tool-based rather than abstract.

Where This Connects — wdt1 and cloud2-2

`wdt1`'s own Network panel chapter shows exactly this kind of information — requests, status, timing — but from inside a browser's dev tools, scoped to a single page load. This chapter's tools work at the OS/network level, for any traffic at all, not just what one browser tab is doing. `cloud2-2`'s own "Diagnosing Connectivity Issues" chapter — working through security groups, NACLs, route tables, and DNS — assumed exactly this level of tool familiarity and protocol knowledge; this chapter is the foundation that material was quietly built on.

TOOL	TESTS	LAYER(S)
ping	Basic reachability	Network (Layer 3)
traceroute	The actual routing path	Network (Layer 3)
ss / netstat	Listening ports, connection state	Transport (Layer 4)

TOOL	TESTS	LAYER(S)
tcpdump	Raw packet contents	All layers, as captured

TCPDUMP USUALLY NEEDS ELEVATED PRIVILEGES

Raw packet capture is a privileged operation on most systems — running `tcpdump` generally requires `sudo` or root, since it can see every packet crossing an interface, not just traffic addressed to the current user's own processes.

A FAILED PING IS NOT PROOF OF AN OUTAGE

Directly building on this chapter's own opening limitation: treating a failed ping as definitive proof that a host or service is down is a common, real mistake. ICMP is frequently blocked deliberately by firewalls while the actual service being checked works completely fine — a failed ping means "ICMP didn't get a reply," nothing more specific than that.

Hands-On Exercises

EXERCISE 1

A web server responds normally to HTTPS requests in a browser, but pinging it fails completely. Using this chapter's own material, explain why this doesn't necessarily mean anything is wrong.

View solution

EXERCISE 2

A traceroute to a remote server shows the path succeeding cleanly through every hop, but the actual application still can't connect. Using this chapter's own tool descriptions, name the next tool to reach for and explain what it would help confirm.

View solution

EXERCISE 3

Using this chapter's own explanation of how traceroute works, explain why increasing TTL values, rather than a single fixed TTL, are necessary to reveal each hop individually.

View solution

CHAPTER 10 QUICK REFERENCE

- **ping** — ICMP-based reachability test; a failure doesn't prove the real service is down
- **traceroute/tracert** — reveals the actual routing path via increasing TTL values
- **ss/netstat** — listening ports and live connection states (ESTABLISHED, SYN_SENT, etc.)
- **tcpdump** — raw packet capture, the deepest level of visibility, usually needs root/sudo
- Workflow: ping → traceroute → ss → tcpdump, narrowing from "is it reachable" to "what's actually happening"
- wdt1's Network panel is the browser-scoped version of what these OS-level tools show for all traffic
- cloud2-2's own connectivity-diagnosis material assumed exactly this toolkit and protocol knowledge

CHAPTER 11 OF 11

Capstone: Tracing a Real Request End to End

Networking Fundamentals

Chapter 11 · Capstone — Tracing a Real Request End to End

`net1-1` promised this exact moment: "the capstone, `net1-11`, walks through this exact same journey again — this time with nothing left unexplained." A user types `https://www.example.com` and hits enter. Here is everything that actually happens, step by step, in order.

Step 1 — DNS Resolution

The browser and OS check their own caches first; if nothing is cached, the recursive resolver walks the full hierarchy — root server, then `.com` TLD server, then `example.com`'s own authoritative server — exactly the nine-step process from `net1-9`. The answer comes back: an A record, say `203.0.113.10`, cached going forward for however long its TTL allows.

Step 2 — Determining the Path

With a destination IP now in hand, the device checks whether `203.0.113.10` falls within its own local subnet, using the subnet-mask math from `net1-5`. It doesn't — the device is almost certainly sitting on a private address (`net1-4`), and `203.0.113.10` is a public internet address. The packet is handed to the default gateway (`net1-6`), which consults its own routing table and forwards it onward — in reality, through many more routers and ISP networks than shown here, coordinated at internet scale by dynamic routing protocols like BGP (`net1-6`'s own honest mention).

Step 3 — The TCP Handshake

Before any real data moves, the OS establishes a TCP connection to `203.0.113.10` on port 443 — HTTPS's own well-known port (`net1-8`). The three-way handshake runs in full: SYN, SYN-ACK, ACK (`net1-7`). The OS assigns a random ephemeral source port for this specific connection — say, `52341` — completing the full four-value socket pair from `net1-8`: `(local IP):52341 → 203.0.113.10:443`.

Step 4 — TLS, Layered On Top

Once the TCP connection is fully established, a genuinely *separate* TLS handshake begins over it — certificate exchange, key negotiation — building an encrypted channel on top of the already-reliable connection underneath. This is exactly the resolution `net1-7` gave to the gap `net1-1` first named in `https1`: two handshakes, strictly sequential, not one. It's also `net1-2`'s own "session/presentation" concerns made concrete, folded into the Application layer in practice per `net1-3`.

Step 5 — The HTTP Request and Response

With an encrypted, fully-established TCP connection now in place, the browser sends the actual HTTP `GET` request — an Application-layer protocol (`net1-3`) traveling over the exact socket identified in Step 3 (`net1-8`). The server processes the request and sends back the page content, riding over that same connection, using TCP's own sequencing (`net1-7`) to guarantee it arrives complete and in the correct order.

Step 6 — If Something Had Gone Wrong

At any of the five steps above, something could genuinely fail: DNS could fail to resolve, routing could break down somewhere along the path, the handshake could hang on a silently dropped SYN-ACK (`net1-7`'s own warn-box), or the HTTP exchange itself could error out. `net1-10`'s own tools — `ping`, `traceroute`, `ss`, `tcpdump` — are exactly what isolates which specific step actually failed, rather than treating "the website won't load" as one undifferentiated mystery. This directly closes the loop on `net1-1`'s own Exercise 3, which asked for exactly these three distinct failure points before any of this course existed to explain them.

Chapter Attribution Table

STEP	CHAPTER(S)
DNS resolution	net1-9
Subnet check and default gateway handoff	net1-4, net1-5, net1-6
TCP handshake and ephemeral port	net1-7, net1-8
TLS layered on top of TCP	net1-2, net1-3, net1-7
HTTP request/response over the socket	net1-3, net1-8
Diagnosing a failure at any step	net1-10

STILL OUT OF SCOPE

This course deliberately never went deep on dynamic routing protocols themselves — BGP/OSPF/RIP were named (`net1-6`) but never taught in depth. Real-world CDN and load-balancing behavior (DNS or anycast routing returning different answers depending on the requester's own location) is a genuine simplification acknowledged here, not

covered. IPv6-specific resolution and routing differences were only given a first look (`net1-4`) — this entire capstone traces the IPv4 case for simplicity. And HTTP itself — headers, caching, methods — was deliberately left uncovered, arguably web-development territory the site already addresses elsewhere.

TRY THIS EXACT JOURNEY YOURSELF

`dig example.com`, then `curl -v https://example.com` — `curl`'s `-v` flag shows the TCP connection being made, the TLS handshake, and the HTTP request/response all in one place: a live, real version of every step in this chapter.

Hands-On Exercises

EXERCISE 1

A request fails specifically at Step 3 (the TCP handshake never completes). Using net1-10's own toolkit, name which tool would most directly reveal this, and what the output would show.

View solution

EXERCISE 2

Explain why Step 4 (TLS) could not have happened before Step 3 (the TCP handshake) completed, using this chapter's own attribution table to trace back to the relevant earlier chapter's reasoning.

View solution

EXERCISE 3

A reader finishes this capstone and concludes they now fully understand how every website request on the internet works. Using this chapter's own warn-box, identify what's still missing from that conclusion.

View solution

CHAPTER 11 QUICK REFERENCE — THE FULL JOURNEY

- 1. DNS resolution — hierarchy walk, TTL caching (net1-9)
- 2. Subnet check → default gateway → routing (net1-4, net1-5, net1-6)
- 3. TCP three-way handshake + ephemeral port → full socket pair (net1-7, net1-8)
- 4. TLS handshake, layered on top of the now-established TCP connection (net1-2, net1-7)
- 5. HTTP request/response over that same socket, sequenced and reliable (net1-3, net1-8)
- 6. Any step can fail independently — net1-10's tools isolate exactly which one

- Still out of scope: BGP/OSPF internals, CDN/anycast behavior, IPv6 specifics, HTTP itself