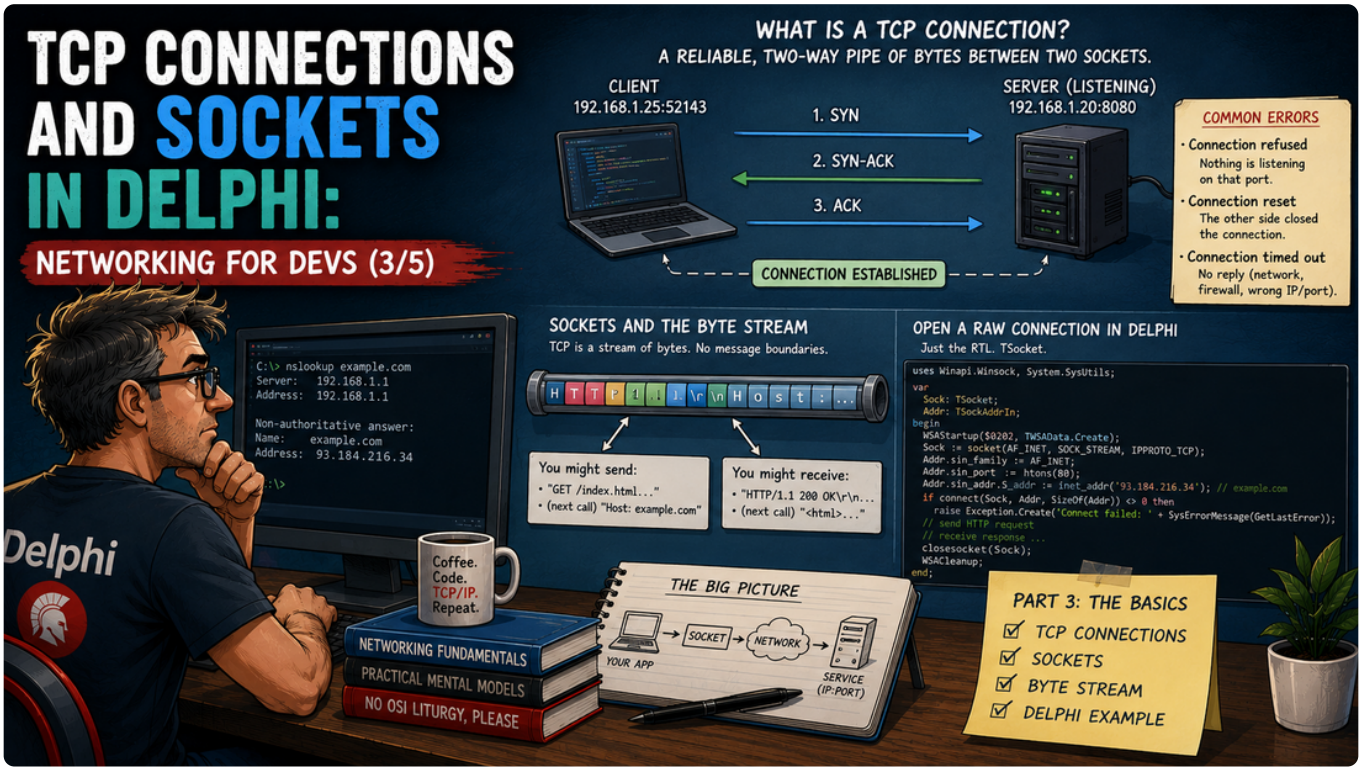
NEWS · JULY 12, 2026

TCP Connections and Sockets in Delphi: Networking for Devs (3/5)

By Dr. Holger Flick



In Part 1 we learned how to name a machine and a door on it — IP addresses and ports. In Part 2 we learned how names become addresses — DNS, the hosts file, and DHCP. So you can now say, precisely, *"the service at 192.168.1.20, port 8080."*

But here's the thing every error dialog assumes you already know and nobody ever explains: what *is* a connection? Your users see **"connection refused"**, **"connection reset"**, **"connection timed out"** — three different messages, three completely different causes — and the word doing all the work in each of them is one we've never actually defined.

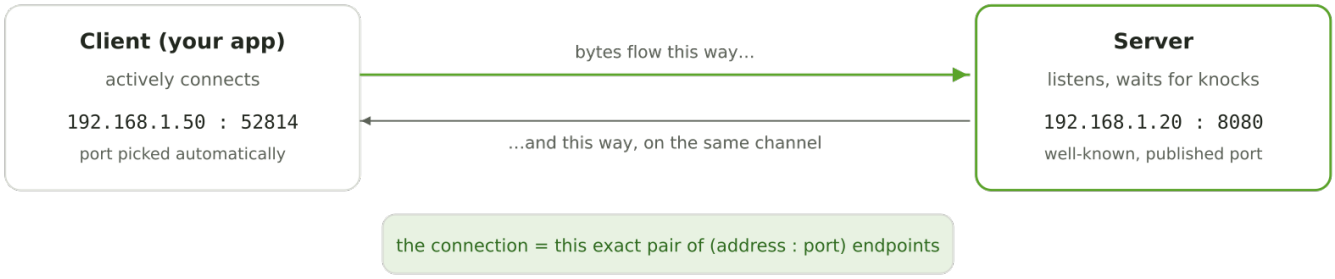
That's today's job. By the end of this post you'll know exactly what a TCP connection is, why one side *listens* while the other side *connects*, why TCP hands you a stream of bytes rather than tidy messages (the number one beginner trap), and — the payoff — you'll open a raw connection from Delphi code and speak to a real web server *by hand*, using nothing but the modern RTL. No components dropped on a form, no third-party library. Just `TSocket` and about twenty lines.

Two roles: one side listens, one side connects

Before anything travels across the wire, the two programs involved play strictly different parts — and this asymmetry is the key that unlocks everything else in this post.

A **server** is a program that tells the operating system: *"I'm interested in port 8080. If anyone knocks, wake me up."* That's called **listening**, and it's completely passive — a listening server consumes almost nothing while it waits. A **client** is the active party: it says *"connect me to 192.168.1.20, port 8080"* and the operating system goes and knocks.

When the knock is answered, something new springs into existence that wasn't there before: a **connection** — a private, two-way channel between exactly those two programs. And here is the part that makes error messages finally make sense: the connection is identified by *four* values, not two.



A connection is the pair of endpoints: client address:port ” server address:port

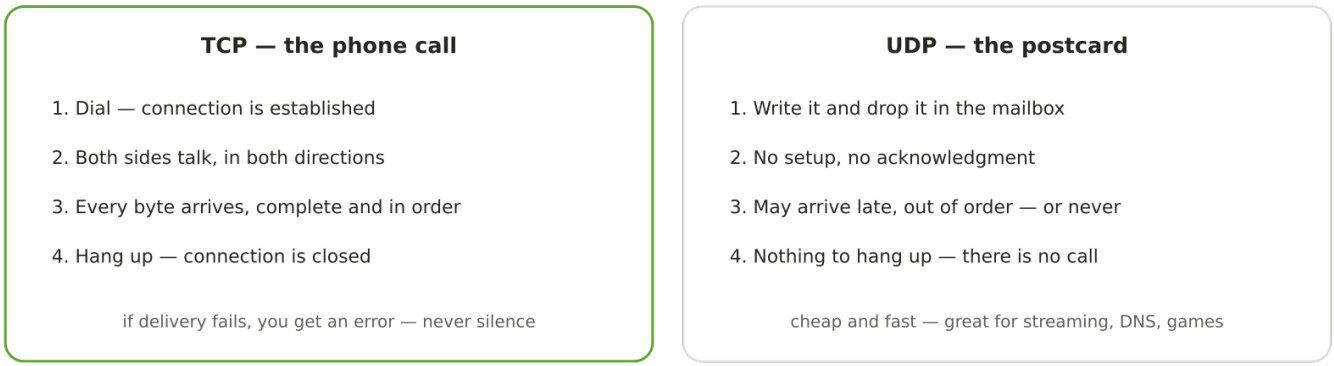
Read the diagram left to right: the server's address and port are the well-known ones — you configured them, you published them, your users type them in. The client's port, though, is one you never chose. When your app calls "connect," the operating system automatically assigns it a temporary local port — an ephemeral port, drawn from a high range and recycled when the connection closes — which is why a single machine can hold thousands of simultaneous connections to the same server: each one has a different client port, so each four-value pair is unique.

That's also why your browser can open five tabs to the same website without them interfering: five connections, five different ephemeral ports, five distinct channels.

TCP is a phone call, UDP is a postcard

The Internet actually offers two fundamentally different ways to send data, and one picture is enough to keep them straight for the rest of your career.

TCP — the Transmission Control Protocol, defined today in RFC 9293 — works like a **phone call**: you dial, the other side picks up, both of you can talk for as long as you like knowing every word arrives in order, and eventually someone hangs up. UDP — the User Datagram Protocol, RFC 768 — works like a **postcard**: you write it, drop it in the box, and hope. No call setup, no confirmation, no guarantee of arrival or ordering.



TCP is a phone call with guarantees; UDP is a postcard with hope

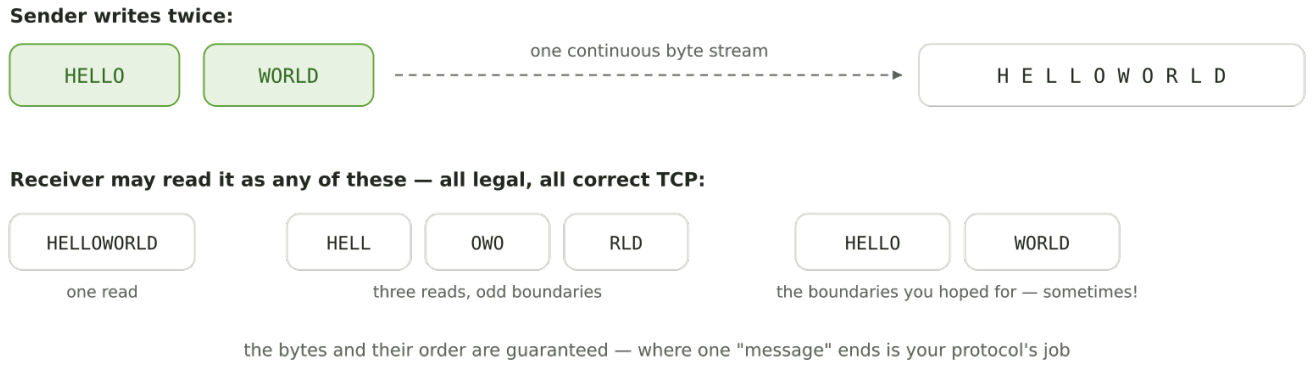
The takeaway from this picture: everything you do as a business-application developer — databases, REST calls, file transfers, web servers — rides on the left side. TCP's promise is exactly what your accounting data needs: **bytes arrive complete and in the order they were sent, or you get an error.** Never silent corruption, never a mysteriously shuffled invoice. UDP has honorable jobs (DNS queries from Part 2 typically travel by UDP, and so do video calls and games, where a late packet is worthless anyway), but we won't need it again in this series.

The "dial" step in the phone call has a famous name you'll encounter in every networking text: the three-way handshake, a quick three-packet exchange in which both sides agree the connection exists — your code never sees it, the operating system handles the whole ritual inside that one "connect" call.

The trap: TCP is a stream, not messages

Now the single most important practical fact in this post — the one that separates developers who fight mysterious intermittent bugs from developers who don't.

You might assume that if the sender transmits `"HELLO"` and then `"WORLD"`, the receiver gets two tidy packages: `"HELLO"`, then `"WORLD"`. **TCP makes no such promise.** TCP guarantees the *bytes* and their *order* — nothing more. It is one continuous stream, like water through a pipe: what you poured in comes out complete and in sequence, but the "bucketfuls" on the receiving end can be split and merged any way the network found convenient.



TCP preserves bytes and order — not the boundaries of your writes

What this diagram is really telling you: **"message" is not a TCP concept.** If your application needs messages — and it almost always does — *something on top of TCP* has to define where one message ends and the next begins. That something is called a **protocol**, and there are two classic techniques: end each message with a known delimiter (a line break, say), or send a length up front ("the next 512 bytes are one message"). This is precisely what HTTP does — header lines end with CRLF, a blank line ends the header block, and a `Content-Length` header announces how many body bytes follow. HTTP, it turns out, is just a set of framing rules for text over a TCP stream. Hold that thought; we're about to prove it.

The insidious part of this trap is that on a fast local network your writes *often do* arrive in one piece each — so the naive code works on your machine, works in the demo, and fails at the customer site under load. If you remember one technical fact from this post, make it this one.

Speaking to a real server, by hand, in Delphi

Time to make all of this concrete. Modern Delphi ships a lightweight socket class in the RTL: `TSocket` in the `System.Net.Socket` unit — a thin, cross-platform wrapper over the operating system's Berkeley sockets API, the same battle-hardened interface virtually every programming language wraps. It's been in the RTL for years and is right there in Delphi 13 Florence, the current release. No packages to install, no components to drop.

And here's the fun part. Instead of connecting to some toy echo service, we'll connect to a **real web server** on port 80 and speak raw HTTP to it — because after the last section you know exactly what HTTP is: text over a TCP stream. We'll use example.com, a domain IANA reserves precisely so that examples like this one always have a safe target.

```
uses
  System.Net.Socket, System.SysUtils;  // TSocket lives in System.Net.Socket

procedure SpeakHttpByHand;
var
  LSocket: TSocket;
begin
  // A TCP socket; strings are sent/received as UTF-8 unless you say otherwise.
  LSocket := TSocket.Create(TSocketType.TCP);
  try
    // "Dial." The first parameter is a host NAME, resolved via DNS (Part 2!).
    // Behind this one call: DNS lookup + the TCP three-way handshake.
    LSocket.Connect('example.com', '', '', 80);

    // We are connected. Note the ephemeral local port the OS picked for us:
    Writeln(Format('connected: local port %d -> %s:%d',
      [LSocket.LocalPort, LSocket.RemoteAddress, LSocket.RemotePort]));

    // Both sides can talk now. We go first: a minimal HTTP/1.1 request.
    // HTTP frames its messages with CRLF line endings; a blank line = "done".
    LSocket.Send(
      'GET / HTTP/1.1'      + #13#10 +
      'Host: example.com'   + #13#10 +
      'Connection: close'   + #13#10 +
      #13#10);

    // Listen for the answer. Give the server five seconds to start talking.
    LSocket.ReceiveTimeout := 5000;  // milliseconds; 0 means wait forever
    Writeln(LSocket.ReceiveString);  // "HTTP/1.1 200 OK" + headers + HTML
  finally
    LSocket.Free;  // hang up — the destructor closes the connection
  end;
end;
```

Run that in a console application and a real web server on the other side of the world answers your hand-typed request with `HTTP/1.1 200 OK`, a block of header lines, and the page's HTML. Every phase of the phone call is visible in the code: `Connect` dials, `Send` and `ReceiveString` are the two sides talking, and `Free` hangs up. The `LocalPort` line makes the ephemeral port from our first diagram real — run the procedure twice and you'll see a different number each time, because each run is a brand-new connection with a fresh client-side port.

Notice also what the snippet does *not* pretend: `ReceiveString` returns whatever bytes have arrived so far,

decoded as text — one bucketful from the stream. For a short response that's often everything; for a longer one it's the first chunk, and a real HTTP client keeps reading until `Content-Length` says it has the whole body. That's not a flaw in our code — it's the stream-not-messages lesson showing up exactly where theory said it would, and it's why production code uses a proper HTTP client library on top of the socket.

> **Try it and poke the stream trap**
>
> Replace `example.com` with a server of your own from Part 1 — or split the `Send` into two calls, one per line, and observe that the server neither notices nor cares. It reads a stream; your call boundaries dissolve into it. Seeing that once teaches more than any paragraph.

Decoding the three error messages

With the mental model in place, the error messages your users have been forwarding you for years suddenly become *diagnoses* rather than mysteries — each one tells you precisely which stage of the phone call failed.

Error message	What actually happened	First thing to check

Connection refused	The machine answered — and said *"nothing is listening on that port."* An active "no."	Is the server program actually running? Right port? (Part 1)
Connection timed out	No answer at all. The dial tone rang into the void — wrong address, machine down, or a firewall silently discarding the knock.	Is the address right? (Part 2) Is a firewall in the way?
Connection reset	The call was up, then the other side hung up mid-sentence — the peer crashed, was killed, or force-closed the connection.	Server logs — *it* ended the call, so it knows why.

The refused/timeout distinction is the most useful one in daily debugging: **refused means you reached the machine** (something is alive at that address, just not your service), while **timeout means you never reached it at all**. Two error strings that look like synonyms are actually pointing at opposite ends of the problem.

> **Works on my machine, fails across the network? Firewall.**
> A connect that succeeds against `127.0.0.1` (Part 1's localhost) but times out from a colleague's machine is the classic signature of a **firewall** — software on the server (or a device in between) that silently drops incoming knocks on ports nobody explicitly allowed. Nothing is wrong with your code or your addressing; the door is simply guarded. On Windows, the built-in Windows Defender Firewall prompts you to allow an app the first time it listens — the rule it creates is exactly "let knocks on this port through."

> **Sockets are a universal, open skill**
> Nothing in this post is Delphi-specific knowledge. `TSocket` wraps the same open Berkeley sockets API that Python's `socket` module, Node.js's `net` module, and every other open-source stack wraps — same connect, same send/receive, same stream semantics, same three error messages. Learn it once here, and you've learned it everywhere.

Takeaways

The word "connection" has been demystified, so let's bank what it now means.

- A **connection** is a private two-way byte channel identified by four values: client address and port, server address and port. The server **listens** (passive, on a published port); the client **connects** (active, from an automatically assigned ephemeral port).
- **TCP is the phone call**: dial (handshake), talk both ways with guaranteed complete, in-order delivery — or an error — then hang up. UDP is the postcard, and business apps essentially never write postcards.
- **TCP carries a stream, not messages.** Your write boundaries are not preserved; protocols like HTTP exist precisely to draw message boundaries onto the stream. This is the trap that bites only in production — respect it.
- **Refused** = reached the machine, nothing listening. **Timeout** = never reached anything (address or firewall). **Reset** = the other side hung up on you.

- Delphi's RTL speaks TCP natively via `TSocket` in `System.Net.Socket` — connect, send, receive, close, in a dozen lines and zero dependencies.

> A connection isn't a cable and it isn't magic — it's an agreement between two programs, brokered by their operating systems, to treat a stream of bytes as a conversation.

And did you notice what we actually did in that code snippet? We spoke **raw HTTP by hand** — typed a request line and headers straight into a TCP stream and got a web page back. That means a web server is just a program that listens on a port, reads that text, and answers with more text. What exactly it does between the reading and the answering — status codes, routes, handlers, and how Delphi's WebBroker lets you write one yourself — is Part 4. The mystery is officially gone; next time we build the other end of the call.

> **This series: Networking for Delphi Developers**
> 1. IP addresses, localhost, and ports — naming the machine and the door 2. DNS, the hosts file, and DHCP — how names become addresses 3. **TCP connections and sockets** — this post 4. How a web server works: HTTP and WebBroker — coming July 13 5. Web services, REST, JSON, and TMS XData — coming July 14

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —