



Sk

Posted on Feb 26 • Edited on Feb 27 • Originally published at [Medium](#)

11



1



2



2

How To Build A Raw TCP Server Where Every Millisecond Counts.

[#javascript](#) [#webdev](#) [#tutorial](#) [#beginners](#)

Low Level JavaScript and Systems (3 Part Series)

- 1 Buffers in Node.js: What They Do & Why You Should Care
- 2 **How To Build A Raw TCP Server Where Every Millisecond C...**
- 3 Building Distributed Systems with Node.js.

TCP is blazingly fast. 🦀

When every millisecond matters—real-time systems, database engines, caches, brokers, mission-critical software; these systems speak **raw TCP**.

Because when speed and control are the priority, **abstraction is the enemy**.

HTTP, while reliable, comes at a cost. It sits on top of TCP and adds overhead: headers, footers, encoding, decoding-**latency creeps in**.

The Cost of HTTP Overhead

Check out what happens under the hood with an HTTP response:

```
StatusCode      : 200
StatusDescription : OK
Content         : {72, 101, 108, 108...}
RawContent      : HTTP/1.1 200 OK
                  Content-Length: 12
                  Hello world!
Headers         : {[Content-Length, 12]}
RawContentLength : 12
```

HTTP is **text-based**. It requires parsing, extra processing, and additional steps before data even hits the network.

Now compare that to raw TCP:

```
const data = Buffer.from("hello")
tcpserverClientConnection.write(data)
```

No overhead. No extra encoding or decoding. **Protocol-less**. Just raw data, **from point A to B, as fast as possible**.

An even faster transport exists: **UDP**-used in real-time apps (P2P gaming, live video streaming). But UDP trades reliability for speed.

TCP: Three Layers Below HTTP




The deeper you go, the **more control you have**:

```
HTTP
└─ Encoding/Decoding
    └─ Binary/Buffers
        └─ TCP
```

If you want to be more than a CRUD engineer, if you want to build networking **systems**-understanding TCP is **non-negotiable**.

TCP: A Practical Introduction

One of the most **powerful** things about Node.js? **It is what you make it.**

- Need a CRUD API? 
- Need a systems engine? 
- Need to extend it with C++? 

Your perspective on Node defines what you can build.

Let's start simple: working directly with raw memory.

```
const data = Buffer.alloc(4)
data.writeInt32BE(7) // Store the number 7 as raw binary
```

What's happening here?

- **Allocate 4 bytes (32 bits) of memory**
- **Write a 32-bit integer (7) as raw data**

This is **lower-level** JavaScript.

Now, let's **go deeper** and build a raw TCP server.

Writing a TCP Server in Node.js

A bare-bones TCP server:

```
// server.js
const net = require("node:net")
const server = net.createServer((c) => {
  c.on("data", (data) => console.log(data))
  c.on("error", (err) => console.log(err))
  c.write("hello world") // Implicitly converts string to a buffer
  c.end()
})
server.listen(3000)
```

Now, run the server and hit it with curl:

```
curl http://localhost:3000
```

 ERROR:

```
curl: The server committed a protocol violation.
```

Why? Because **we're speaking raw TCP**, and curl expects HTTP (headers, status codes, and structured responses).

Let's create a **TCP client** instead:

```
//client.js
const net = require("node:net")
const c = net.createConnection({ port: 3000, host: "localhost" })
c.on("data", (data) => console.log(data))
c.write("hello")
```

Run the server and the client, and here's the output:

```
<Buffer 48 65 6c 6c 6f 20 77 6f 72 6c 64 21>
```

That's raw TCP at work. But we can **decode it** easily since it's only a string:

```
c.on("data", (data) => {
  console.log(data.toString()) // "hello world"
})
```

The Power of Raw TCP

So why does this matter? Why should you care?

Because some of the most legendary systems you use daily run on pure TCP:

- **MySQL clients**
- **RabbitMQ**
- **Redis**
- **Neo4j**

- **Email clients (SMTP, IMAP, POP3)**

These systems don't need HTTP's baggage-no cookies, no headers, no JSON parsing. Just **raw, efficient, custom protocols**.

Building a Custom Protocol

A protocol is simply an agreement:

"When you send me data, I expect it in **this** format, or I reject it."

Here's how HTTP enforces its protocol:

```
HTTP/1.1 200 OK
Content-Length: 12
Hello world!
```

Let's build our **own** simple **binary protocol** on top of raw TCP.

Defining Our Protocol

We'll structure our buffer like this:

```
buffer = msg length (4 bytes) | msg (variable length) | metadata (variable leng
```

First 4 bytes → Message length

Next bytes → Message

Remaining bytes → Metadata

Now let's **encode a message** server-side:

```
const server = net.createServer((c) => {
  c.on("data", (data) => console.log(data.toString()))
  const data = Buffer.from("hello world") // Encode message
  const metadata = Buffer.from([0x00]) // No metadata
  const len = Buffer.alloc(4) // Allocate 4 bytes for length
```

```
len.writeInt32BE(data.length, 0) // Store message length
const combinedBuffer = Buffer.concat([len, data, metadata])
c.write(combinedBuffer) // Send to client
c.end()
})
```

Now, on the **client side**, we'll decode this structured message:

```
const c = net.createConnection({ port: 3000, host: "localhost" })
c.on("data", (data) => {
  console.log(data)
  const len = data.readInt32BE(0) // Read message length
  console.log(len)
  const msg = data.subarray(4, 4 + len) // Extract message
  console.log(msg.toString())
  const metadata = data.subarray(4 + len) // Extract metadata
  console.log(metadata)
})

c.write("hello")
```



And just like that, we've built a structured TCP messaging system.

The Bigger Picture

Believe it or not, this is how real-world networking applications work. In C, this buffer-based approach is the standard for high-performance data exchange.

By going beyond HTTP and working with raw TCP, you unlock the ability to build low-level networking systems.

What's Next?

This article is a tiny snippet from a larger series where we build a message broker for distributed systems from scratch.



image no longer exists

[Beyond APIs and Endpoints: Building a TCP-Powered Message Broker. - Payhip](#)

payhip.com

You'll learn about:

- ✓ Long-lived TCP connections
- ✓ Buffer handling
- ✓ Serialization & deserialization
- ✓ Connection management
- ✓ Distributed systems architecture

If you've made it this far, you already think like a systems engineer-and that's rare.

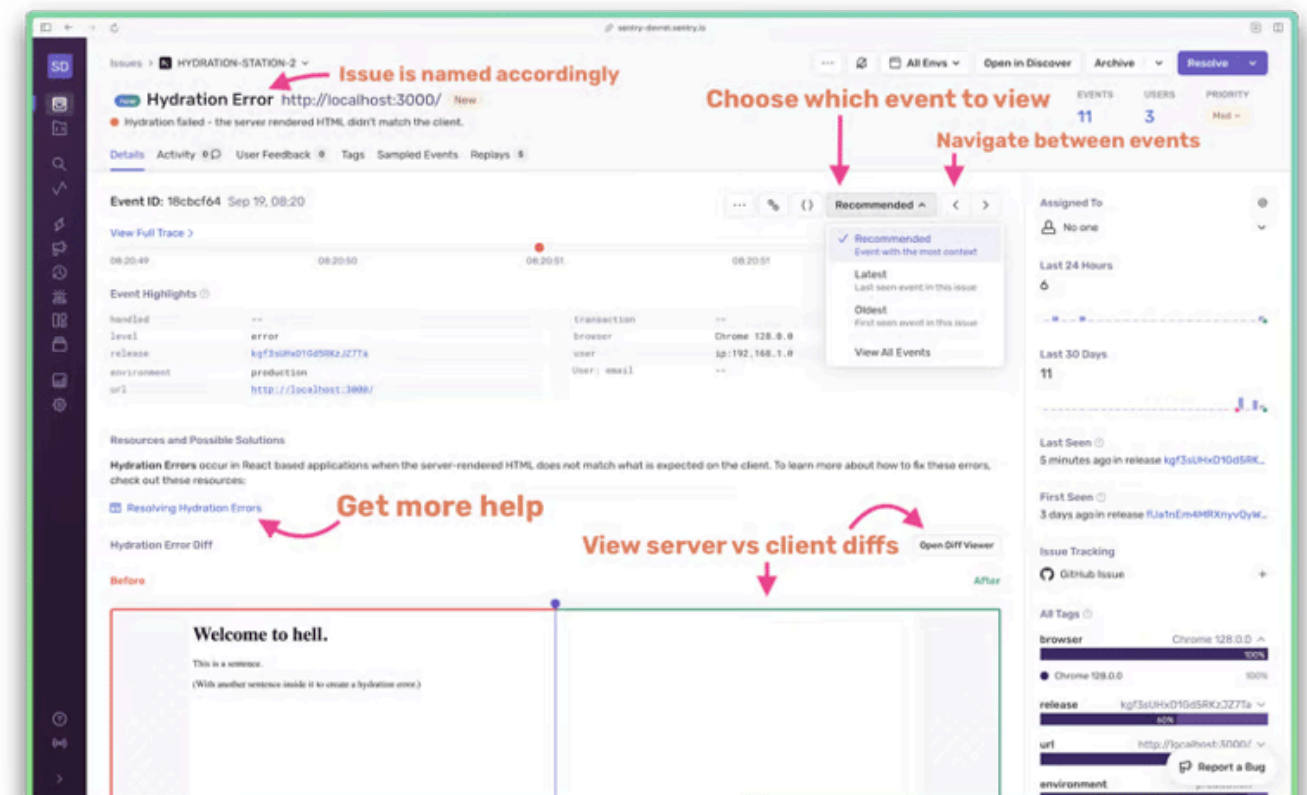
The deeper you go, the more valuable you become.

You can find me on [x](#)

see ya! 🙌

Low Level JavaScript and Systems (3 Part Series)

- 1 Buffers in Node.js: What They Do & Why You Should Care
- 2 **How To Build A Raw TCP Server Where Every Millisecond C...**
- 3 Building Distributed Systems with Node.js.

[Read More](#)

Top comments (8)



Sk • Feb 26



Bonus you can turn a TCP socket to HTTP, by simply passing an HTTP formatted protocol:


```
const CRLF = '\r\n';

c.write(

    `HTTP/1.1 200 OK${CRLF}` +

    `Content-Length: 11${CRLF}${CRLF}` +

    `Hello world`

);
```

Now curl and the browser will understand you



Dave Cridland · Mar 4



Although use legal HTTP/1.1, so include a Host header field, and quite possibly Connection too unless you really wanted it to stay open.



Shekhar Rajput · Feb 26



For anything you are willing to save even a few millisecond, choosing Node is like using bicycle to go on a moto gp



Sk · Feb 26



A pure JavaScript BSON serializer can outperform one built with C++ NAPI. Node.js itself is C++, which is why performance depends on the layer you're using, TCP, is just a thin wrapper around native code (tcp_wrap.cpp).

There's a reason we have tensorflow.js with cpp bindings just as python has C bindings It's perspective and how deep you can go! that's all



Shekhar Rajput · Feb 27



Check out some benchmarks with different HTTP servers setup on different language. Then you'll get to know, when requests reach >40-50K it's so

slow.



Sk · Feb 27

1. HTTP is not TCP but built on tcp **the entire idea of the article**
2. HTTP speak strings, TCP speaks raw bytes/binary entire point of the article so.....
3. TCP !=HTTP the article is about TCP

HTTP:

```

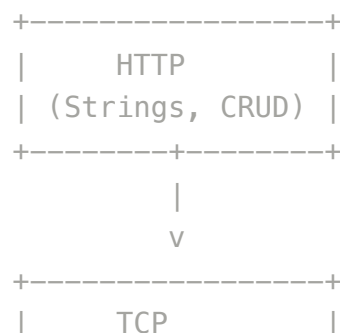
StatusCode      : 200
StatusDescription : OK
Content          : {72, 101, 108, 108...}
RawContent       : HTTP/1.1 200 OK
                  Content-Length: 12
                  Hello world!
Headers          : {[Content-Length, 12]}
RawContentLength : 12
  
```

TCP:

```
<Buffer 48 65 6c 6c 6f 20 77 6f 72 6c 64 21>
```

In conclusion as I did in my comment and article: Node.js is not for CRUD(HTTP) only, its a c++ engine with raw binary capabilities as TCP has shown.

BSON serializer and TCP has nothing to do with CRUD and HTTP. CRUD mindset will be the death of software 🙄



| (Raw Bytes) |
+-----+



Dan Silcox • Mar 4



Dave Cridland • Mar 4



There's some slightly odd things here, like suggesting that IMAP is "pure TCP"; IMAP has just as much parsing overhead as HTTP (possibly more, in fact). I've written parsers for both, and with the latest HTTP/1.1 specifications, it's reasonably easy - whereas IMAP has nested structures and literals and other weird things. HTTP/2 is rather better in this regard than either. XMPP, despite appearances, is pretty fast to parse (you can get it to close to a byte-by-byte scan). SMTP is middling; but in general we're more concerned with throughput on SMTP than we are about latency.

Also, TCP is surprisingly not that fast - it's fine once "warmed up", which is one of the reasons why later versions of HTTP (like HTTP/1.1, but also HTTP/2) use long-lived TCP connections. But it takes a while to get going, which is one of the reasons behind QUIC and other things.

Where using "raw TCP" comes into its own is in three areas:

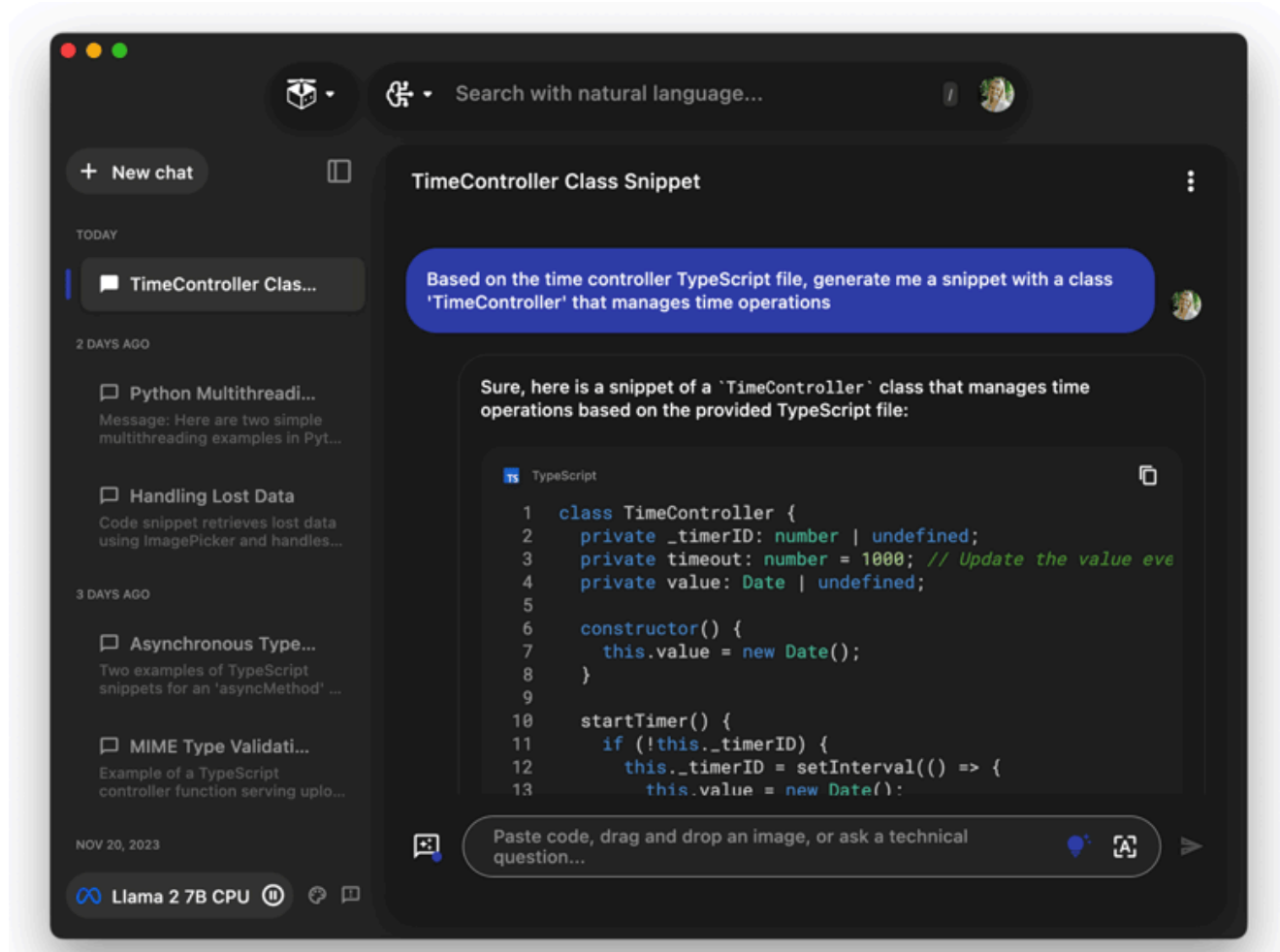
- First, long-lived connections mean that you can switch to stateful messages, which allow you to amortise information across multiple messages (or, put more simply, you can rely on the state instead of carrying it with every message, so your messages are smaller).
- Second, it allows for more expensive connection setup - email and XMPP both use SASL as a multi-round-trip authentication, instead of the username/password plain text protocols HTTP is generally forced into.
- Finally, you can play about with alternative message structures instead of simple request/response, which allows you to optimise response.

All of these are addressed to some degree in HTTP/2 - and HTTP/3, being on QUIC, rids itself of the TCP legacies as well.

 Pieces.app PROMOTED

...

[A Workflow Copilot. Tailored to You.](#)



Our desktop app, with its intelligent copilot, streamlines coding by generating snippets, extracting code from screenshots, and accelerating problem-solving.

[Read the docs](#)**Sk**

Software & ML Developer @darkpools.ai, lead backend engineer.

JOINED

Dec 23, 2020

More from Sk

How to Say "Hello World" in x86 Assembly

#tutorial #beginners #javascript #assembly

Building Distributed Systems with Node.js.

#webdev #beginners #tutorial #javascript

Buffers in Node.js: What They Do & Why You Should Care

#tutorial #javascript #webdev #node



Neon PROMOTED



Guide to Soft Deletes in Laravel and Postgres

Learn how to implement and optimize soft deletes in Laravel for improved data management and integrity.

[See Article →](#)

Guide to Fine-Grained Authorization in Laravel with Postgres

Learn how to set up and utilize Laravel's powerful authorization features.

[See Article →](#)
