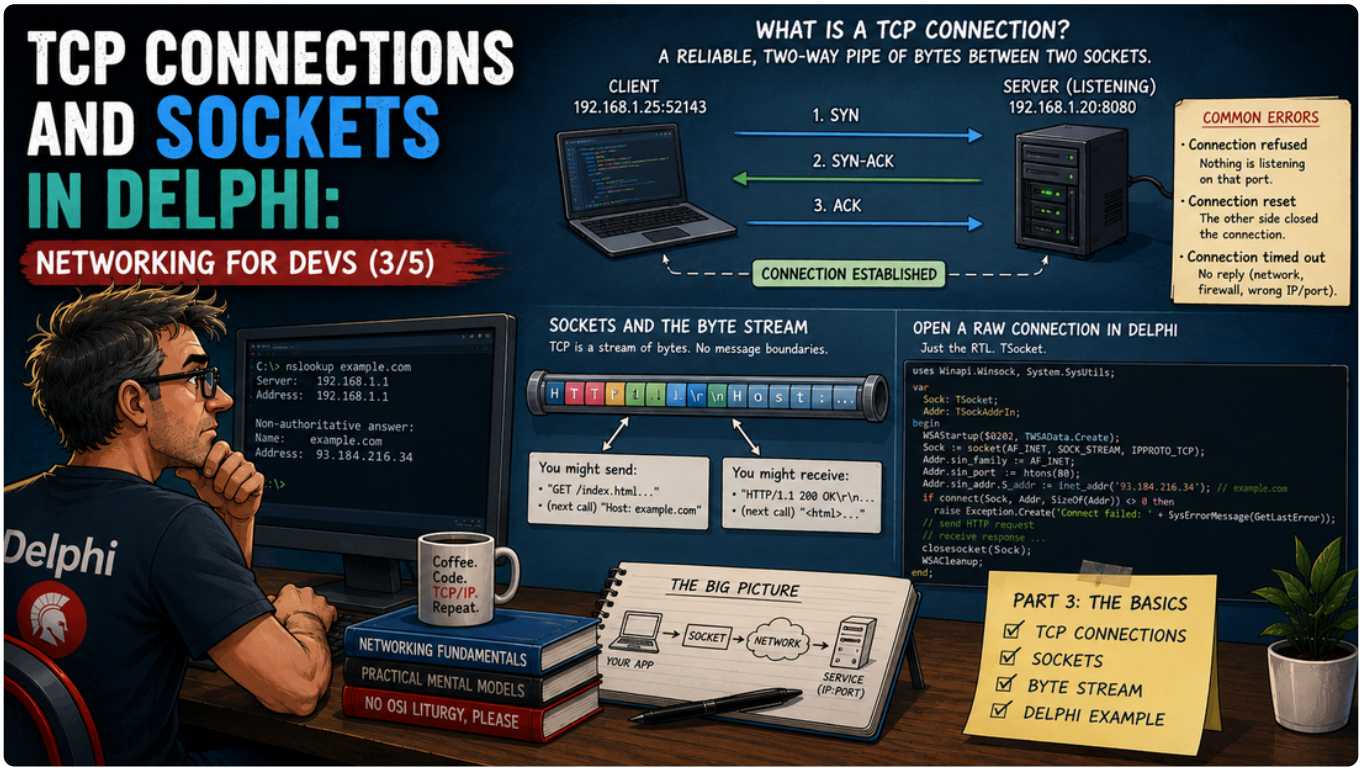
NEWS · JULY 12, 2026

TCP-Verbindungen und Sockets in Delphi: Netzwerk-Grundlagen für Entwickler (3/5)

By Dr. Holger Flick



In Teil 1 haben wir gelernt, wie man eine Maschine und eine Tür an ihr benennt — IP-Adressen und Ports. In Teil 2 haben wir gesehen, wie aus Namen Adressen werden — DNS, die hosts-Datei und DHCP. Sie können also inzwischen präzise sagen: *„der Dienst auf 192.168.1.20, Port 8080."*

Aber jetzt kommt das, was jeder Fehlerdialog stillschweigend voraussetzt und niemand je erklärt: Was *ist* eigentlich eine Verbindung? Ihre Anwender sehen **"connection refused"**, **"connection reset"**, **"connection timed out"** — drei verschiedene Meldungen, drei völlig verschiedene Ursachen — und das Wort, das in allen dreien die ganze Arbeit leistet, haben wir nie wirklich definiert.

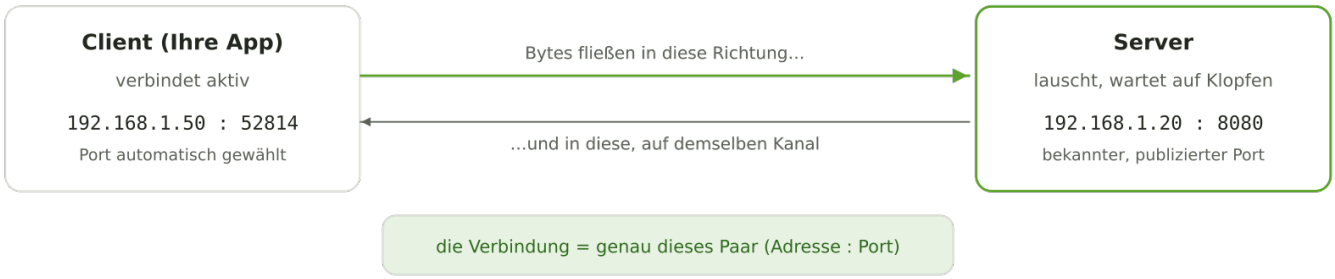
Genau das holen wir heute nach. Am Ende dieses Beitrags wissen Sie exakt, was eine TCP-Verbindung ist, warum eine Seite *lauscht*, während die andere *verbindet*, warum TCP Ihnen einen Strom von Bytes liefert statt sauber verpackter Nachrichten (die Anfängerfalle Nummer eins) — und als Belohnung öffnen Sie aus Delphi-Code heraus eine rohe Verbindung und sprechen *von Hand* mit einem echten Webserver, mit nichts als der modernen RTL. Keine Komponenten auf einem Formular, keine Drittanbieter-Bibliothek. Nur `TSocket` und rund zwanzig Zeilen.

Zwei Rollen: eine Seite lauscht, eine Seite verbindet

Bevor auch nur ein Byte über die Leitung geht, spielen die beiden beteiligten Programme streng unterschiedliche Rollen — und diese Asymmetrie ist der Schlüssel zu allem Weiteren in diesem Beitrag.

Ein **Server** ist ein Programm, das dem Betriebssystem sagt: *„Ich interessiere mich für Port 8080. Wenn jemand anklopft, weck mich auf."* Das nennt man **Lauschen** (listening), und es ist völlig passiv — ein lauschender Server verbraucht beim Warten fast nichts. Ein **Client** ist die aktive Partei: Er sagt *„verbinde mich mit 192.168.1.20, Port 8080"*, und das Betriebssystem geht hin und klopft an.

Wird das Klopfen beantwortet, entsteht etwas Neues, das vorher nicht da war: eine **Verbindung** — ein privater Zwei-Wege-Kanal zwischen genau diesen beiden Programmen. Und hier kommt der Teil, der Fehlermeldungen endlich verständlich macht: Die Verbindung wird durch *vier* Werte identifiziert, nicht durch zwei.



Eine Verbindung ist das Paar der Endpunkte: Client-Adresse:Port ” Server-Adresse:Port

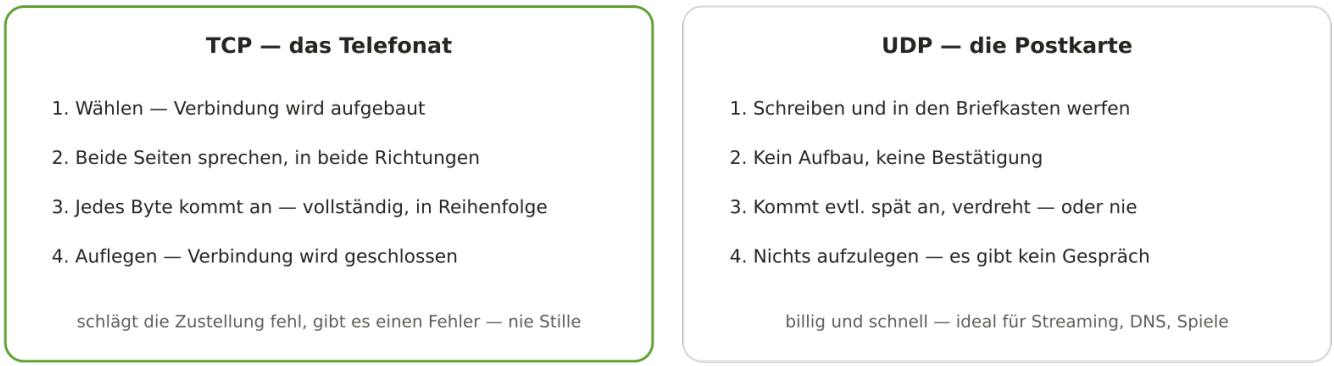
Lesen Sie das Diagramm von links nach rechts: Adresse und Port des Servers sind die bekannten Werte — Sie haben sie konfiguriert, Sie haben sie veröffentlicht, Ihre Anwender tippen sie ein. Den Port des Clients dagegen haben Sie nie gewählt. Wenn Ihre App „connect" aufruft, weist das Betriebssystem ihr automatisch einen temporären lokalen Port zu — einen Ephemeral Port, gezogen aus einem hohen Bereich und beim Schließen der Verbindung wieder freigegeben. Genau deshalb kann eine einzige Maschine Tausende gleichzeitiger Verbindungen zum selben Server halten: Jede hat einen anderen Client-Port, also ist jedes Vierer-Tupel eindeutig.

Deshalb kann Ihr Browser auch fünf Tabs derselben Website öffnen, ohne dass sie sich in die Quere kommen: fünf Verbindungen, fünf verschiedene Ephemeral Ports, fünf getrennte Kanäle.

TCP ist ein Telefonat, UDP eine Postkarte

Das Internet bietet tatsächlich zwei grundverschiedene Arten an, Daten zu verschicken — und ein einziges Bild genügt, um sie für den Rest Ihrer Laufbahn auseinanderzuhalten.

TCP — das Transmission Control Protocol, heute definiert in RFC 9293 — funktioniert wie ein **Telefonat**: Sie wählen, die Gegenseite nimmt ab, beide können beliebig lange sprechen und wissen, dass jedes Wort in der richtigen Reihenfolge ankommt — und irgendwann legt jemand auf. UDP — das User Datagram Protocol, RFC 768 — funktioniert wie eine **Postkarte**: Sie schreiben sie, werfen sie in den Kasten und hoffen. Kein Verbindungsaufbau, keine Bestätigung, keine Garantie für Ankunft oder Reihenfolge.



TCP ist ein Telefonat mit Garantien; UDP eine Postkarte mit Hoffnung

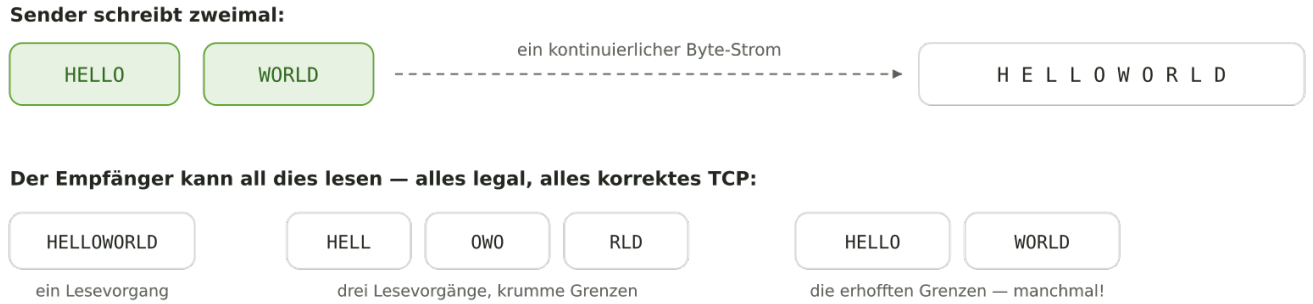
Die Lehre aus diesem Bild: Alles, was Sie als Entwickler von Geschäftsanwendungen tun — Datenbanken, REST-Aufrufe, Dateiübertragungen, Webserver — läuft auf der linken Seite. TCPs Versprechen ist genau das, was Ihre Buchhaltungsdaten brauchen: **Bytes kommen vollständig und in Sendereihenfolge an — oder Sie bekommen einen Fehler.** Nie stille Datenkorruption, nie eine geheimnisvoll durchgewürfelte Rechnung. UDP hat ehrbare Aufgaben (die DNS-Anfragen aus Teil 2 reisen typischerweise per UDP, ebenso Videotelefonate und Spiele, wo ein verspätetes Paket ohnehin wertlos ist), aber wir werden es in dieser Serie nicht mehr brauchen.

Der „Wählen"-Schritt des Telefonats trägt einen berühmten Namen, der Ihnen in jedem Netzwerk-Text begegnen wird: der Drei-Wege-Handshake, ein kurzer Austausch von drei Paketen, in dem sich beide Seiten darauf einigen, dass die Verbindung existiert — Ihr Code bekommt davon nichts mit, das Betriebssystem erledigt das gesamte Ritual innerhalb dieses einen „connect"-Aufrufs.

Die Falle: TCP ist ein Stream, keine Nachrichten

Jetzt zur wichtigsten praktischen Tatsache dieses Beitrags — derjenigen, die Entwickler, die mit mysteriösen sporadischen Bugs kämpfen, von denen trennt, die es nicht tun.

Man könnte annehmen: Sendet der Absender `"HELLO"` und danach `"WORLD"`, erhält der Empfänger zwei saubere Pakete — `"HELLO"`, dann `"WORLD"`. **TCP verspricht nichts dergleichen.** TCP garantiert die *Bytes* und ihre *Reihenfolge* — mehr nicht. Es ist ein kontinuierlicher Strom, wie Wasser durch ein Rohr: Was Sie hineingegossen haben, kommt vollständig und in der richtigen Abfolge heraus, aber die „Eimerfüllungen" auf der Empfängerseite können beliebig geteilt und zusammengelegt sein — so, wie es dem Netz gerade passte.



Bytes und Reihenfolge sind garantiert — wo eine „Nachricht" endet, ist Sache Ihres Protokolls

TCP bewahrt Bytes und Reihenfolge — nicht die Grenzen Ihrer Schreibvorgänge

Was dieses Diagramm Ihnen wirklich sagt: **„Nachricht" ist kein TCP-Konzept.** Wenn Ihre Anwendung Nachrichten braucht — und das tut sie fast immer —, muss *etwas oberhalb von TCP* festlegen, wo eine Nachricht endet und die nächste beginnt. Dieses Etwas heißt **Protokoll**, und es gibt zwei klassische Techniken: jede Nachricht mit einem bekannten Trennzeichen abschließen (etwa einem Zeilenumbruch) oder die Länge vorweg schicken („die nächsten 512 Bytes sind eine Nachricht"). Genau das tut HTTP — Header-Zeilen enden mit CRLF, eine Leerzeile beendet den Header-Block, und ein `Content-Length`-Header kündigt an, wie viele Body-Bytes folgen. HTTP ist, wie sich zeigt, nichts weiter als ein Satz Framing-Regeln für Text über einen TCP-Stream. Behalten Sie den Gedanken; wir beweisen ihn gleich.

Das Heimtückische an dieser Falle: In einem schnellen lokalen Netz kommen Ihre Schreibvorgänge *häufig tatsächlich* jeweils am Stück an — der naive Code funktioniert also auf Ihrem Rechner, funktioniert in der Demo und scheitert beim Kunden unter Last. Wenn Sie sich eine einzige technische Tatsache aus diesem Beitrag merken, dann diese.

Von Hand mit einem echten Server sprechen — in Delphi

Zeit, das alles konkret zu machen. Modernes Delphi liefert eine leichtgewichtige Socket-Klasse direkt in der RTL: `TSocket` in der Unit `System.Net.Socket` — ein dünner, plattformübergreifender Wrapper über die Berkeley-Sockets-API des Betriebssystems, dieselbe kampferprobte Schnittstelle, die praktisch jede Programmiersprache kapselt. Sie ist seit Jahren Teil der RTL und selbstverständlich auch in Delphi 13 Florence enthalten, dem aktuellen Release. Keine Pakete zu installieren, keine Komponenten abzulegen.

Und jetzt der unterhaltsame Teil: Statt uns mit irgendeinem Spielzeug-Echo-Dienst zu verbinden, verbinden wir uns mit einem **echten Webserver** auf Port 80 und sprechen rohes HTTP mit ihm — denn nach dem letzten Abschnitt wissen Sie genau, was HTTP ist: Text über einen TCP-Stream. Wir verwenden example.com, eine Domain, die die IANA genau dafür reserviert, damit Beispiele wie dieses immer ein sicheres Ziel haben.

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

procedure SpeakHttpByHand;
var
  LSocket: TSocket;
begin
  // Ein TCP-Socket; Strings werden als UTF-8 gesendet/empfangen, sofern nicht anders angegeben.
  LSocket := TSocket.Create(TSocketType.TCP);
  try
    // "Wählen." Der erste Parameter ist ein Host-NAME, aufgelöst per DNS (Teil 2!).
    // Hinter diesem einen Aufruf: DNS-Lookup + der TCP-Drei-Wege-Handshake.
    LSocket.Connect('example.com', '', '', 80);

    // Wir sind verbunden. Man beachte den Ephemeral Port, den das OS für uns gewählt hat:
    Writeln(Format('connected: local port %d -> %s:%d',
      [LSocket.LocalPort, LSocket.RemoteAddress, LSocket.RemotePort]));

    // Beide Seiten können jetzt sprechen. Wir beginnen: ein minimaler HTTP/1.1-Request.
    // HTTP rahmt seine Nachrichten mit CRLF-Zeilenenden; eine Leerzeile = "fertig".
    LSocket.Send(
      'GET / HTTP/1.1'      + #13#10 +
      'Host: example.com'   + #13#10 +
      'Connection: close'   + #13#10 +
      #13#10);

    // Auf die Antwort lauschen. Dem Server fünf Sekunden Zeit geben loszulegen.
    LSocket.ReceiveTimeout := 5000;  // Millisekunden; 0 heißt: ewig warten
    Writeln(LSocket.ReceiveString);  // "HTTP/1.1 200 OK" + Header + HTML
  finally
    LSocket.Free;  // auflegen – der Destruktor schließt die Verbindung
  end;
end;
```

Führen Sie das in einer Konsolenanwendung aus, und ein echter Webserver am anderen Ende der Welt beantwortet Ihre handgetippte Anfrage mit `HTTP/1.1 200 OK`, einem Block Header-Zeilen und dem HTML der Seite. Jede Phase des Telefonats ist im Code sichtbar: `Connect` wählt, `Send` und `ReceiveString` sind die beiden sprechenden Seiten, und `Free` legt auf. Die `LocalPort`-Zeile macht den Ephemeral Port aus unserem ersten Diagramm greifbar — rufen Sie die Prozedur zweimal auf, und Sie sehen jedes Mal eine andere Nummer, denn jeder Lauf ist eine brandneue Verbindung mit frischem Client-Port.

Beachten Sie auch, was das Snippet *nicht* vortäuscht: `ReceiveString` liefert die bislang eingetroffenen Bytes,

als Text dekodiert — eine Eimerfüllung aus dem Stream. Bei einer kurzen Antwort ist das oft alles; bei einer längeren ist es der erste Brocken, und ein echter HTTP-Client liest weiter, bis `Content-Length` sagt, dass der Body vollständig ist. Das ist kein Fehler in unserem Code — es ist die Stream-statt-Nachrichten-Lektion, die genau dort auftaucht, wo die Theorie es vorhergesagt hat. Und es ist der Grund, warum Produktionscode eine richtige HTTP-Client-Bibliothek oberhalb des Sockets verwendet.

> **Probieren Sie es aus und stochern Sie in der Stream-Falle**
> Ersetzen Sie `example.com` durch einen eigenen Server aus Teil 1 — oder teilen Sie das `Send` in zwei Aufrufe auf, einen pro Zeile, und beobachten Sie, dass der Server das weder bemerkt noch es ihn kümmert. Er liest einen Stream; Ihre Aufrufgrenzen lösen sich darin auf. Das einmal gesehen zu haben, lehrt mehr als jeder Absatz.

Die drei Fehlermeldungen entschlüsselt

Mit dem mentalen Modell im Kopf werden die Fehlermeldungen, die Ihre Anwender Ihnen seit Jahren weiterleiten, plötzlich zu *Diagnosen* statt zu Rätseln — jede sagt Ihnen präzise, in welcher Phase das Telefonat gescheitert ist.

Fehlermeldung	Was tatsächlich passiert ist	Zuerst prüfen
Connection refused	Die Maschine hat geantwortet — und gesagt: *„Auf diesem Port lauscht nichts."* Ein aktives „Nein".	Läuft das Serverprogramm überhaupt? Richtiger Port? (Teil 1)
Connection timed out	Gar keine Antwort. Das Freizeichen verhallte im Nichts — falsche Adresse, Maschine aus, oder eine Firewall verwirft das Klopfen stillschweigend.	Stimmt die Adresse? (Teil 2) Steht eine Firewall im Weg?
Connection reset	Das Gespräch lief — dann hat die Gegenseite mitten im Satz aufgelegt: Der Peer ist abgestürzt, wurde beendet oder hat die Verbindung hart geschlossen.	Server-Logs — *er* hat das Gespräch beendet, also weiß er, warum.

Die Unterscheidung refused/timeout ist die nützlichste im Debugging-Alltag: **refused heißt, Sie haben die Maschine erreicht** (an dieser Adresse lebt etwas, nur nicht Ihr Dienst), während **timeout heißt, Sie haben sie nie erreicht**. Zwei Fehlertexte, die wie Synonyme aussehen, zeigen in Wahrheit auf entgegengesetzte Enden des Problems.

> **Funktioniert lokal, scheitert übers Netz? Firewall.**
> Ein Connect, der gegen `127.0.0.1` (das localhost aus Teil 1) gelingt, aber vom Rechner eines Kollegen in einen Timeout läuft, ist die klassische Signatur einer **Firewall** — Software auf dem Server (oder ein Gerät dazwischen), die eingehendes Klopfen auf nicht ausdrücklich erlaubten Ports stillschweigend verwirft. An Ihrem Code oder Ihrer Adressierung ist nichts falsch; die Tür wird schlicht bewacht. Unter Windows fragt die eingebaute Windows Defender Firewall beim ersten Lauschen einer App nach — die dabei erzeugte Regel bedeutet exakt „lass Klopfen auf diesem Port durch".

> **Sockets sind eine universelle, offene Fähigkeit**
> Nichts in diesem Beitrag ist Delphi-spezifisches Wissen. `TSocket` kapselt dieselbe offene Berkeley-Sockets-API wie Pythons `socket`-Modul, Node.js' `net`-Modul und jeder andere Open-Source-Stack — dasselbe connect, dasselbe send/receive, dieselbe Stream-Semantik, dieselben drei Fehlermeldungen. Einmal hier gelernt, überall gelernt.

Fazit

Das Wort „Verbindung" ist entmystifiziert — halten wir fest, was es jetzt bedeutet.

- Eine **Verbindung** ist ein privater Zwei-Wege-Bytekanal, identifiziert durch vier Werte: Client-Adresse und -Port, Server-Adresse und -Port. Der Server **lauscht** (passiv, auf einem publizierten Port); der Client **verbindet** (aktiv, von einem automatisch zugewiesenen Ephemeral Port).
-

TCP ist das Telefonat: wählen (Handshake), in beide Richtungen sprechen mit garantiert vollständiger Zustellung in richtiger Reihenfolge — oder einem Fehler —, dann auflegen. UDP ist die Postkarte, und Geschäftsanwendungen schreiben praktisch nie Postkarten.

- **TCP transportiert einen Stream, keine Nachrichten.** Ihre Schreibgrenzen bleiben nicht erhalten; Protokolle wie HTTP existieren genau deshalb, um Nachrichtengrenzen auf den Stream zu zeichnen. Das ist die Falle, die erst in Produktion zuschnappt — nehmen Sie sie ernst.
- **Refused** = Maschine erreicht, nichts lauscht. **Timeout** = nie etwas erreicht (Adresse oder Firewall). **Reset** = die Gegenseite hat aufgelegt.
- Delphis RTL spricht TCP nativ über `TSocket` in `System.Net.Socket` — connect, send, receive, close, in einem Dutzend Zeilen und ohne jede Abhängigkeit.

> Eine Verbindung ist kein Kabel und keine Magie — sie ist eine Vereinbarung zwischen zwei Programmen, vermittelt durch ihre Betriebssysteme, einen Strom von Bytes als Gespräch zu behandeln.

Und ist Ihnen aufgefallen, was wir in dem Code-Snippet tatsächlich getan haben? Wir haben **rohes HTTP von Hand gesprochen** — eine Request-Zeile und Header direkt in einen TCP-Stream getippt und eine Webseite zurückbekommen. Das heißt: Ein Webserver ist bloß ein Programm, das auf einem Port lauscht, diesen Text liest und mit weiterem Text antwortet. Was genau zwischen dem Lesen und dem Antworten passiert — Statuscodes, Routen, Handler, und wie Sie mit Delphis WebBroker selbst einen schreiben — ist Teil 4. Das Rätsel ist offiziell gelöst; nächstes Mal bauen wir das andere Ende des Gesprächs.

Diese Serie: Netzwerk-Grundlagen für Delphi-Entwickler

1. IP-Adressen, localhost und Ports — die Maschine und die Tür benennen 2. DNS, die hosts-Datei und DHCP — wie aus Namen Adressen werden 3. **TCP-Verbindungen und Sockets** — dieser Beitrag 4. Wie ein Webserver funktioniert: HTTP und WebBroker — erscheint am 13. Juli 5. Web Services, REST, JSON und TMS XData — erscheint am 14. Juli

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 —