Socket Programming with Node.js

Flavio Copes

Updated August 3, 2026

Contents

Build TCP and UDP programs with Node.js, frame messages, handle unreliable connections, and design a small protocol you can inspect on the wire.

What you'll learn: Build and test TCP and UDP clients and servers that frame input, apply limits, handle failures, and support multiple clients safely.

TCP foundations

Build a TCP server and client, exchange bytes, and control the listening address.

Build a TCP server

Create a loopback TCP server, accept a connection, send one message, and close it cleanly.

A **socket** is the endpoint your program uses to talk over a network. For TCP, one side listens and the other connects. In this lesson you build the listening side.

Node's `node:net` module exposes TCP as asynchronous sockets. A server accepts connections; each accepted socket represents one independent byte stream between your process and one client.

Create `server.mjs`:

```
import net from 'node:net'

const server = net.createServer(socket => {
  socket.end('hello from TCP\n')
})

server.listen(4000, '127.0.0.1', () => {
  console.log('listening on 127.0.0.1:4000')
})
```

`net.createServer()` takes a callback that runs once per accepted connection. The `socket` argument is a duplex stream: you read client bytes from it and write replies to it. Here we write one line and close with `socket.end()`.

`server.listen(4000, '127.0.0.1')` asks the operating system to reserve TCP port 4000 on the loopback interface. From that point the kernel completes handshakes for you and hands each new connection to the callback.

Verify the listener

Run the server, then check the port from another terminal:

```
ss -lnt | grep 4000
# LISTEN 0  511  127.0.0.1:4000  0.0.0.0:*
```

On macOS use `lsof -nP -iTCP:4000 -sTCP:LISTEN` instead. Either way you should see exactly one listener, and the local address column should show it bound only to loopback.

Now connect as a client with netcat:

```
nc 127.0.0.1 4000
# hello from TCP
```

`nc` prints the greeting and exits, because the server closed the connection right after writing.

When the port is taken

Start a second copy of the server while the first is still running:

```
Error: listen EADDRINUSE: address already in use 127.0.0.1:4000
```

`EADDRINUSE` means another process already owns that address and port pair. Find it with `ss -lntp` on Linux or the `lsof` command above, stop it, then retry. You'll hit this constantly in development, usually because an old copy of your own server is still running in a forgotten terminal.

Keep the first lab on `127.0.0.1`. Binding to all interfaces exposes the service to other reachable devices, and this server has no authentication yet.

Build a TCP client

Connect to the local server, read stream data, handle errors, and observe connection closure.

The client is the side that initiates. It needs two things: the server's address and its port. Together they identify one specific listener on one specific machine.

A TCP client connects to an address and port, then exchanges bytes on the resulting stream. The connection event means the three-way handshake completed; it does not guarantee a useful application response. The server could accept you and then say nothing.

Create `client.mjs`:

```
import net from 'node:net'

const socket = net.createConnection({ host: '127.0.0.1', port: 4000 })

socket.on('data', data => process.stdout.write(data))
socket.on('end', () => console.log('server closed'))
socket.on('error', error => console.error(error.message))
```

`net.createConnection()` starts the handshake immediately. Each `data` event delivers whatever bytes the server sent. The `end` event fires when the server closes its side of the stream.

Run the server from the previous lesson first, then the client:

```
node client.mjs
# hello from TCP
# server closed
```

When nobody is listening

Stop the server and run the client again:

```
node client.mjs
# connect ECONNREFUSED 127.0.0.1:4000
```

`ECONNREFUSED` means your packet reached the machine, but no process was listening on that port, so the kernel actively rejected the connection. It's a different failure from a timeout, where packets vanish and you wait. Refusal is fast and definitive: wrong port, or the server isn't running.

Seeing connection refusal as a separate failure matters when debugging. Refused points at the server process. A hang points at the network or a firewall.

Sending data too

A real client writes as well as reads. Add a request:

```
socket.on('connect', () => {
```

```
  socket.write('PING\n')
})
```

You can call `socket.write()` before the connection is up — Node queues the bytes and flushes them after the handshake — but writing inside the `connect` handler makes the sequence explicit.

Always attach an error handler. An unhandled `error` event on a socket throws, and that can terminate the whole Node process. Without the handler, a routine network failure like a refused connection becomes a crash.

Treat TCP as a byte stream

Stop assuming one write becomes one data event and inspect how application bytes can be split or combined.

TCP looks like it sends messages. It does not. TCP is a **byte stream**: it preserves byte order, not message boundaries.

One `socket.write()` may arrive through several `data` events, and several writes may arrive together in one. The kernel and the network decide how bytes get grouped, based on timing, buffer sizes, and packet boundaries you don't control.

Make the server send three pieces:

```
socket.write('one')
socket.write('two')
socket.end('three')
```

On the client, log each chunk in hexadecimal and as text:

```
socket.on('data', chunk => {
  console.log(chunk.length, chunk.toString('hex'), JSON.stringify(chunk.toString()))
})
```

On loopback you'll often see all three writes arrive as a single chunk:

```
11 6f6e6574776f7468726565 "onetwothree"
```

Repeat the run. Add a small `setTimeout` between the writes, or run the pair over a real network, and the grouping changes: sometimes `"one"` then `"twothree"`, sometimes three separate events. Chunk boundaries are an implementation detail, not your protocol.

Why this bites people

The classic mistake is code that works on a laptop:

```
socket.on('data', chunk => {
  const command = chunk.toString() // assumes one event = one command
  handle(command)
})
```

This passes every local test, because small writes on loopback usually arrive whole. Then in production a command arrives split across two events, `JSON.parse()` throws on half a document, and you're debugging a ”random” crash that only shows up under load or on slow links.

The fix is **framing**: your protocol must define where one message ends and the next begins. The two common tools are a delimiter like `\n` and an explicit length prefix. That's what the next module

builds.

Never parse one `data` event as one complete command unless the protocol itself guarantees a fixed length you enforce. If you take one thing from this course, take this lesson.

Choose a listening address

Bind to loopback, one interface, or all interfaces and verify the resulting exposure.

The listening address decides which local interfaces accept connections. It's a security decision, not a formality.

Every machine has several addresses. `127.0.0.1` is loopback: only processes on the same machine can reach it. Your LAN address (something like `192.168.1.20`) is reachable by other devices on that network. `0.0.0.0` is the wildcard: it accepts IPv4 traffic reaching any interface. `::` is the IPv6 equivalent, and on many systems accepts IPv4 too.

Compare two explicit bindings:

```
server.listen(4000, '127.0.0.1')
// later, in an authorized lab only
server.listen(4000, '0.0.0.0')
```

If you omit the host argument entirely, Node listens on all interfaces. That default surprises people: a ”local test server” is suddenly reachable by everyone on the coffee shop wifi.

Verify the exposure

Inspect listeners after each run:

```
ss -lnt | grep 4000
# 127.0.0.1:4000   reachable from this machine only
# 0.0.0.0:4000     reachable on every IPv4 interface
```

The local address column tells you what the kernel is actually doing, regardless of what you think your code says. Trust the tool, not your memory.

To prove the difference, connect from another authorized device on the same network:

```
nc 192.168.1.20 4000
```

Against the loopback binding this fails. Against `0.0.0.0` it connects — unless a firewall blocks port 4000 in between. Test from another authorized device only when the firewall and network are understood; the result teaches you which layer is doing the protecting.

Choosing

My advice: bind to `127.0.0.1` by default. Widen the binding only when a specific consumer needs it, and you know who can reach the interface you're opening.

A common production pattern keeps the Node service on loopback with nginx or Caddy in front, so only the proxy is exposed and it handles TLS and access control.

Binding broadly is not authentication. Anyone who can reach the port can speak your protocol. Add firewall and application controls before exposing a service beyond loopback.

Check your understanding: tcp foundations

Check the commands, decisions, safety boundaries, and failure cases from tcp foundations.

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

Take this interactive quiz online

Message framing

Build a newline protocol, buffer partial input, parse JSON messages, and enforce size limits.

Design a newline protocol

Define one command per line with explicit encoding, replies, errors, and connection closure.

TCP gives you a byte stream with no message boundaries. The simplest fix is a **newline protocol**: one message per line, terminated by `\n`.

This is how a lot of real protocols work. SMTP, POP3, and HTTP/1.1 headers are all line-based text. Lines are easy to type, easy to log, and easy to debug with netcat.

A small text protocol still needs grammar. Decide the encoding, the commands, and the replies before writing any parser code. We will use UTF-8 lines ending in `\n`, commands `PING` and `ECHO text`, and numeric replies.

Write the protocol before the parser:

```
PING          -> 200 PONG\n
ECHO hello    -> 200 hello\n
anything else -> 400 unknown command\n
```

The numeric code plays the same role as HTTP status codes: a client can branch on `200` versus `400` without parsing the human-readable part after it.

Test the contract by hand

The whole point of a text protocol is that you can be the client. Test the examples with `nc 127.0.0.1 4000`:

```
nc 127.0.0.1 4000
PING
200 PONG
ECHO hello
200 hello
DELETE
400 unknown command
```

You type a line, the server answers with a line. The written contract tells both endpoints when one message ends: at the `\n`.

The limit you must set

Set a maximum line length, and make it part of the contract:

```
maximum line length: 4096 bytes
on violation: 400 line too long, then close
```

Here's why. Your server buffers bytes until it sees `\n`. A client must not be able to grow the server buffer forever by withholding a newline — without the limit, one shell one-liner piping `/dev/zero` into `nc` exhausts your server's memory.

One more decision while you're writing the grammar: what happens to `\r`? Clients like telnet send `\r\n` line endings. Either strip a trailing `\r` before parsing or document that bare `\n` is required. Leaving it ambiguous produces "works with my client, fails with yours" bugs that waste an afternoon.

Buffer partial lines

Accumulate chunks, extract every complete line, and keep the unfinished remainder for the next event.

The newline protocol is defined. Now the parser has to survive TCP's chunking: a line can arrive in pieces, and several lines can arrive together.

The parser keeps per-socket state, because each TCP connection can stop in the middle of a line. That state is one string: the bytes received so far that don't yet end in `\n`.

Add a line buffer:

```
let pending = ''

socket.on('data', chunk => {
  pending += chunk.toString('utf8')
  const lines = pending.split('\n')
  pending = lines.pop()

  for (const line of lines) handleLine(line)
})
```

Walk through it. Each chunk is appended to `pending`. `split('\n')` breaks the accumulated text into pieces; every piece except the last was terminated by a newline, so it's a complete line. The last piece is either an empty string (the chunk ended exactly at `\n`) or a partial line — `lines.pop()` puts it back into `pending` to wait for the rest.

Prove both cases work

Interactive netcat sends whole lines, so use a small script to control the chunking:

```
import net from 'node:net'

const c = net.createConnection({ host: '127.0.0.1', port: 4000 })
c.write('PI')
setTimeout(() => c.write('NG\nECHO hi\nEC'), 300)
setTimeout(() => c.write('HO bye\n'), 600)
c.on('data', data => process.stdout.write(data))
```

Send half a command, pause, then finish it. Send two commands together. The first write parks `PI` in `pending`. The second completes `PING`, delivers a whole `ECHO hi`, and parks `EC`. The third

completes `ECHO bye`. Both cases must produce exactly the intended replies — three, no extras, none missing.

Keep the buffer honest

Enforce the maximum line length here, before retaining more input:

```
if (pending.length > 4096) return socket.destroy()
```

Validate UTF-8 and length before retaining untrusted input. One subtlety: `chunk.toString('utf8')` can mangle a multi-byte character split across two chunks; `StringDecoder` from `node:string_decoder` handles that boundary if your protocol carries non-ASCII text.

And never share `pending` between sockets. Declare it inside the connection callback so each client gets its own. A module-level buffer mixes two clients' bytes into one garbled stream — a bug that only appears with concurrent connections, which is exactly when you least want to debug it.

Send length-prefixed messages

Frame binary or multiline data with an explicit length and parse it without reading past the message.

Newline framing breaks down when the payload can contain newlines: binary data, file contents, multiline text. A **length prefix** removes the problem. A length prefix lets the payload contain any byte, including newlines.

The sender writes a fixed-size header holding the payload size, then the payload itself. The receiver first reads the fixed-size length, then exactly that many payload bytes. No delimiter, no escaping.

Encode one four-byte length and payload:

```
const payload = Buffer.from('hello')
const frame = Buffer.alloc(4 + payload.length)
frame.writeUInt32BE(payload.length, 0)
payload.copy(frame, 4)
socket.write(frame)
```

`writeUInt32BE` stores the length big-endian, most significant byte first. That's **network byte order**, the convention wire protocols use for multi-byte integers. Both sides must agree: a big-endian writer paired with a little-endian reader turns length `5` into 83,886,080.

Capture the bytes and confirm:

```
console.log(frame.toString('hex'))
// 0000000568656c6c6f
```

The first four bytes represent `5` in network byte order, followed by `68 65 6c 6c 6f` — ”hello”.

Parse without reading past the message

The receiver accumulates a Buffer and loops. Build a parser that waits for the complete frame:

```
let pending = Buffer.alloc(0)

socket.on('data', chunk => {
  pending = Buffer.concat([pending, chunk])
```

```
  while (pending.length >= 4) {
    const size = pending.readUInt32BE(0)
    if (size > 1_000_000) return socket.destroy()
    if (pending.length < 4 + size) break
    handleMessage(pending.subarray(4, 4 + size))
    pending = pending.subarray(4 + size)
  }
})
```

The `while` loop matters: one chunk may contain several complete frames, and each must be handled. The `break` matters too: if the payload hasn't fully arrived, stop and wait for the next `data` event instead of consuming a partial frame.

The size check is not decoration. Reject impossible or excessive lengths before allocating memory or waiting for payload data. A single hostile 4-byte header claiming a 4 GB payload would otherwise make your server buffer forever for a message that never comes — pick a ceiling your protocol actually needs and destroy the connection past it.

Frame JSON messages

Send newline-delimited JSON, validate the parsed shape, and return errors without crashing the server.

Lines and length prefixes solve framing but leave the payload unstructured. JSON gives values structure — objects, strings, numbers — but does not frame a TCP stream. You need both.

Newline-delimited JSON combines them: one compact JSON document per line. It works because compact JSON never contains a literal newline — inside strings, JSON escapes them as `\n`. Framing comes from the newline, structure comes from JSON, and the line buffer you already built keeps working unchanged.

Only the line handler changes. Handle one parsed message:

```
function handleLine(line) {
  try {
    const message = JSON.parse(line)
    if (message.type !== 'echo' || typeof message.text !== 'string') {
      return socket.write('{"error":"invalid message"}\n')
    }
    socket.write(JSON.stringify({ text: message.text }) + '\n')
  } catch {
    socket.write('{"error":"invalid json"}\n')
  }
}
```

There are two distinct failure layers here. The `catch` handles bytes that aren't JSON at all. The `if` handles valid JSON with the wrong shape — a missing `type`, a numeric `text`. Both get a bounded, structured error reply, and the connection stays open.

The `try` is load-bearing. `JSON.parse()` throws on malformed input, and one bad line from one client must never crash a server with a hundred others connected. This is the single most common way naive TCP servers die.

Test all three cases

Send valid JSON, malformed JSON, and a valid object with the wrong shape:

```
nc 127.0.0.1 4000
{"type":"echo","text":"hi"}
{"text":"hi"}
{"type":"echo"}
{"error":"invalid message"}
not json at all
{"error":"invalid json"}
```

The first line you type gets echoed back as `{"text":"hi"}`. The wrong-shape object and the garbage line each get exactly one error reply. Every case should receive one bounded reply — never silence, never a crash, never two replies for one message.

Machine-readable errors pay off quickly: a client can branch on the `error` field the same way it branched on `400` in the text protocol.

Parsing proves syntax only. A message can be perfect JSON, the right shape, and still ask for something this client isn't allowed to do. Validate types, lengths, and authorization before acting.

Check your understanding: message framing

Check the commands, decisions, safety boundaries, and failure cases from message framing.

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

Take this interactive quiz online

Connection reliability

Handle timeouts, half-closes, resets, backpressure, and cleanup without leaking resources.

Add an idle timeout

Close connections that stop making progress and distinguish inactivity from a total request deadline.

A client can connect and send nothing forever. Each idle connection holds a file descriptor, buffer memory, and a slot in your per-client state. Enough of them and the server can't serve anyone — the cheapest denial of service there is.

An **idle timeout** bounds how long the socket remains inactive. In Node, `socket.setTimeout()` starts an inactivity timer that resets on every read or write.

Add a ten-second inactivity timeout:

```
socket.setTimeout(10_000)
socket.on('timeout', () => {
  socket.end('408 timeout\n')
})
```

Node does nothing on its own when the timer fires — it only emits the `timeout` event. You decide the response. Here `socket.end()` sends a final line and closes our side. If you'd rather not trust the peer to complete the close, follow up with `socket.destroy()` after a short grace period.

Verify it

Connect with netcat and wait:

```
nc 127.0.0.1 4000
# ...say nothing for ten seconds...
# 408 timeout
```

While waiting, open a second `nc` in another terminal and use it normally. Confirm the server closes the idle socket while continuing to accept and serve other clients. The timeout is per connection, not per server.

Idle is not the same as slow

The timer resets on any activity. A hostile client can send one byte every nine seconds and hold your connection open indefinitely while never completing a request — the slowloris pattern. An idle timeout alone doesn't catch it.

If your protocol needs a total request deadline, track it separately: record a timestamp when the request starts and enforce a maximum age regardless of activity. Idle timeout says "you went quiet". A deadline says "you took too long overall". Robust servers need both.

Picking the number

Choose a timeout from protocol needs, not from a random blog post. An interactive command protocol where humans type can justify 30 to 60 seconds. Machine-to-machine request/reply traffic can be far stricter.

A timeout that is too short can break slow but legitimate clients — a person testing with `nc`, a device on a congested link. Start generous, log how long real connections actually sit idle, then tighten with data.

Handle end, close, and errors

Distinguish a peer half-close, final socket closure, and an exceptional failure.

A TCP connection can finish several ways, and Node reports each with a distinct event. Handling all of them cleanly is the difference between a server that runs for months and one that slowly leaks state.

The `end` event means the readable side received **FIN** — the peer announced it will send no more data. TCP allows a **half-close**: the socket may still be writable, so you can send a final reply after the client finished sending. In Node that only happens if you create the server with `allowHalfOpen: true`; by default Node closes the writable side back automatically.

The `close` event means the handle is fully gone. It fires exactly once for every connection, however it ended. `error` carries a failure, and `close` follows it with `hadError` set to `true`.

Log the lifecycle:

```
socket.on('end', () => console.log('peer ended writes'))
socket.on('close', hadError => console.log('closed', { hadError }))
socket.on('error', error => console.error(error.code))
```

Watch each ending

Close a client normally — quit `nc` with Ctrl-C, which makes the kernel send FIN:

```
peer ended writes
closed { hadError: false }
```

Now force an abrupt reset. Have a test client call `socket.resetAndDestroy()`, which sends RST instead of FIN:

```
ECONNRESET
closed { hadError: true }
```

`ECONNRESET` also shows up in the wild when a peer crashes with data in flight or a middlebox kills the connection. Finally, stop the server while a client is connected and watch the client side see `end` — closure looks graceful from the other end too.

Compare event order across runs: `error` comes first when present, `end` appears only on graceful closes, and `close` is always last.

Clean up on close, not on end

Per-connection resources — timers, entries in a clients map, pending work — belong in the `close` handler:

```
socket.on('close', () => {
  clearTimeout(deadline)
  clients.delete(socket)
})
```

Cleanup timers and per-client state on close. Do not assume every connection reaches the happy-path `end` event: resets, crashes, and your own `socket.destroy()` calls all skip it. State you only release on `end` leaks until the process restarts, and `close` is the one event guaranteed to fire.

Respect backpressure

Stop writing when the socket buffer fills and resume only after the drain event.

A fast producer can outpace a slow receiver. Say your server streams a large file to a client on a slow link: you can read from disk far faster than TCP can deliver. Every byte the socket can't send yet has to wait somewhere.

That somewhere is Node's internal write queue. `socket.write()` never blocks — it queues the chunk and returns a boolean. `true` means keep going. `false` means the queued bytes crossed the stream's `highWaterMark` threshold (16 KiB by default for sockets). Node queues writes, but an unbounded queue consumes memory, and nothing stops you from ignoring the `false`.

Backpressure is the discipline of pausing the producer when the consumer falls behind.

Check the return value from `write()`:

```
if (!socket.write(chunk)) {
  source.pause()
  socket.once('drain', () => source.resume())
}
```

`source` is whatever produces the data — a file stream, another socket. When `write()` returns

`false`, pause it. The socket emits `drain` once its queue empties, and you resume.

If you're just connecting two streams, `source.pipe(socket)` — or better, `pipeline()` from `node:stream` — implements exactly this logic plus error handling. Write it by hand once so you know what pipe is doing for you.

See it happen

Make a deliberately slow client by pausing its reads. Throttle the client or pause its reads, then send a large authorized test stream from the server:

```
socket.on('data', () => {
  socket.pause()
  setTimeout(() => socket.resume(), 100)
})
```

Pausing stops Node from draining the kernel buffer, TCP's flow control fills the sender's window, and the server's queue backs up. Log every `write()` return value and every `drain`: you should observe write returning false, a pause, and repeated drain-and-resume cycles.

Now watch process memory during the transfer. With backpressure it stays flat. Comment out the `pause()` and it climbs with the size of the transfer — that's the production failure mode: one slow client, one loop writing at full speed, and the process dies with an out-of-memory error while the socket looked perfectly healthy.

Backpressure is flow control inside the process. It does not authorize unlimited payload size or connection count — those need explicit limits, coming next.

Limit connections and work

Bound clients, input size, queued output, and per-message work to keep one peer from exhausting the service.

Timeouts solve only one resource problem. A server also needs limits on simultaneous clients, buffered bytes, message rate, and expensive operations. Without them, one aggressive peer — or one buggy legitimate one — takes the service down for everyone.

Start with the connection count. Track active sockets:

```
const clients = new Set()

server.on('connection', socket => {
  if (clients.size >= 100) return socket.destroy()
  clients.add(socket)
  socket.on('close', () => clients.delete(socket))
})
```

The `Set` gives you an exact live count. Above the cap, `socket.destroy()` drops the newcomer immediately: no protocol reply, no allocated state. Rejecting cheaply matters — an overload defense that does expensive work per rejection is itself an attack surface.

The `close` handler keeps the count honest. Every accepted socket must remove itself on every exit path, which is why the previous lesson insisted on cleaning up in `close`.

Verify it: open several lab clients (a shell loop of `nc` calls works), watch the count rise, close them,

and confirm the set returns to zero after closure. If it doesn't, you have a leak that will silently hit the cap in production days later.

The other budgets

Connection count is one axis. A single client inside its one allowed connection can still hurt you:

```
// bound the outgoing queue per client
if (socket.writableLength > 1_000_000) socket.destroy()
```

Three budgets to set deliberately. Input size: the maximum line or frame length from the framing module. Output size: `socket.writableLength` tells you how many bytes are queued for a client that reads slowly or not at all. Work rate: count messages per connection per second and throttle or disconnect past a ceiling.

Each limit needs a number, and the number should come from measurement — typical real usage times a safety factor — not from guessing.

Add metrics for rejected connections. Even one log line per rejection tells you whether the cap is doing routine protection, you're under attack, or legitimate users are being turned away and the limit needs raising.

A process limit complements operating-system, firewall, and upstream controls; kernel backlog and file-descriptor limits sit underneath yours either way. Test failure behavior before production: hit the cap on purpose and confirm existing clients keep working while newcomers are refused.

Check your understanding: connection reliability

Check the commands, decisions, safety boundaries, and failure cases from connection reliability.

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

Take this interactive quiz online

UDP datagrams

Build UDP endpoints, preserve message boundaries, and add retries, identifiers, and duplicate handling.

Build a UDP server

Bind a loopback UDP socket, receive datagrams, and inspect sender address and port.

TCP gave you a connected byte stream. **UDP** gives you the opposite: independent messages called **datagrams**, delivered without a connection handshake. Each datagram arrives whole or not at all — no stream, no ordering guarantee, no automatic retransmission.

UDP delivers independent datagrams, and each `message` event includes exactly one datagram plus the remote endpoint that sent it.

Node exposes UDP through `node:dgram`. Create `udp-server.mjs`:

```
import dgram from 'node:dgram'

const socket = dgram.createSocket('udp4')
```

```
socket.on('message', (message, remote) => {
  console.log(message.toString(), remote)
})
socket.bind(41234, '127.0.0.1')
```

Notice what's missing compared to TCP: no `createServer`, no per-connection callback, no accepted socket. One socket receives datagrams from every sender. The `remote` object is how you tell senders apart — and where you'd send a reply.

`bind(41234, '127.0.0.1')` claims UDP port 41234 on loopback. UDP and TCP ports are separate namespaces, so a TCP listener on 41234 wouldn't conflict.

Verify it

Confirm the UDP listener:

```
ss -lnu | grep 41234
# UNCONN 0 0 127.0.0.1:41234 0.0.0.0:*
```

The state says `UNCONN` instead of `LISTEN` — UDP sockets don't listen for connections, they just receive whatever arrives.

Send a datagram with netcat in UDP mode:

```
echo hello | nc -u 127.0.0.1 41234
```

The server prints:

```
hello { address: '127.0.0.1', family: 'IPv4', port: 58239, size: 6 }
```

One send should produce one `message` event. That's the property UDP gives you that TCP doesn't: message boundaries are preserved. Six bytes sent (five letters plus the newline) arrive as one six-byte datagram — never split across events, never merged with the next one. No framing code needed.

The `port` in the remote info is the sender's ephemeral port, picked by its OS. Run the `nc` command twice and you'll see a different number each time.

One caution before you build on this. UDP source addresses can be spoofed in some networks — nothing in the protocol proves a datagram came from the address it claims. Do not treat a received datagram as authenticated identity; anything that changes state needs validation of its own.

Send a UDP datagram

Send one message, close the client after completion, and observe the absence of a handshake.

The UDP client is even smaller than the server. A UDP sender can transmit without first proving a receiver exists — there's no handshake, so there's nothing to wait for before sending.

Create `udp-client.mjs`:

```
import dgram from 'node:dgram'

const socket = dgram.createSocket('udp4')
const message = Buffer.from('hello')

socket.send(message, 41234, '127.0.0.1', error => {
  if (error) console.error(error)
```

```
  socket.close()
})
```

`socket.send()` takes the payload, the destination port, and the destination address. The callback fires when the datagram has been handed off. Success from `send()` means local handling completed — the kernel accepted the datagram for transmission. It says nothing about arrival, and even less about the receiving application processing it.

The `socket.close()` in the callback matters for a different reason: an open dgram socket keeps the Node event loop alive, so without it the client never exits.

Run it against a dead port

Run it with the server up:

```
node udp-client.mjs
# server terminal prints: hello { address: '127.0.0.1', ... }
```

Now stop the server and run the client again:

```
node udp-client.mjs
# ...nothing. No error. Exit code 0.
```

Compare what the sender can prove in each case: nothing differs from the sender's point of view. Contrast this with TCP, where connecting to a dead port failed fast with `ECONNREFUSED`. The UDP client reports success either way, because success only ever meant "sent".

Some systems generate an ICMP port-unreachable error when the destination port is closed, but you can't depend on receiving it, and firewalls often drop it. Design as if no error will ever come back.

Confirmation is your job

That's the deal you accept with UDP: fast, connectionless, and silent about failure. Applications needing confirmation must define acknowledgments, deadlines, and retry limits themselves. The receiver sends a reply datagram; the sender waits with a deadline and resends a bounded number of times:

```
send request -> wait up to 500 ms for reply
no reply     -> resend, at most 3 attempts
still nothing -> report failure to the caller
```

The next two lessons build exactly this: request identifiers so retries can be recognized, and explicit handling for loss and reordering.

Add request identifiers

Give UDP requests unique IDs so replies, retries, and duplicates can be matched safely.

Once a UDP client retries, a new problem appears: duplicates. A retry may cause the receiver to process the same logical request more than once. The original datagram might not be lost — just slow. Now both copies arrive, and if the request was "add 10 credits", you added twenty.

A **request ID** lets both sides recognize the operation. The client generates a unique ID per logical request and reuses the same ID on every retry of it. Replies carry the ID back, so the client can match replies to requests. The server remembers recently handled IDs so it can deduplicate.

Send a small JSON datagram:

```
const request = { id: crypto.randomUUID(), type: 'status' }
socket.send(JSON.stringify(request), 41234, '127.0.0.1')
```

`crypto.randomUUID()` gives 122 random bits, so collisions are not a practical concern. Sequence numbers work too, but need care to stay unique across client restarts.

On the server, return the same ID in the reply, and cache what you already handled:

```
const seen = new Map() // id -> { reply, at }

socket.on('message', (message, remote) => {
  const request = JSON.parse(message) // wrap in try/catch as in the framing module
  const cached = seen.get(request.id)
  if (cached) {
    return socket.send(cached.reply, remote.port, remote.address)
  }
  const reply = JSON.stringify({ id: request.id, ok: true })
  seen.set(request.id, { reply, at: Date.now() })
  socket.send(reply, remote.port, remote.address)
})
```

A duplicate now receives the cached reply instead of a second execution. This makes retries safe even when the operation itself isn't naturally repeatable.

Have the server cache recent completed IDs for a bounded time — a few retry windows is enough. Sweep entries older than, say, 60 seconds. An unbounded map is a slow memory leak that an attacker can accelerate by flooding you with fresh IDs.

Verify

Retry after a deadline: make the client resend the identical request object after 500 ms, then read both logs. The server should show one ”executed” and one ”answered from cache”. The client should tolerate receiving two identical replies and treat the second as noise — match by ID, act once.

An ID supports deduplication but does not authenticate the sender. A forged datagram can carry any ID it likes. Validate and authorize separately.

Design for loss and reordering

Test dropped, delayed, duplicated, and reordered datagrams before depending on UDP behavior.

UDP networks can lose, duplicate, or reorder messages. On loopback you'll almost never see any of it, which makes loopback a dangerously polite test environment. Code that assumes delivery works fine in the lab and fails quietly in the field.

You can't fix loss at the UDP layer. The application decides whether late data is useful and how much recovery is worth doing. A telemetry stream can shrug at 2% loss — the next reading supersedes the lost one anyway. A control command needs acknowledgment and retries. Neither answer is wrong; leaving the question unanswered is.

So make the lab hostile on purpose. Add a controlled random drop in the lab server:

```
if (Math.random() < 0.25) {
```

```
  console.log('dropped for lab')
  return
}
```

One in four requests now disappears, honestly logged so you can correlate.

Give messages sequence numbers so you can see what happened:

```
let seq = 0
setInterval(() => {
  const message = JSON.stringify({ seq: seq++, sentAt: Date.now() })
  socket.send(message, 41234, '127.0.0.1')
}, 200)
```

Send numbered messages with retry deadlines, using the request-ID machinery from the previous lesson. On the receiving side, track the highest `seq` seen and log anomalies:

```
if (message.seq <= highest) console.log('duplicate or out of order', message.seq)
```

Record duplicates and out-of-order arrivals over a few hundred messages. The drop code plus client retries will manufacture duplicates reliably. Reordering is rare on loopback, but real networks with multiple paths provide it, so the handling code must exist either way.

Then make the receiver's behavior explicit, in writing:

```
lost:      sender retries up to 3 times, 500 ms apart, then reports failure
duplicate: deduplicated by request ID
reordered: seq older than current state is discarded
```

Those three lines are the difference between a protocol and a hope. Any rule missing from that list gets defined later, by a production incident, at the worst possible time.

One boundary: do not use random fault injection in real traffic without a bounded experiment and approval. The drop snippet is for your lab server on loopback. In shared environments, injected failure is a change that needs the same review as any other.

Check your understanding: udp datagrams

Check the commands, decisions, safety boundaries, and failure cases from udp datagrams.

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

Take this interactive quiz online

Build and test a protocol

Specify commands, validate limits, support concurrent clients, inspect packets, and finish a small service.

Write the protocol contract

Document framing, encoding, commands, replies, errors, limits, timeouts, and closure before adding features.

This final module builds a small key-value service end to end. It starts where every protocol should

start: with a document, not with code.

A protocol is an agreement between independently running programs. Your server and client share an author today. They won't forever — someone rewrites the client in Go, a monitoring script starts probing the port, a teammate builds a dashboard. The only thing keeping the endpoints compatible is the observable behavior on the wire. Write the observable contract so either endpoint can be replaced.

Create a compact contract:

```
encoding: UTF-8
framing: one JSON object per newline
commands: set, get, delete, quit
maximum line: 4096 bytes
idle timeout: 30 seconds
errors: structured JSON with code and message
```

Every line answers a question that otherwise gets answered by accident. Framing: how do I know where a message ends? Limits: how much can I send? Timeout: how long may I stay silent? Errors: what does failure look like, so I can handle it in code?

For each command, add request and reply shapes with examples:

```
{"type":"set","key":"color","value":"blue"}  -> {"ok":true}
{"type":"get","key":"color"}                 -> {"ok":true,"value":"blue"}
{"type":"get","key":"missing"}               -> {"ok":false,"code":"not_found"}
```

Examples are the most-read part of any protocol document. Make them copy-pasteable straight into `nc`.

Test the document, not the code

Have a second client implementation use only this document. A colleague works, and so do you next week, writing a small script while deliberately not looking at the server source. Every question the implementer must answer by reading server code is a hole in the contract. Any ambiguity it finds belongs in the contract.

Classic holes: are keys case-sensitive? Which characters can a key contain? What's the maximum value size? What exactly happens on a line over 4096 bytes — an error reply, or an immediate close?

Do not let implementation accidents become undocumented protocol behavior. If your server happens to tolerate trailing whitespace and one client starts relying on it, that accident is now protocol you support forever. The contract decides; the code follows.

Validate before changing state

Parse syntax, validate shape and limits, authenticate, authorize, then perform one bounded operation.

A network message is untrusted input. By the time a line reaches your command handler it has survived framing and `JSON.parse` — and neither says anything about whether the request is safe to execute.

Treat message handling as a pipeline with distinct stages: parse syntax, validate shape and limits,

authenticate, authorize, then perform one bounded operation. Separate parsing from validation and side effects so malformed data cannot partially mutate state.

Validation goes first, in one place:

```
function validate(message) {
  if (typeof message.key !== 'string') return 'invalid_key'
  if (message.key.length > 256) return 'key_too_long'
  if (message.type === 'set' && typeof message.value !== 'string') return 'invalid_value'
  if (message.type === 'set' && message.value.length > 4096) return 'value_too_long'
  return null
}
```

Only after validation passes, dispatch. Use an explicit command switch:

```
switch (message.type) {
  case 'get':
    return readValue(message.key)
  case 'set':
    return writeValue(message.key, message.value)
  default:
    return sendError('unknown_command')
}
```

The explicit `switch` with a `default` is deliberate. The tempting shortcut — `handlers[message.type](message)` on a plain object — lets a client send `"type": "constructor"` or `"type": "toString"` and reach inherited properties you never meant to expose. Listing commands by hand keeps the attack surface exactly as large as your protocol.

Attack your own server

Send missing keys, oversized values, unknown commands, and repeated requests:

```
nc 127.0.0.1 4000
{"type":"set","key":"color"}
{"ok":false,"code":"invalid_value"}
{"type":"drop_everything"}
{"ok":false,"code":"unknown_command"}
```

Every invalid message should return a stable error without changing data. Stable means the same bad input always yields the same code — clients can branch on it and your tests can assert it.

The property this buys you is no partial mutation. A `set` with a valid key but an invalid value must not create the key, touch a counter, or leave any trace. Validate everything, then act once.

One rule with no exceptions: never construct shell commands or file paths directly from client values. `exec('cat ' + message.key)` or reading `./data/${message.key}` hands remote callers your filesystem — `../../etc/passwd` is the classic payload. If a client value must reach a path, match it against a strict allowlist pattern first.

Support concurrent clients

Keep connection state isolated and decide how shared data changes are serialized.

Node handles concurrent connections without threads: the event loop interleaves socket events, and

each callback runs to completion. Node can maintain many sockets, but shared state can still create ordering races. The bugs aren't corrupted memory — they're logical: two clients touching shared data in an order you didn't anticipate.

Two disciplines keep this manageable.

Isolate per-connection state

Everything belonging to one client — line buffer, auth state, timers — lives in the connection callback's scope. Per-socket buffers and authentication must never leak between clients:

```
const server = net.createServer(socket => {
  let pending = ''            // this client's partial line
  let authenticated = false   // this client's auth state
  // handlers close over these, one set per connection
})
```

Name and observe each connection:

```
const id = crypto.randomUUID()
console.log({ event: 'connected', id, remote: socket.remoteAddress })
socket.on('close', () => console.log({ event: 'closed', id }))
```

Tag every log line with the connection `id` and interleaved logs from many clients become readable per client. Without it, debugging three simultaneous conversations from one log stream is guesswork.

Decide how shared writes are ordered

The key-value store is shared by design. A synchronous handler is safe: it runs to completion before the next event. The race appears the moment a handler awaits between reading and writing:

```
case 'increment': {
  const current = store.get(message.key) ?? 0
  await audit(message)                 // another client's handler can run here
  store.set(message.key, current + 1)  // writes a stale value
}
```

Two concurrent increments read the same `current`, and one update is lost. The fixes are ordinary: read after the `await` instead of before, or add a sequence or queue for shared writes if order matters — a per-key promise chain is a compact queue.

Verify with interleaving

Open three clients, interleave commands, and verify replies return to the correct socket. Set a key on client A, read it from B and C, and issue overlapping writes. Check that no reply lands on the wrong connection and that the logs show three distinct IDs, each with a clean connect-to-close lifecycle.

Connection IDs belong in logs, not as proof of user identity. The ID says ”same TCP connection”, nothing more. Identity comes from authentication inside the protocol.

Capture and finish the service

Inspect the final protocol on loopback, run failure tests, and document how to start, stop, and remove it.

The service works. The last step is looking at it from outside: capture actual packets and confirm the bytes on the wire match the contract.

A packet capture verifies framing and timing outside both implementations. That independence is the point — if your client and server share the same framing bug, they agree with each other and every test passes anyway. tcpdump has no such loyalty. It shows what actually crossed the interface, and plaintext lab traffic will reveal the protocol exactly.

Capture only the lab port:

```
sudo tcpdump -ni lo0 port 4000
# Linux commonly uses: sudo tcpdump -ni lo port 4000
```

`-n` skips name resolution, `-i` selects the loopback interface (macOS calls it `lo0`, Linux `lo`), and `port 4000` filters to your service. Add `-A` to print payloads as ASCII and you'll see your JSON lines exactly as sent.

Run a session against the capture and read it: the three-way handshake (flags `S`, `S.`, then `.`), data segments carrying your frames with the `P` push flag, and the FIN exchange at the end. Check that segment payload lengths line up with your maximum line size and that the idle timeout closes the connection when it should.

The final test matrix

Run normal, fragmented, oversized, idle, and abrupt-close cases:

```
normal:       set/get/delete round trip -> contract replies
fragmented:   command split across writes -> one correct reply
oversized:    line beyond 4096 bytes -> error and close
idle:         silent connection -> closed at 30 s
abrupt close: kill the client mid-command -> server logs and cleans up
```

Save expected outcomes and a cleanup command with the project: how to start the service, how to stop it, and how to confirm the port is free again (`ss -lnt | grep 4000` should print nothing). A service you can't remove cleanly isn't finished.

Two boundaries as you leave the lab. Capture only traffic you own or are authorized to inspect — on shared systems tcpdump sees everyone's packets, and authorization is not optional. And everything in this course ran plaintext on loopback, which is exactly why the capture was so informative. Add TLS before sending secrets over a real network, because everyone on the path gets the same view tcpdump just gave you.

Check your understanding: build and test a protocol

Check the commands, decisions, safety boundaries, and failure cases from build and test a protocol.

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

Take this interactive quiz online

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