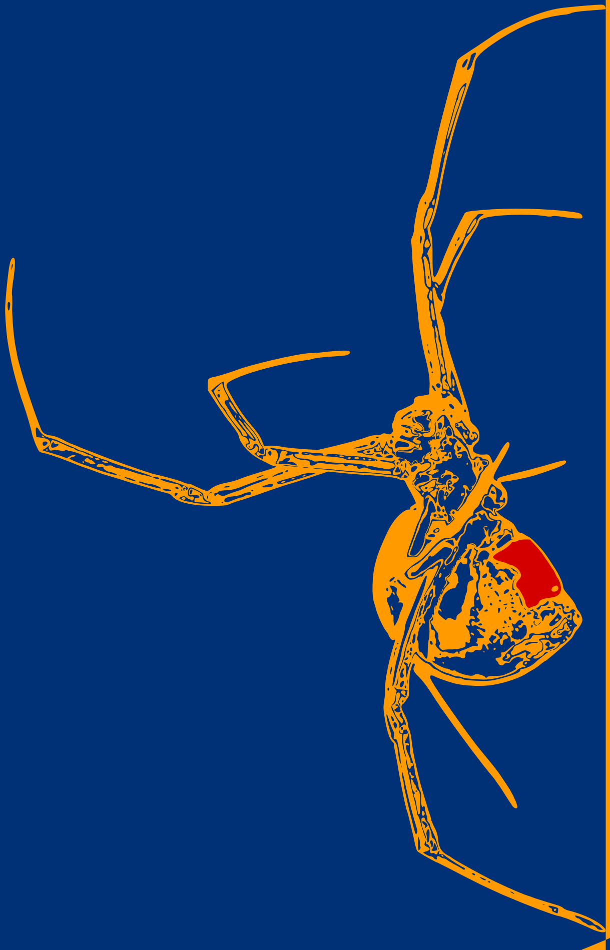
Project-Based Learning

Recreate critical tech
Master fundamentals

Build Your Own Web Server From Scratch in Node.JS

Network protocol
HTTP in detail
Concurrency
WebSocket



TypeScript

James Smith
build-your-own.org

Build Your Own
Web Server
From Scratch
in Node.JS

Learn network programming,
the HTTP protocol, and its applications
by coding your own Web Server.

James Smith

2024-02-02

build-your-own.org

Contents

CHAPTER 01

Introduction

1.1 Why Code a Web Server?

Most people use HTTP daily, but few understand its inner workings. This "Build Your Own X" book dives deep, teaching basics from scratch for a clearer understanding of the tools and tech we rely on.

Network Programming

The first step is to make programs talk over a network. This is also called *socket programming*. But socket programming is more than just gluing APIs together! It's easy to end up with half-working solutions if you skip the basics.

Protocols & Communication

In order to communicate over a network, the data sent over the network must conform to a specific format called a "protocol". Learn how to create or implement any network protocols by using HTTP as the target.

HTTP in Detail

You probably already know something about HTTP, such as URLs, different methods like GET and POST, response codes, various headers, and etc. But have you ever thought that you can create all the details from scratch, by your own code? It's not very complicated and it's rewarding.

1.2 Build Your Own X From Scratch

Why take on a build-your-own-X challenge? A few scenarios to consider:

- Students: Solidify learning, build portfolio, stand out in future careers.
- Developers: Master fundamentals beyond frameworks and tools.
- Hobbyists: Explore interests with flexible, extensible projects.

1.3 The Book

Project-Centric

Designed to guide you through your own web server implementation, the book follows a *step-by-step* approach, and each chapter builds on the previous one.

An outline of each step:

1. TCP echo server.
    - Socket primitives in Node.JS.
    - Event-based programming.
    - `async/await` and promises.
2. A simple messaging protocol.
    - Implementing a network protocol in TCP.
    - Working with byte streams and managing buffers.
3. Basic HTTP server.
    - HTTP semantics and syntax.
    - Generating dynamic content.
        - Async generators.
4. Core applications.
    - Serving static files.
        - File IO in Node.JS.
        - Programming tip: avoiding resource leaks.
        - Range requests.
    - Caching control.
    - Compression.
        - The stream and pipe abstraction.
5. WebSocket.
    - Message-based designs.
    - Concurrent programming.
        - Blocking queues.

Producing code is the easiest part and you'll also need to *debug* the code to make it work. So I have included some tips on this.

Discussions at the End of Chapter

Getting your own web server running is rewarding, but it should not be your only goal. There are many blog posts that show you a toy HTTP server. But how do you get beyond toys?

At the end of each chapter, there are discussions about:

- What's missing from the code? The gap between toys and the real thing, such as optimizations and applications.
- Important concepts beyond coding, such as event loops and backpressure. These are what you are likely to overlook.
- Design choices. Why stuff has to work that way? You can learn from both the good ones and the bad ones.
- Alternative routes. Where you can deviate from this book.

Node.js and TypeScript

The project uses Node.js without any dependencies, but many concepts are *language-agnostic*, so it's valuable for learners of any language.

Code samples use TypeScript with type annotations for readability, but the differences from JS are minor.

The code is mostly *data structures* + *functions*, avoiding fancy abstractions to maximize clarity.

Book Series

This is part of the "*Build Your Own X*" book series, which includes books on building your own Redis[1], database[2], and compiler[3].

https://build-your-own.org

[1] https://build-your-own.org/redis/
[2] https://build-your-own.org/database/
[3] https://build-your-own.org/compiler/

CHAPTER 02

HTTP Overview

2.1 Overview

As you may already know, the HTTP protocol sits above the TCP protocol. How TCP itself works in detail is not our concern; what we need to know about TCP is that it's a bidirectional channel for transmitting raw bytes — a carrier for other application protocols such as HTTP or SSH.

Although each direction of a TCP connection can operate independently, many protocols follow the request-response model. The client sends a request, then the server sends a response, then the client might use the same connection for further requests and responses.

```
 client        server
 ------        ------

| req1 |  ==>
         <==  | res1 |
| req2 |  ==>
         <==  | res2 |
         ...
```

An HTTP request or response consists of a *header* followed by an optional *payload*. The header contains the URL of the request, or the response code, followed by a list of *header fields*.

2.2 HTTP by Example

To get the hang of network protocols, let's start by making an HTTP request from the command line.

Run the netcat command:

```
nc example.com 80
```

The `nc` (netcat) command creates a TCP connection to the destination host and port, and then attaches the connection to stdin and stdout. We can now start typing in the terminal and the data will be transmitted:

```
GET / HTTP/1.0
Host: example.com
(empty line)
```

(Note the extra empty line at the end!)

We will get the following response:

```
HTTP/1.0 200 OK
Age: 525410
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Thu, 20 Oct 2020 11:11:11 GMT
Etag: "1234567890+gzip+ident"
Last-Modified: Thu, 20 Oct 2019 11:11:11 GMT
Vary: Accept-Encoding
Content-Length: 1256
Connection: close

<!doctype html>
<!-- omitted -->
```

Making HTTP requests from the command line is very easy. Let's take a look at the data and try to figure out what it means.

The first line of the request — `GET / HTTP/1.0` — contains the HTTP method `GET`, the URI `/`, and the HTTP version `1.0`. This is easy to figure out.

And the first line of the response — `HTTP/1.0 200 OK` — contains the HTTP version and the response code `200`.

Following the first line is a list of header fields in the format of `Key: value`. The request has a single header field — `Host` — which contains the domain name.

The response contains many fields, and their functions are not as obvious as the `Host` field. Many HTTP header fields are optional, and some are even useless. We will learn more about this in later chapters.

The response header is followed by the payload, which in our case is an HTML document. Payload and header are separated by an empty line. The GET request has no payload so it ends with an empty line.

This is just a simple example that you can play with from the command line. We will examine the HTTP protocol in more detail later.

2.3 The Evolution of HTTP

HTTP/1.0: The Prototype

The example above uses HTTP/1.0, which is an ancient version of HTTP. HTTP/1.0 doesn't support multiple requests over a single connection at all, and you need a new connection for every request. This is problematic because typical web pages depend on many extra resources, such as images, scripts, or stylesheets; the latency of the TCP handshake makes HTTP/1.0 very suboptimal.

HTTP/1.1 fixed this and became a practical protocol. You can try using the `nc` command to send multiple requests on the same connection by simply changing `HTTP/1.0` to `HTTP/1.1`.

HTTP/1.1: Production-Ready & Easy-to-Understand

This book will focus on HTTP/1.1, as it is still very popular and easy to understand. Even software systems that have nothing to do with the Web have adopted HTTP as the basis of their network protocol. When a backend developer talks about an "API", they likely mean an HTTP-based one, even for internal software services.

Why is HTTP so popular? One possible reason is that it can be used as a generic request-response protocol; developers can reply on HTTP instead of inventing their own protocols. This makes HTTP a good target for learning how to build network protocols.

HTTP/2: New Capacities

There have been further developments since HTTP/1.1. HTTP/2, related to SPDY, is the next iteration of HTTP. In addition to incremental refinements such as compressed headers, it has 2 new capacities.

- *Server push*, which is sending resources to the client before the client requests them.
- *Multiplexing* of multiple requests over a single TCP connection, which is an attempt to address *head-of-line blocking*.

With these new features, HTTP/2 is no longer a simple request-response protocol. That's why we start with HTTP/1.1 because it's simple enough and easy to understand.

HTTP/3: More Ambition

HTTP/3 is much larger than HTTP/2. It replaces TCP and uses UDP instead. So it needs to replicate most of the functionality of TCP, this TCP alternative is called QUIC. The motivations behind QUIC are userspace congestion control, multiplexing, and head-of-line blocking.

You can learn a lot by reading about these new technologies, but you may be overwhelmed by these concepts and jargon. So let's start with something small and simple: coding an HTTP/1.1 server.

2.4 Command Line Tools

You are introduced to the command line because it allows for quick testing and debugging. which is handy when coding your own HTTP server. Although you may want to store the request data in a file instead of typing it each time.

```
nc example.com 80 <request.txt
```

In practice, you may encounter some quirks of the `nc` command, such as not sending EOF, or multiple versions of `nc` with incompatible flags. You can use the modern replacement `socat` instead.

```
socat tcp:example.com:80 -
```

The `telent` command is also popular in tutorials.

```
telnet example.com 80
```

You can also use an existing HTTP client instead of manually constructing the request data. Try the `curl` command:

```
curl -vvv http://example.com/
```

Most sites support HTTPS alongside plaintext HTTP. HTTPS adds an extra protocol layer called "TLS" between HTTP and TCP. TLS is not plaintext, so you cannot use netcat to test an HTTPS server. But TLS still provides a byte stream like TCP, so you just need to replace netcat with a TLS client.

```
openssl s_client -verify_quiet -quiet -connect example.com:443
```

In the next chapter, we will do some actual coding.

CHAPTER 03

Code A TCP Server

Our first step is to get familiar with the socket API, so we will code a simple TCP server in this chapter.

3.1 TCP Quick Review

Layers of Protocols

Network protocols are divided into different layers, where the higher layer depends on the lower layer, and each layer provides different capacities.

```
 top
  /\    | App |    message or whatever
  ||    | TCP |    byte stream
  ||    | IP  |    packets
  ||    | ... |
bottom
```

The layer below TCP is the IP layer. Each IP packet is a message with 3 components:

- The sender's address.
- The receiver's address.
- The message data.

Communication with a packet-based scheme is not easy. There are lots of problems for applications to solve:

- What if the message data exceeds the capacity of a single packet?
- What if the packet is lost?
- Out-of-order packets?

To make things simple, the next layer is added on top of IP packets. TCP provides:

- Byte streams instead of packets.
- Reliable and ordered delivery.

A byte stream is simply an ordered sequence of bytes. A *protocol*, rather than the application, is used to make sense of these bytes. Protocols are like file formats, except that the total length is unknown and the data is read in one pass.

UDP is on the same layer as TCP, but is still packet-based like the lower layer. UDP just adds port numbers over IP packets.

TCP Byte Stream vs. UDP Packet

The key difference: boundaries.

- UDP: Each *read* from a socket corresponds to a single *write* from the peer.
- TCP: No such correspondence! Data is a continuous flow of bytes.

TCP simply has no mechanism for preserving boundaries.

1. TCP send buffer: This is where data is stored before transmission. Multiple writes are indistinguishable from a single write.
2. Data is encapsulated as one or more IP packets, IP boundaries have no relationship to the original write boundaries.
3. TCP receive buffer: Data is available to applications as it arrives.

The No. 1 beginner trap in socket programming is "concatenating & splitting TCP packets" because there is no such thing as "TCP packets". Protocols are required to interpret TCP data by imposing boundaries within the byte stream.

Byte Stream vs. Packet: DNS as an Example

To help you understand the implications of the byte stream, let's use the DNS protocol (domain name to IP address lookup) as an example.

DNS runs on UDP, the client sends a single request message and the server responds with a single response message. A DNS message is encapsulated in a UDP packet.

```
| IP header |         IP payload          |
            \............................../
             | UDP header |  UDP payload  |
                          \.............../
                           | DNS message |
```

Due to the drawbacks of packet-based protocols, e.g., the inability to use large messages, DNS is also designed to run on TCP. But TCP knows nothing about "message", so when sending DNS messages over TCP, a **2-byte length field** is prepended to each DNS message so that the server or client can tell which part of the byte stream is which message. This 2-byte length field is the simplest example of an *application protocol* on top of TCP. This protocol allows for multiple *application messages* (DNS) in a single TCP byte stream.

```
| len1 | msg1 | len2 | msg2 | ...
```

TCP Start with a Handshake

To establish a TCP connection, there should be a client and a server (ignoring the simultaneous case). The server waits for the client at a specific address (IP + port), this step is called *bind & listen*. Then the client can *connect* to that address. The "connect" operation involves a 3-step handshake (SYN, SYN-ACK, ACK), but this is not our concern because the OS does it transparently. After the OS completes the handshake, the connection can be *accepted* by the server.

TCP is Bidirectional & Full-Duplex

Once established, the TCP connection can be used as a bi-directional byte stream, with 2 channels for each direction. Many protocols are request-response like HTTP/1.1, where a peer is either sending a request/response or receiving a response/request. But TCP isn't restricted to this mode of communication. Each peer can send and receive at the same time (e.g. WebSocket), this is called *full-duplex* communication.

TCP End with 2 Handshakes

A peer tells the other side that no more data will be sent with the FIN flag, then the other side ACKs the FIN. The remote application is notified of the termination when reading from the channel.

Each direction of channels can be terminated independently, so the other side also performs the same handshake to *fully* close the connection.

3.2 Socket Primitives

The socket API comes in different shapes in different languages and libraries. You are likely to get confused if you jump into the API documentation without knowing the basics.

Applications Refer to Sockets by Opaque OS Handles

When you create a TCP connection, the connection is managed by your operating system, and you use the socket handle to refer to the connection in the socket API. In Linux, a socket handle is simply a file descriptor (fd). In Node.js, socket handles are wrapped into JS objects with methods on them.

Any OS handle must be closed by the application to terminate the underlying resource and recycle the handle.

Listening Socket & Connection Socket

A TCP server listens on a particular address (IP + port) and accepts client connections from that address. The listening address is also represented by a socket handle. And when you accept a new client connection, you get the socket handle of the TCP connection.

Now you know that there are 2 types of socket handles.

1. Listening sockets. Obtained by *listening* on an address.
2. Connection sockets. Obtained by *accepting* a client connection from a listening socket.

End of Transmission

Send and receive are also called *read* and *write*. For the write side, there are ways to tell the peer that no more data will be sent.

- *Closing* a socket terminates a connection and causes the TCP FIN to be sent. Closing a handle of any type also recycles the handle itself. (Once the handle is gone, you cannot do anything with it.)
- You can also *shutdown* your side of the transmission (also send FIN) while still being able to receive data from the peer; this is called a *half-open* connection, more on this later.

For the read side, there are ways to know when the peer has ended the transmission (received FIN). The end of transmission is often called the *end of file* (EOF).

List of Socket Primitives

In summary, there are several socket primitives that you need to know about.

- Listening socket:
  - bind & listen
  - accept
  - close
- Connection socket:
  - read
  - write
  - close

3.3 Socket API in Node.js

We will introduce the socket API with a small exercise: a TCP server that reads data from clients and writes the same data back. This is called an “echo server”.

Step 1: Create A Listening Socket

All the networking stuff is in the **net** module.

```
import * as net from "net";
```

Different types of sockets are represented as JS objects. The **net.createServer()** function creates a listening socket whose type is **net.Server**. **net.Server** has a **listen()** method to bind and listen on an address.

```
let server = net.createServer();
server.listen({host: '127.0.0.1', port: 1234});
```

Step 2: Accept New Connections

The next thing is the *accept* primitive for getting new connections. Unfortunately, there is no **accept()** function that simply returns a connection.

Here we need some background knowledge about IO in JS: There are 2 styles of handling IO in JS, the first style is using callbacks; you request something to be done and register a callback with the runtime, and when the thing is done, the callback is invoked.

```
function newConn(socket: net.Socket): void {
    console.log('new connection', socket.remoteAddress, socket.remotePort);
    // ...
}

let server = net.createServer();
server.on('connection', newConn);
server.listen({host: '127.0.0.1', port: 1234});
```

In the above code listing, **server.on('connection', newConn)** registers the callback function **newConn**. The runtime will automatically perform the *accept* operation and invoke the callback with the new connection as an argument of type **net.Socket**. This callback is registered once, but will be called for each new connection.

Step 3: Error Handling

The `'connection'` argument is called an *event*, which is something you can register callbacks on. There are other events on a listening socket. For example, there is the `'error'` event, which is invoked when an error occurs.

```
server.on('error', (err: Error) => { throw err; });
```

Here we simply throw the exception and terminate the program. You can test this by running 2 servers on the same address and port, the second server will fail.

As this book is not a manual, we will not list everything here. Read the Node.js documentation[1] to find out other potentially useful events.

Step 4: Read and Write

Data received from the connection is also delivered via callbacks. The relevant events for reading from a socket are the `'data'` event and the `'end'` event. The `'data'` event is invoked whenever data arrives from the peer, and the `'end'` event is invoked when the peer has ended the transmission.

```
    socket.on('end', () => {
        // FIN received. The connection will be closed automatically.
        console.log('EOF.');
    });
    socket.on('data', (data: Buffer) => {
        console.log('data:', data);
        socket.write(data); // echo back the data.
    });
```

The `socket.write()` method sends data back to the peer.

Step 5: Close The Connection

The `socket.end()` method ends the transmission and closes the socket. Here we call `socket.end()` when the data contains the letter “q” so we can easily test this scenario.

[1] https://nodejs.org/api/net.html#class-netserver

```
    socket.on('data', (data: Buffer) => {
        console.log('data:', data);
        socket.write(data); // echo back the data.

        // actively closed the connection if the data contains 'q'
        if (data.includes('q')) {
            console.log('closing.');
            socket.end();   // this will send FIN and close the connection.
        }
    });
```

When the transmission is ended from either side, the socket is automatically closed by the runtime. There is also the `'error'` event on `net.Socket` that reports IO errors. This event also causes the runtime to close the socket.

Step 6: Test It

Here is the complete code for our echo server.

```
import * as net from "net";

function newConn(socket: net.Socket): void {
    console.log('new connection', socket.remoteAddress, socket.remotePort);

    socket.on('end', () => {
        // FIN received. The connection will be closed automatically.
        console.log('EOF.');
    });
    socket.on('data', (data: Buffer) => {
        console.log('data:', data);
        socket.write(data); // echo back the data.

        // actively closed the connection if the data contains 'q'
        if (data.includes('q')) {
            console.log('closing.');
            socket.end();   // this will send FIN and close the connection.
        }
    });
}
```

```
let server = net.createServer();
server.on('error', (err: Error) => { throw err; });
server.on('connection', newConn);
server.listen({host: '127.0.0.1', port: 1234});
```

Start the echo server by running `node --enable-source-maps echo_server.js`. And test it with the `nc` or `socat` command.

3.4 Discussion: Half-Open Connections

Each direction of a TCP connection is ended independently, and it is possible to make use of the state where one direction is closed and the other is still open; this unidirectional use of TCP is called TCP half-open. For example, if peer A half-closes the connection to peer B:

- A cannot send any more data, but can still receive from B.
- B gets EOF, but can still send to A.

Not many applications make use of this. Most applications treat EOF the same way as being fully closed by the peer, and will also close the socket immediately.

The socket primitive for this is called *shudown*[2]. Sockets in Node.js do not support half-open by default, and are automatically closed when either side sends or receives EOF. To support TCP half-open, an additional flag is required.

```
let server = net.createServer({allowHalfOpen: true});
```

When the `allowHalfOpen` flag is enabled, you are responsible for closing the connection, because `socket.end()` will no longer close the connection, but will only send EOF. Use `socket.destroy()` to close the socket manually.

3.5 Discussion: The Event Loop & Concurrency

JS Code Runs Within the Event Loop

As you can see, callbacks are needed to do anything in our echo server. This is how an *event loop* works. It's a mechanism of the Node.js runtime that is invisible to the programmer. The runtime does something like this:

[2] https://man7.org/linux/man-pages/man2/shutdown.2.html

```
// pseudo code!
while (running) {
    let events = wait_for_events(); // blocking
    for (let e of events) {
        do_something(e);    // may invoke callbacks
    }
}
```

The runtime polls for IO events from the OS, such as a new connection arriving, a socket becoming ready to read, or a timer expiring. Then the runtime reacts to the events and invokes the callbacks that the programmer registered earlier. This process repeats after all events have been handled, thus it's called the *event loop*.

JS Code and Runtime Share a Single OS Thread

The event loop is *single-threaded*; execution is either on the runtime code or on the JS code (callbacks or the main program). This works because when a callback returns, or `await`s, control is back to the runtime, so the runtime can emit events and schedule other tasks. This implies that **any JS code is expected to finish in a short time** because the event loop is halted when executing JS code.

Concurrency in Node.JS is Event-Based

To help you understand the implication of the event loop, let's now consider *concurrency*. A server can have multiple connections simultaneously, and each connection can emit events.

While an event handler is running, the single-threaded runtime cannot do anything for the other connections until the handler returns. The longer you process an event, the longer everything else is delayed.

3.6 Discussion: Asynchronous vs. Synchronous

Blocking & Non-Blocking IO

It's vital to avoid staying in the event loop for too long. One way to cause such trouble is to run CPU-intensive code. This can be solved by …

- Voluntarily yield to the runtime.
- Moving the CPU-intensive code out of the event loop via multi-threading or multi-processing.

These topics are beyond the scope of this book, and our primary concern is IO.

The OS provides both blocking mode and non-blocking mode for network IO.

- In blocking mode, the calling OS thread blocks until the result is ready.
- In non-blocking mode, the OS immediately returns if the result is not ready (or is ready), and there is a way to be notified of readiness (for event loops).

The Node.JS runtime uses non-blocking mode because blocking mode is incompatible with event-based concurrency. The only blocking operation in an event loop is polling the OS for more events when there is nothing to do.

IO in Node.js is Asynchronous

Most Node.js library functions related to IO are either callback-based or promise-based. Promises can be viewed as another way to manage callbacks. These are also described as *asynchronous*, meaning that the result is delivered via a callback. These APIs do not block the event loop because the JS code doesn't wait for the result; instead, the JS code returns to the runtime, and when the result is ready, the runtime invokes the callback to continue your program.

The opposite is the *synchronous* API, which blocks the calling OS thread to wait for the result. For example, let's take a look at the documentation of the `fs` module, file APIs are available in all 3 types[3].

```
// promise
filehandle.read([options]);
// callback
fs.read(fd[, options], callback);
// synchronous, do not use!
fs.readSync(fd, buffer[, options]);
```

The synchronous API is what you do NOT use in network applications since it blocks the event loop. They exist for some simple use cases (like scripting) that do not depend on the event loop at all.

Event-Based Programming Beyond Networking

IO is more than disk files and networking. In GUI systems, user input from the mouse and keyboard is also IO. And event loops are not unique to the Node.js runtime; Web browsers and all other GUI applications also use event loops under the hood. You can transfer your experience in GUI programming to network programming and vice versa.

[3] https://nodejs.org/api/fs.html

3.7 Discussion: Promise-Based IO

As we mentioned before, there is another style of writing IO code. The alternative style uses **`Promises`** instead of callbacks. The advantage of promise-based APIs is that you can **`await`** on them and get the result, thus avoiding breaking your program into tiny callbacks that scattered all over the place.

A hypothetical promised-based API for the *accept* primitive looks like this:

```
// pseudo code!
while (running) {
    let socket = await server.accept();
    newConn(socket);    // no `await` on this
}
```

And the hypothetical API for the *read* and *write* primitive looks like this:

```
// pseudo code!
async function newConn(socket) {
    while (true) {
        let data = await socket.read();
        if (!data) {
            break;  // EOF
        }
        await socket.write(data);
    }
}
```

The above pseudo code appears to be synchronous, but without blocking the event loop. Although the advantage may not be clear at this point, since our program is very simple.

Some Node.js APIs, but not all of them, are available in both callback-based and promise-based styles. However, with some effort, callback-based APIs can be converted to promised-based ones, as we will see in the next chapter.

CHAPTER 04

Promises and Events

In this chapter, we will implement the echo server again, but using the promise-based style which is the basis for the rest of the book. This decision will be explained later.

4.1 Introduction to `async` and `await`

Let's take a quick tour of `async/await` and the `Promise` type in JS, in case you are not already familiar with them.

Using Callbacks

An example of a callback-based API. The application logic is continued in a callback function.

```
function my_app() {
    do_something_cb((err, result) => {
        if (err) {
            // fail.
        } else {
            // succuss, use the result.
        }
    });
}
```

Using Promises and `await`

An example of using `await` on a promise. The application logic continues in the same `async` function.

```
function do_something_prmoise(): Promise<T>;

async function my_app() {
    try {
        const result: T = await do_something_prmoise();
    } catch (err) {
        // fail.
    }
}
```

Advantage: The application logic is not broken into multiple functions.

Creating Promises

An example of creating promises: converting a callback-based API to promise-based.

```
function do_something_prmoise() {
    return new Promise<T>((resolve, reject) => {
        do_something_cb((err, result) => {
            if (err) {
                reject(err);
            } else {
                resolve(result);
            }
        });
    });
}
```

Callbacks are unavoidable in JS. When creating a promise object, an *executor* callback is passed as an argument to receive 2 more callbacks:

- `resolve()` causes the `await` statement to return a value.
- `reject()` causes the `await` statement to throw an exception.

You must call one of them when the result is available or the operation has failed. This may happen outside the executor function, so you may need to store these callbacks somewhere.

Terminology for promises:

- **Fulfilled**: `resolve()` called.
- **Rejected**: `reject()` called.
- **Settled**: Either fulfilled or rejected.
- **Pending**: Not settled.

4.2 Understanding `async` and `await`

Normal Functions Yield to the Runtime by `return`

There are 2 types of JS functions: normal functions and `async` functions. Normal functions execute from start to `return` (either explicitly or implicitly). Since the JS runtime is single-threaded and event-based, you cannot do blocking IOs in JS, instead you register callbacks for the completion of IOs, and then the JS code ends. Once back in the runtime, the runtime can poll for events and invoke callbacks, that's the event loop we talked about earlier!

`async` Functions Yield to the Runtime by `await`

Initially, the `Promise` type is just a way to manage callbacks. It allows chaining multiple callbacks without too many nested functions. However, we will not bother with this use of promises because of the addition of async functions.

Unlike normal functions, async functions can return to the runtime in the middle of execution; this happens when you use the `await` statement on a promise. And when the promise is settled, execution of the async function *resumes* with the result of the promise. This is a superior coding experience because you can write *sequential* IO code in the *same* function without being interrupted by callbacks.

Calling `async` Functions Start New Tasks

Invoking an async function results in a promise that settles itself when the async function returns or throws. You can `await` on it like a normal promise, but if you don't, the async function will still be scheduled by the runtime. This is similar to starting a thread in multi-threaded programming. But all JS code shares a single OS thread, so a better word to use is *task*.

A list of ways to start tasks in different environments:

- In JS, you start background tasks by not waiting for promises.
- In Go, you use the `go` statement.
- In Python, `async/await` is similar to JS, except that you have to run an event loop yourself, as it's not a language built-in.
- In environments without an event loop, you can start OS threads instead of user-level tasks.

4.3 From Events To Promises

The `net` module doesn't provide a promise-based API, so we have to implement the hypothetical API from the last chapter.

```
function soRead(conn: TCPConn): Promise<Buffer>;
function soWrite(conn: TCPConn, data: Buffer): Promise<void>;
```

Step 1: Analyze The Solution

The `soRead` function returns a promise which is resolved with socket data. It depends on 3 events.

1. The `'data'` event fulfills the promise.
2. While reading a socket, we also need to know if the EOF has occurred. So the `'end'` event also fulfills the promise. A common way to indicate the EOF is to return zero-length data.
3. There is also the `'error'` event, we want to reject the promise when this happens, otherwise, the promise hangs forever.

To resolve or reject the promise from these events, the promise has to be stored somewhere. We will create the `TCPConn` wrapper object for this purpose.

```
// A promise-based API for TCP sockets.
type TCPConn = {
    // the JS socket object
    socket: net.Socket;
    // the callbacks of the promise of the current read
    reader: null|{
        resolve: (value: Buffer) => void,
        reject: (reason: Error) => void,
    };
};
```

The promise's `resolve` and `reject` callbacks are stored in the `TCPConn.reader` field.

Step 2: Handle the 'data' Event

Let's try to implement the `'data'` event now. Here we have a problem: the `'data'` event is emitted whenever data arrives, but the promise only exists when the program is reading from the socket. So there must be a way to control when the `'data'` event is ready to fire.

```
socket.pause();      // pause the 'data' event
socket.resume();     // resume the 'data' event
```

With this knowledge, we can now implement the `soRead` function.

```
// create a wrapper from net.Socket
function soInit(socket: net.Socket): TCPConn {
    const conn: TCPConn = {
        socket: socket, reader: null,
    };
    socket.on('data', (data: Buffer) => {
        console.assert(conn.reader);
        // pause the 'data' event until the next read.
        conn.socket.pause();
        // fulfill the promise of the current read.
        conn.reader!.resolve(data);
        conn.reader = null;
    });
    return conn;
}

function soRead(conn: TCPConn): Promise<Buffer> {
    console.assert(!conn.reader);   // no concurrent calls
    return new Promise((resolve, reject) => {
        // save the promise callbacks
        conn.reader = {resolve: resolve, reject: reject};
        // and resume the 'data' event to fulfill the promise later.
        conn.socket.resume();
    });
}
```

Since the `'data'` event is paused until we read the socket, the socket should be paused by default after it is created. There is a flag to do this.

```
const server = net.createServer({
    pauseOnConnect: true,   // required by `TCPConn`
});
```

Step 3: Handle the 'end' and 'error' Event

Unlike the `'data'` event, the `'end'` and `'error'` events cannot be paused and are emitted as they happen. We can handle this by storing them in the wrapper object and checking them in `soRead`.

```
// A promise-based API for TCP sockets.
type TCPConn = {
    // the JS socket object
    socket: net.Socket;
    // from the 'error' event
    err: null|Error;
    // EOF, from the 'end' event
    ended: boolean;
    // the callbacks of the promise of the current read
    reader: null|{
        resolve: (value: Buffer) => void,
        reject: (reason: Error) => void,
    };
};
```

If there is a current reader promise, resolve or reject it.

```
// create a wrapper from net.Socket
function soInit(socket: net.Socket): TCPConn {
    const conn: TCPConn = {
        socket: socket, err: null, ended: false, reader: null,
    };
    socket.on('data', (data: Buffer) => {
        // omitted ...
    });
    socket.on('end', () => {
        // this also fulfills the current read.
        conn.ended = true;
        if (conn.reader) {
            conn.reader.resolve(Buffer.from(''));   // EOF
            conn.reader = null;
        }
    });
    socket.on('error', (err: Error) => {
        // errors are also delivered to the current read.
        conn.err = err;
        if (conn.reader) {
            conn.reader.reject(err);
            conn.reader = null;
        }
    });
    return conn;
}
```

Events that happened before **soRead** are stored and checked.

```
// returns an empty `Buffer` after EOF.
function soRead(conn: TCPConn): Promise<Buffer> {
    console.assert(!conn.reader);   // no concurrent calls
    return new Promise((resolve, reject) => {
        // if the connection is not readable, complete the promise now.
        if (conn.err) {
            reject(conn.err);
            return;
        }
        if (conn.ended) {
            resolve(Buffer.from(''));   // EOF
            return;
        }

        // save the promise callbacks
        conn.reader = {resolve: resolve, reject: reject};
        // and resume the 'data' event to fulfill the promise later.
        conn.socket.resume();
    });
}
```

Step 4: Write to Socket

The **socket.write** method accepts a callback to notify the completion of the write, so the conversion to promise is trivial.

```
function soWrite(conn: TCPConn, data: Buffer): Promise<void> {
    console.assert(data.length > 0);
    return new Promise((resolve, reject) => {
        if (conn.err) {
            reject(conn.err);
            return;
        }

        conn.socket.write(data, (err?: Error) => {
            if (err) {
                reject(err);
            } else {
                resolve();
            }
        });
    });
}
```

There is also the `'drain'` event[1] in the Node.js documentation which can be used for this task. Node.js libraries often give you multiple ways to do the same thing, you can just choose one and ignore the others.

4.3 Using `async` and `await`

Let's return to the echo server implementation. In order to use `await` on the promise-based API, the handler for new connections (`newConn`) becomes an `async` function.

```
async function newConn(socket: net.Socket): Promise<void> {
    console.log('new connection', socket.remoteAddress, socket.remotePort);
    try {
        await serveClient(socket);
    } catch (exc) {
        console.error('exception:', exc);
    } finally {
        socket.destroy();
    }
}
```

We also wrapped our code in a try-catch block because the `await` statement can throw exceptions when rejected. Although you may want to actually handle errors in production code instead of using a catch-all exception handler.

[1] https://nodejs.org/api/net.html#event-drain

```
// echo server
async function serveClient(socket: net.Socket): Promise<void> {
    const conn: TCPConn = soInit(socket);
    while (true) {
        const data = await soRead(conn);
        if (data.length === 0) {
            console.log('end connection');
            break;
        }

        console.log('data', data);
        await soWrite(conn, data);
    }
}
```

The code to use the socket now becomes straightforward. There are no callbacks to interrupt the application logic.

Note that the `newConn` async function is not `await`ed anywhere. It is simply invoked as a callback of the listening socket. This means that multiple connections are handled concurrently.

Exercise for the reader: convert the "accept" primitive to promise-based.

```
type TCPListener = {
    socket: net.Socket;
    // ...
};

function soListen(...): TCPListener;
function soAccept(listener: TCPListner): Promise<TCPConn>;
```

4.5 Discussion: Backpressure

Waiting for Socket Writes to Complete?

Our new echo server has a major difference — we now wait for `socket.write()` to complete. But what does the "completion of the write" mean? And why do we have to wait for it?

To answer the question, `socket.write()` is completed when the data is submitted to the OS, but a new question arises, why the data cannot be submitted to the OS immediately. This question actually goes deeper than network programming itself.

Producers are Bottlenecked by Consumers

Wherever there is asynchronous communication, there are queues or buffers that connect producers to consumers. Queues and buffers in our physical world are bounded in size and cannot hold an infinite amount of data. One problem with asynchronous communication is that **what happens when the producer is producing faster than the consumer is consuming?** There must be a mechanism to prevent the queue or buffer from overflowing. This mechanism is often called *backpressure* in network applications.

Backpressure in TCP: Flow Control

Backpressure in TCP is known as *flow control*.

- The consumer's TCP stack stores incoming data in a receive buffer for the application to consume.
- The amount of data the producer's TCP stack can send is bounded by a *window* known to the producer's TCP stack, and it will *pause* sending data when the window is full.
- The consumer's TCP stack manages the window; when the app drains from the receive buffer, it *moves* the window forward and notifies the producer's TCP stack to *resume* sending.

The effect of flow control: TCP can pause and resume transmission so that the consumer's receive buffer is bounded.

```
                               flow ctrl     bounded!
|producer| ==> |send buf| ===========> |recv buf| ==> |consumer|
    app             OS          TCP           OS             app
```

TCP flow control should not be confused with TCP congestion control, which also controls the window.

Backpressure Between Applications & OS

This nice mechanism needs to be implemented not only in TCP, but also in applications. Let's focus on the producer side. The application produces data and submits it to the OS, the data goes to the send buffer, and the TCP stack consumes from the send buffer and transmits the data.

```
             write()  may block!
|producer| ========> |send buf| =====> ...
    app                  OS         TCP
```

How does the OS prevent the send buffer from overflowing? Simple, the application cannot write more data when the buffer is full. Now the application is responsible for throttling itself from overproducing, because the data has to go somewhere, but memory is finite.

If the application is doing blocking IO, the call will block when the send buffer is full, so backpressure is effortless. However, this is not the case when coding in JS with an event loop.

Unbounded Queues are Footguns

We can now answer the question: why wait for writes to complete? Because while the application is waiting, it cannot produce! The `socket.write()` will always succeed even if the runtime cannot submit more data to the OS due to a full send buffer, but the data has to go somewhere, it goes to an unbounded internal queue in the runtime, which is a footgun that can cause unbounded memory usage.

```
           write()    unbounded!    event loop
|producer| ======> |internal queue| =========> |send buf| =====> ...
    app                 Node.js                    OS      TCP
```

Taking our old echo server as an example, the server is both a producer and a consumer, as is the client. If the client produces data faster than the client consumes the echoed data (or the client does not consume any data at all), the server's memory will grow indefinitely if the server does not wait for writes to complete.

Backpressure should exist in any system that connects producers to consumers. A rule of thumb is to look for unbounded queues in software systems, as they are a sign of the lack of backpressure.

4.6 Discussion: Events and Ordered Execution

Another difference from the old one is the use of `socket.pause()`. You can now understand why this is essential, because it is used to implement backpressure.

There is another reason to pause the `'data'` event. In callback-based code, when the event handler returns, the runtime can fire the next `'data'` event if it is not paused. The problem is that **the completion of the event callback doesn't mean the completion of the event handling** – the handling can continue with further callbacks. And the interleaved handling can cause problems, considering that the data is an ordered sequence of bytes!

This situation is called a *race condition*, and is a class of problems related to concurrency. In this situation, unwanted concurrency is introduced.

4.7 Conclusion: Promise vs. Callback

Following the discussions above, we can now explain why we switched to the promise-based API, because there are advantages.

- If you stick to promises and `async/await`, it's harder to create the kind of race conditions described above because things happen in order.
- With callback-based code, it's not only harder to figure out the order of code execution, it's also harder to control the order. In short, callbacks are harder to read and more error-prone to write.
- Backpressure is naturally present when using the promse-based style. This is similar to coding with blocking IO (which you can't do in Node.js).

We have learned the basics of the socket API. Let's move on to the next topic: protocol.

CHAPTER 05

A Simple Network Protocol

5.1 Message Echo Server

Parsing HTTP is a non-trivial amount of work, so let's take a smaller step first. We will implement a simpler protocol to illustrate the most important function of a protocol: splitting byte stream into messages.

Our protocol consists of messages separated by `'\n'` (the newline character). The server reads messages and sends back replies using the same protocol.

- If the client sends `'quit'`, reply with `'Bye.'` and close the connection.
- Otherwise, echo the message back with the prefix `'Echo: '`.

```
client      server
------      ------

msg1\n  ==>
       <==  Echo: msg1\n
msg2\n  ==>
       <==  Echo: msg2\n
quit\n  ==>
       <==  Bye.\n
```

5.2 Dynamic Buffers

The Need for Dynamic Buffers

Messages do not come from the socket by themselves, we need to store the incoming data in a buffer, then we can try to split messages from it.

A `Buffer` object in Node.JS is a fixed-size chunk of binary data. It cannot grow by appending data. What we can do is to concatenate 2 buffers into a new buffer.

```
buf = Buffer.concat([buf, data]);
```

However, this is an anti-pattern that can lead to O(n^2) algorithm complexity.

```
// pseudo code! bad example!
while (need_more_data()) {
    buf = Buffer.concat([buf, get_some_data()]);
}
```

Each time you append new data by concatenation, the old data is copied. To amortize the cost of copying, *dynamic arrays* are used:

- C++: **`std::vector`** and **`std::string`**.
- Python: **`bytearray`**.
- Go: slice.

Coding a Dynamic Buffer

The buffer has to be larger than needed for the appended data to use the extra capacity to amortize the copying. The **`DynBuf`** type stores the actual data length.

```typescript
// A dynamic-sized buffer
type DynBuf = {
    data: Buffer,
    length: number,
};
```

The syntax of the **`copy()`** method is **`src.copy(dst, dst_offset, src_offset)`**. This is for appending data.

```typescript
// append data to DynBuf
function bufPush(buf: DynBuf, data: Buffer): void {
    const newLen = buf.length + data.length;
    if (buf.data.length < newLen) {
        // grow the capacity ...
    }
    data.copy(buf.data, buf.length, 0);
    buf.length = newLen;
}
```

`Buffer.alloc(cap)` creates a new buffer of a given size. This is for resizing the buffer. The new buffer has to grow *exponentially* so that the amortized cost is O(1). We'll use power of two series for the buffer capacity.

```typescript
// append data to DynBuf
function bufPush(buf: DynBuf, data: Buffer): void {
    const newLen = buf.length + data.length;
    if (buf.data.length < newLen) {
        // grow the capacity by the power of two
        let cap = Math.max(buf.data.length, 32);
        while (cap < newLen) {
            cap *= 2;
        }
        const grown = Buffer.alloc(cap);
        buf.data.copy(grown, 0, 0);
        buf.data = grown;
    }
    data.copy(buf.data, buf.length, 0);
    buf.length = newLen;
}
```

5.3 Implementing a Message Protocol

Step 1: The Server Loop

At a high level, the server should be a loop.

1. Parse and remove a complete message from the incoming byte stream.
    - Append some data to the buffer.
    - Continue the loop if the message is incomplete.
2. Handle the message.
3. Send the response.

```
async function serveClient(socket: net.Socket): Promise<void> {
    const conn: TCPConn = soInit(socket);
    const buf: DynBuf = {data: Buffer.alloc(0), length: 0};
    while (true) {
        // try to get 1 message from the buffer
        const msg: null|Buffer = cutMessage(buf);
        if (!msg) {
            // need more data
            const data: Buffer = await soRead(conn);
            bufPush(buf, data);
            // EOF?
            if (data.length === 0) {
                // omitted ...
                return;
            }
            // got some data, try it again.
            continue;
        }
        // omitted. process the message and send the response ...
    } // loop for messages
}
```

A socket read is not related to any message boundary. What we do is append data to a buffer until it contains a complete message.

The `cutMessage()` function tells if the message is complete.

- It returns `null` if not.
- Otherwise, it removes the message and returns it.

Step 2: Split Messages

The `cutMessage()` function tests if the message is complete using the delimiter `'\n'`.

```
function cutMessage(buf: DynBuf): null|Buffer {
    // messages are separated by '\n'
    const idx = buf.data.subarray(0, buf.length).indexOf('\n');
    if (idx < 0) {
        return null;    // not complete
    }
    // make a copy of the message and move the remaining data to the front
    const msg = Buffer.from(buf.data.subarray(0, idx + 1));
    bufPop(buf, idx + 1);
    return msg;
}
```

Then it makes a copy of the message data, because it will be removed from the buffer.

- `buf.subarray()` returns reference of a subarray without copying.
- `Buffer.from()` creates a new buffer by copying the data from the source.

The message is removed by moving the remaining data to the front.

```
// remove data from the front
function bufPop(buf: DynBuf, len: number): void {
    buf.data.copyWithin(0, len, buf.length);
    buf.length -= len;
}
```

`buf.copyWithin(dst, src_start, src_end)` copies data within a buffer, source and destination can overlap, like `memmove()` in C. This way of handling buffers is not very optimal, more on this later.

Step 3: Handle Requests

Requests are handled and responses are sent in the server loop.

```
    while (true) {
        // try to get 1 message from the buffer
        const msg = cutMessage(buf);
        if (!msg) {
            // omitted ...
            continue;
        }

        // process the message and send the response
        if (msg.equals(Buffer.from('quit\n'))) {
            await soWrite(conn, Buffer.from('Bye.\n'));
            socket.destroy();
            return;
        } else {
            const reply = Buffer.concat([Buffer.from('Echo: '), msg]);
            await soWrite(conn, reply);
        }
    } // loop for messages
```

Our message echo server is now complete. Test it with the `socat` command.

5.4 Discussion: Pipelined Requests

When removing a message from the buffer, we moved the remaining data to the front. You may wonder how there can be any data left in the buffer, since in a request-response scenario, the client waits for the response before sending more requests. This is to support an optimization.

Reduce Latency by Pipelining Requests

Consider a typical modern web page that involves many scripts and style sheets. It takes many HTTP requests to load the page, and each request increases the load time by at least one roundtrip time (RTT). If we can send multiple requests at once, without waiting for the responses one by one, the load time could be greatly reduced. On the server side, the server shouldn't tell the difference because a TCP connection is just a byte stream.

```
client          server
------          ------

|req1|  ==>
|req2|  ==>
|req3|  ==>  <==  |res1|
 ...         <==  |res2|
                   ...
```

This is called *pipelined* requests. It's a common way to reduce RTTs in request-response protocols.

This is why we kept the remaining buffer data around, because there can be more than 1 message in it.

Support Pipelined Requests

While you can make pipelined requests to many well-implemented network servers, such as Redis, NGINX, some less common implementations are problematic! Web browsers do not use pipelined HTTP requests due to buggy servers, they may use multiple concurrent connections instead.

But if you treat TCP data *strictly* as a continuous stream of bytes, pipelined messages should be indistinguishable, because the parser doesn't depend on the size of the buffered data, it just consumes elements one by one.

So pipelined messages are a way to check the correctness of a protocol parser. If the server treats a socket read as a "TCP packet", it would easily fail.

You can test the pipelined scenario with the following command line.

```
echo -e 'asdf\n1234' | socat tcp:127.0.0.1:1234 -
```

The server will probably receive 2 messages in a single `'data'` event, and our server handled them correctly.

Deadlock by Pipelining

A caveat about pipelining requests: pipelining too many requests can lead to deadlocks; because both the server and the client can be sending at the same time, and if both their send buffers are full, it's deadlocked as they are both stuck at sending and cannot drain the buffer.

5.5 Discussion: Smarter Buffers

Removing Data from the Front

There is still $O(n^2)$ behavior in our buffer code; whenever we removed a message from the buffer, we moved the remaining data to the front. This can be triggered by pipelining many small messages.

To fix this, the data movement has to be amortized. This can be done by deferring the data movement. We can keep the remaining data temporarily in place until the wasted space in the front reaches a threshold (such as 1/2 capacity).

This fix requires us to keep track of the beginning of the data, so you'll need a new method for retrieving the real data. The code is left as an exercise to you.

Using Fixed Sized Buffer without Reallocations

Sometimes it's possible to use a large enough buffer without resizing if the message size in the protocol is limited to a reasonably small value.

For example, many HTTP implementations have a small limit on header size as there is no legitimate use case for putting lots of data in the header. For these implementations, one buffer allocation per connection is enough, and the buffer is also sufficient for reading the payload if it doesn't need to store the entire payload in memory.

The advantages of this are:

- No dynamic memory management (except for the initial buffer).
- Memory usage is easy to predict.

This is not very relevant for Node.JS apps, but these advantages are desirable in environments with manual memory management or constrained hardware.

5.6 Conclusion: Network Programming Basics

A summary of what we have learned so far:

- The socket API: listen, accept, read, write, and etc.
- The event loop and its implications.
- Callback vs. `Promise`. Using `async` and `await`.
- Backpressure, queues and buffers.
- TCP byte steam, protocol parser, pipelined message.

CHAPTER 06

HTTP Semantics and Syntax

HTTP is very human-readable, which means you can build a server by looking at examples instead of the specification. However, this approach results in buggy toy code, and you won't learn much. So you need to consult the specification — a series of RFC documents.

6.1 High-Level Structures

Let's review what you already know from the introductory chapter:

- An HTTP request message consists of:
  - The method, which is a verb such as `GET`, `POST`.
  - The URI.
  - A list of header fields, which is a list of key-value pairs.
  - A payload body, which follows the request header. Special case: `GET` and `HEAD` have no payload.
- An HTTP response consists of:
  - A status code, mostly to indicate whether the request was successful.
  - A list of header fields.
  - An optional payload body.

These things are mostly the same from HTTP/1.0 to HTTP/3.

```
HTTP/1.0 200 OK
Age: 525410
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Thu, 20 Oct 2020 11:11:11 GMT
Etag: "1234567890+gzip+ident"
Last-Modified: Thu, 20 Oct 2019 11:11:11 GMT
Vary: Accept-Encoding
Content-Length: 1256
Connection: close

<!doctype html>
<!-- omitted -->
```

6.2 Content-Length

HTTP semantics are mostly about interpreting header fields, which is described in RFC 9110[1]. Try reading it yourself.

[1] https://www.rfc-editor.org/rfc/rfc9110.html

The most important header fields are `Content-Length` and `Transfer-Encoding`, because they determine the length of an HTTP message, which is the most important function of a protocol.

The Length of the HTTP Header

Both a request and a response consist of 2 parts: header + body. They are separated by an empty line. A line ends with `'\r\n'`. So the header ends with `'\r\n\r\n'` including the empty line. That's how we determine the length of the header.

The Length of the HTTP Body

The length of the body is complicated because there are 3 ways to determine it. The **first** way is to use `Content-Length`, which contains the length of the body.

Some ancient HTTP/1.0 software doesn't use `Content-Length`, so the body is just the rest of the connection data, the parser reads the socket to EOF and that's the body. This is the **second** way to determine the body length. This way is problematic because you cannot tell if the connection is ended prematurely.

6.3 Chunked Transfer Encoding

Generate and Send Data on the Fly

The **third** way is to use `Transfer-Encoding: chunked` instead of `Content-Length`. This is called *chunked transfer encoding*. It can mark the end of the payload without knowing its size in advance.

This allows the server to send the response while generating it on the fly. This use case is called *streaming*. An example is displaying real-time logs to the client without waiting for the process to finish.

Another Layer of Protocol

How does chunked encoding work? As the sender, we don't know the total payload length, but we do know the portion of the payload we have. So we can send it in a mini-message format called a "chunk". And a special chunk marks the end of end stream.

The receiver parses the byte stream into chunks and consumes the data, until the special chunk is received. Here is a concrete example:

```
4\r\nHTTP\r\n5\r\nserver\r\n0\r\n\r\n
```

It is parsed into 3 chunks:

- `4\r\nHTTP\r\n`

- `6\r\nserver\r\n`
- `0\r\n\r\n`

You can easily guess how this works. Chunks start with the size of the data, and a 0-sized chunk marks the end of the stream.

Chunks Are Not Messages

Note that the chunk data boundaries are just side effects. These chunks are not represented to the application as individual messages; the application still sees the payload as a byte stream.

6.4 Ambiguities in HTTP

The Happy Cases of Body Length

In summary, this is how to determine the length of the payload body (if the HTTP method allows the payload body, i.e., `POST` or `PUT`).

1. If `Transfer-Encoding: chunked` is present. Parse chunks.
2. If `Content-Length: number` is valid. The length is known.
3. If neither field is present, use the rest of the connection data as the payload.

There are also special cases, such as `GET` and `HEAD`, 304 (Not Modified) status code, which make HTTP not easy to implement.

Mind the Nasty Cases

You may wonder what happens if *both* header fields are present, as there is no clear way to interpret this. This kind of ambiguity is a source of security exploits known as "HTTP request smuggling[2]".

Another ambiguity is the nonexistent payload body for the `GET` request, what if the the request includes `Content-Length`? Should the server ignore the field or forbid the field? What about `Content-Length: 0`?

Also, should the server or client even allow users to mess with the `Content-Length` and `Transfer-Encoding` fields at all? There are many discussions on the Internet, and although the RFC tried to enumerate the cases[3], different implementations handle them differently.

An exercise for the reader: If you are designing a new protocol, how do you avoid ambiguities like this?

[2] https://en.wikipedia.org/wiki/HTTP_request_smuggling
[3] https://www.rfc-editor.org/rfc/rfc9112#section-6.3

6.5 HTTP Message Format

RFC 9112[4] describes exactly how bits are transmitted over the network.

Read the BNF Language

The HTTP message format is described in a language called *BNF*. Go to the "2. Message" section in RFC 9112 and you will see things like this:

```
HTTP-message   = start-line CRLF
                 *( field-line CRLF )
                 CRLF
                 [ message-body ]
start-line     = request-line / status-line
```

This says: An HTTP message is either a request message or a response message. A message starts with either a request line or a status line, followed by multiple header fields, then an empty line, then the optional payload body. Lines are separated by CRLF, which is the ASCII string `'\r\n'`. The BNF language is much more concise and less ambiguous than English.

HTTP Header Fields

```
field-line   = field-name ":" OWS field-value OWS
```

The header field name and value are separated by a colon, but the rules for field name and value are defined in RFC 9110 instead.

```
field-name     = token
token          = 1*tchar
tchar          = "!" / "#" / "$" / "%" / "&" / "'" / "*"
               / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~"
               / DIGIT / ALPHA
               ; any VCHAR, except delimiters

OWS            = *( SP / HTAB )
               ; optional whitespace

field-value    = *field-content
field-content  = field-vchar
                 [ 1*( SP / HTAB / field-vchar ) field-vchar ]
field-vchar    = VCHAR / obs-text
obs-text       = %x80-FF
```

This is the general rule for field name and value. `SP`, `HTAB`, and `VCHAR` refer to space, tab, and printable ASCII character, respectively. Some characters are forbidden in header fields, especially CR and LF.

[4] https://www.rfc-editor.org/rfc/rfc9112.html

Some header fields have additional rules for interpretation, such as comma-separated values or quoted strings. For now, we can just leave them as they are until we need them.

The HTTP specification is very large, and this chapter only covers the most important bits of implementing an HTTP server, which we will do in the next chapter.

6.6 Common Header Fields

Many header fields are either interpreted by applications or used by optional HTTP features, and are not immediately relevant to our implementation. You can become familiar with them by inspecting HTTP headers in browser dev tools.

Table 1: Common header fields. (C means client and S means server.)

Header Field	By	Description
`Content-Length: 60`	C/S	Discussed.
`Transfer-Encoding: chunked`	C/S	Discussed.
`Accept: text/html`	C	For negotiating content types.
`Content-Type: text/html`	S	Content type.
`Accept-Encoding: gzip`	C	For negotiating content compression.
`Content-Encoding: gzip`	S	Compressed response.
`Vary: content-encoding`	S	Tell proxies about content negotiations.
`Authorization: Basic dTpw`	C	Authorization by username and password.
`Cache-Control: no-cache`	C/S	Affect caching behavior.
`Age: 60`	S	How long is the item cached by the proxy?
`Set-Cookie: k=v`	S	HTTP cookie.
`Cookie: k=v`	C	HTTP cookie.
`Date`	C/S	Not very useful.
`Expect: 100-continue`	C	An obscure feature.
`Host: example.com`	C/S	The host name of the URL.
`Last-Modified`	S	For cache validation and 304 Not Modified.
`If-Modified-Since`	C	Validate `Last-Modified`.
`ETag: abcd`	S	For cache validation and 304 Not Modified.
`If-None-Match: abcd`	C	Validate `ETag`.
`Range: 10-`	C	Range request. Get a portion of the response.
`Content-Range: bytes 10-/60`	S	Range response.
`Accept-Ranges: bytes`	S	Indicate that range requests are allowed.
`Referer: http://foo.com/`	C	Where is the user from?
`Transfer-Encoding: gzip`	S	An alternative way to achieve compression.
`TE: gzip`	C	For negotiating `Transfer-Encoding`.
`Trailer: Foo`	C/S	Obscure feature: Header fields after payload.
`User-Agent: Foo`	C	Client software.
`Server: Foo`	S	Server software.

Header Field	By	Description
`Upgrade: websocket`	C/S	Create WebSockets.
`Access-Control-*`	S	For cross-origin resource sharing (CORS).
`Origin`	C	CORS.
`Location: http://bar.com/`	S	For 3xx redirections.

6.7 HTTP Methods

Read-Only Methods

The 2 most important HTTP methods are `GET` and `POST`. Why do we need different HTTP methods? Besides the obvious fact that a `POST` request can carry a payload where a `GET` cannot, it is also a good idea to separate *read-only* operations from *write* operations. You can use `GET` for read-only operations and `POST` for the rest.

A read-only method is called a "safe" method. There are 3 safe methods:

- `GET`.
- `HEAD`, like `GET` but without the response body.
- `OPTIONS`, rarely used, for identifying allowed request methods and CORS related things.

Cacheability

One reason for separating read-only operations from write operations is that read-only operations are generally cacheable. On the other hand, it makes no sense to cache write operations as they are state-changing.

However, the rules for cacheability are more complicated than different HTTP methods.

- `GET` and `HEAD` are considered to be cacheable methods. But `OPTIONS` is not, as it is for special purposes.
- The status code also affects cacheability.
- `Cache-Control` header can affect cacheability.
- `POST` is usually not cacheable, unless[5] an obscure header field (`Content-Location`) is used and certain cache directives are present.
- Different implementations have different cacheability rules.

[5] https://www.rfc-editor.org/rfc/rfc9110#section-9.3.3-5

CRUD and Resources

You may have wondered why there are so many HTTP methods. Wouldn't just `GET` and `POST` suffice? In fact, that's what many applications do. More methods were added to HTTP because people imagined HTTP as a protocol for managing "resources". For example, a forum user can manipulate his posts as resources:

- **Create** a post via `PUT`.
- **Read** a post via `GET`.
- **Update** a post via `PATCH`.
- **Delete** a post via `DELETE`.

These 4 verbs are often referred to as **CRUD**.

Idempotence

But why add CRUD as HTTP methods? A forum user may also move a post to another forum, should HTTP also include a `MOVE` method? Mirroring arbitrary English verbs is not a good reason to define HTTP methods. One of the better reasons is to define the *idempotence* of operations.

An idempotent operation is one that can be repeated with the same effect. This means that you can safely *retry* the operation until it succeeds. For example, if you `rm` a file over SSH and the connection breaks before you see the result, so the state of the file is unknown to you, but you can always blindly `rm` it again (if it's really the same file):

- If you were to fail, you'll probably fail again.
- If the previous `rm` failed but this one succeeds, your intention is just fulfilled.
- If the previous `rm` succeeded, there is no harm in doing it again.

An idempotent operation over HTTP can still result in a different status code, just like the return code of `rm`.

Idempotence in HTTP:

- Read-only methods (`GET` and `HEAD`) are obviously idempotent.
- `PUT` and `DELETE` are idempotent, as are overwriting and deleting files.
- `POST` and `PATCH` are NOT defined as idempotent. They may or may not be.

Idempotence in browsers:

- If you submit a `<form>` via `POST` and then refresh the page, the browser will warn you against resubmitting the potentially non-idempotent form.
- HTML forms are limited to `GET` and `POST`[6]. You need AJAX to use these idempotent methods.

[6] https://softwareengineering.stackexchange.com/a/211790

But this still doesn't answer the puzzle of why there are so many verbs, because HTTP could just add 1 more method for idempotent writes instead of 3 (`PATCH`, `PUT`, `DELETE`). In fact, there may be no strong reason for apps to use them all.

Comparison of HTTP Methods

A summary of general-purpose HTTP methods.

Verb	Safe	Idempotent	Cacheable	`<form>`	CRUD	Req body	Res body
`GET`	Yes	Yes	Yes	Yes	read	No	Yes
`HEAD`	Yes	Yes	Yes	No	read	No	No
`POST`	No	No	No*	Yes	-	Yes	Yes
`PATCH`	No	No	No*	No	update	Yes	May
`PUT`	No	Yes	No	No	create	Yes	May
`DELETE`	No	Yes	No	No	delete	May	May

Note: Cacheable `POST` or `PATCH` is possible, but rarely supported.

6.8 Discussion: Text vs. Binary

HTTP is designed in a way that you can send requests from `telnet`, so you can learn it by poking around. However, textual protocols have downsides.

Text is Often Ambiguous

One downside is that human-readable formats are often less machine-readable, because they are more flexible than necessary. Consider the way HTTP payload length is determined:

- The common cases are `Content-Length` and `Transfer-Encoding`.
- There are special cases for some HTTP methods and status codes.
- There are cases that cause different interpretations, e.g., request smuggling.

HTTP is a simple protocol, where simple means it's easy to look at. Writing code for it is not simple because there are too many rules for interpreting it, and the rules still leave you with ambiguities.

Text is More Work & Error-Prone

Another downside is that dealing with text is a lot more work. To properly handle text strings, you need to know their length first, which is often determined by delimiters. The extra work of looking for delimiters is the cost of human-readable formats.

It's also error-prone; in C programming, null-terminated strings (0-delimited) have caused many security exploits.

HTTP/2 is binary and more complex than HTTP/1.1, but parsing the protocol is still easier because you don't have to deal with elements of unknown length.

6.9 Discussion: Delimiters

Serialization Errors in Delimited Data

Delimiters are everywhere in textual protocols. For example, in HTTP …

- Lines in the header are delimited by CRLF
- The header and body are delimited by an empty line.

One problem with delimiters is that the data cannot contain the delimiter itself. Failure to enforce this rule can lead to some injection exploits.

If a malicious client can trick a buggy server into emitting an header field value with CRLF in it, and the header field is the last field, then the payload body starts with the part of the field value that the attacker controls. This is called "HTTP response splitting[7]".

A proper HTTP server/client must forbid CRLF in header fields as there is no way to encode them. However, this is not true for many generic data formats. For example, JSON uses `{}[],:` to delimit elements, but a JSON string can contain arbitrary characters, so strings are *quoted* to avoid ambiguity with delimiters. But the quotes themselves are also delimiters, so *escape sequences* are needed to encode quotes.

This is why you need a JSON library to produce JSON instead of concatenating strings together. And HTTP is less well defined and more complicated than JSON, so pay attention to the specifications.

[7] https://en.wikipedia.org/wiki/HTTP_response_splitting

Length-Prefixed Data in Binary Protocols

Delimiters in text are used to separate elements. In binary protocols and formats, a better and simpler alternative is to use *length-prefixed* data, that is, to specify the length of the element before the element data. Some examples are:

- The chunked transfer encoding. Although the length itself is still delimited.
- The WebSocket frame format. No delimiters at all.
- HTTP/2. Frame-based.
- The MessagePack[8] serialization format. Some kind of binary JSON.

[8] https://github.com/msgpack/msgpack/blob/master/spec.md

CHAPTER 07

Code A Basic HTTP Server

Our HTTP server is based on the message echo server from the previous chapter, with the “message” replaced by the HTTP message.

7.1 Start Coding

The code is broken into small steps and follows a top-down approach.

Step 1: Types and Structures

Our first step is to define the structure for HTTP messages based on our understanding of HTTP semantics.

```
// a parsed HTTP request header
type HTTPReq = {
    method: string,
    uri: Buffer,
    version: string,
    headers: Buffer[],
};

// an HTTP response
type HTTPRes = {
    code: number,
    headers: Buffer[],
    body: BodyReader,
};
```

We use `Buffer` instead of `string` for the URI and header fields. Although HTTP is mostly plaintext, there is no guarantee that URI and header fields must be ASCII or UTF-8 strings. So we just leave them as bytes until we need to parse them.

The `BodyReader` type is the interface for reading data from the body payload.

```
// an interface for reading/writing data from/to the HTTP body.
type BodyReader = {
    // the "Content-Length", -1 if unknown.
    length: number,
    // read data. returns an empty buffer after EOF.
    read: () => Promise<Buffer>,
};
```

The payload body can be arbitrarily long, it may not even fit in memory, thus we have to use the `read()` function to read from it instead of a simple `Buffer`. The `read()` function follows the convention of the `soRead()` function — the end of data is signaled by an empty `Buffer`.

And when using chunked encoding, the length of the body is not known, which is another reason why this interface is needed.

Step 2: The Server Loop

The server loop follows the pattern from the previous chapter. Except that the `cutMessage()` function only parses the HTTP header; the payload body is expected to be read while handling the request, or discarded after handling the request. In this way, we don't store the entire payload body in memory.

```
async function serveClient(conn: TCPConn): Promise<void> {
    const buf: DynBuf = {data: Buffer.alloc(0), length: 0};
    while (true) {
        // try to get 1 request header from the buffer
        const msg: null|HTTPReq = cutMessage(buf);
        if (!msg) {
            // need more data
            const data = await soRead(conn);
            bufPush(buf, data);
            // EOF?
            if (data.length === 0 && buf.length === 0) {
                return; // no more requests
            }
            if (data.length === 0) {
                throw new HTTPError(400, 'Unexpected EOF.');
            }
            // got some data, try it again.
            continue;
        }

        // process the message and send the response
        const reqBody: BodyReader = readerFromReq(conn, buf, msg);
        const res: HTTPRes = await handleReq(msg, reqBody);
        await writeHTTPResp(conn, res);
        // close the connection for HTTP/1.0
        if (msg.version === '1.0') {
            return;
        }
        // make sure that the request body is consumed completely
        while ((await reqBody.read()).length > 0) { /* empty */ }
    } // loop for IO
}
```

The `HTTPError` is a custom exception type defined by us. It is used to generate an error response and close the connection. Note that this thing exists only to make our code simpler by deferring the unhappy case of error handling. You probably don't want to throw exceptions around like this in production code.

```
async function newConn(socket: net.Socket): Promise<void> {
    const conn: TCPConn = soInit(socket);
    try {
        await serveClient(conn);
    } catch (exc) {
        console.error('exception:', exc);
        if (exc instanceof HTTPError) {
            // intended to send an error response
            const resp: HTTPRes = {
                code: exc.code,
                headers: [],
                body: readerFromMemory(Buffer.from(exc.message + '\n')),
            };
            try {
                await writeHTTPResp(conn, resp);
            } catch (exc) { /* ignore */ }
        }
    } finally {
        socket.destroy();
    }
}
```

Step 3: Split the Header

The HTTP header ends with `'\r\n\r\n'`, which is how we determine its length.

In theory, there is no limit to the size of the header, but in practice there is. Because we are going to parse and store the header in memory, and memory is finite.

```
// the maximum length of an HTTP header
const kMaxHeaderLen = 1024 * 8;

// parse & remove a header from the beginning of the buffer if possible
function cutMessage(buf: DynBuf): null|HTTPReq {
    // the end of the header is marked by '\r\n\r\n'
    const idx = buf.data.subarray(0, buf.length).indexOf('\r\n\r\n');
    if (idx < 0) {
        if (buf.length >= kMaxHeaderLen) {
            throw new HTTPError(413, 'header is too large');
        }
        return null;    // need more data
    }
    // parse & remove the header
    const msg = parseHTTPReq(buf.data.subarray(0, idx + 4));
    bufPop(buf, idx + 4);
    return msg;
}
```

Parsing is also easier when we have the complete data. That's another reason why we waited for the full HTTP header before parsing anything.

Step 4: Parse the Header

To parse an HTTP header, we can first split the data into lines by CRLF since we have the complete header in the buffer. Then we can process each line individually.

```
// parse an HTTP request header
function parseHTTPReq(data: Buffer): HTTPReq {
    // split the data into lines
    const lines: Buffer[] = splitLines(data);
    // the first line is `METHOD URI VERSION`
    const [method, uri, version] = parseRequestLine(lines[0]);
    // followed by header fields in the format of `Name: value`
    const headers: Buffer[] = [];
    for (let i = 1; i < lines.length - 1; i++) {
        const h = Buffer.from(lines[i]);    // copy
        if (!validateHeader(h)) {
            throw new HTTPError(400, 'bad field');
        }
        headers.push(h);
    }
    // the header ends by an empty line
    console.assert(lines[lines.length - 1].length === 0);
    return {
        method: method, uri: uri, version: version, headers: headers,
    };
}
```

The first line is simply 3 pieces separated by space. The rest of the lines are header fields. Although we're not trying to parse the header fields here, it's still a good idea to do some validations on them.

The **`splitLines()`**, **`parseRequestLine()`**, and **`validateHeader()`** functions are not very interesting, so we will not show them here. You can easily code them yourself according to RFCs.

Step 5: Read the Body

Before handling the request, we must first construct the **`BodyReader`** object which will be passed to the handler function. There are 3 ways to read the payload body, as we mentioned earlier.

```
// BodyReader from an HTTP request
function readerFromReq(
    conn: TCPConn, buf: DynBuf, req: HTTPReq): BodyReader
{
    let bodyLen = -1;
    const contentLen = fieldGet(req.headers, 'Content-Length');
    if (contentLen) {
        bodyLen = parseDec(contentLen.toString('latin1'));
        if (isNaN(bodyLen)) {
            throw new HTTPError(400, 'bad Content-Length.');
        }
    }
    const bodyAllowed = !(req.method === 'GET' || req.method === 'HEAD');
    const chunked = fieldGet(req.headers, 'Transfer-Encoding')
        ?.equals(Buffer.from('chunked')) || false;
    if (!bodyAllowed && (bodyLen > 0 || chunked)) {
        throw new HTTPError(400, 'HTTP body not allowed.');
    }
    if (!bodyAllowed) {
        bodyLen = 0;
    }

    if (bodyLen >= 0) {
        // "Content-Length" is present
        return readerFromConnLength(conn, buf, bodyLen);
    } else if (chunked) {
        // chunked encoding
        throw new HTTPError(501, 'TODO');
    } else {
        // read the rest of the connection
        throw new HTTPError(501, 'TODO');
    }
}
```

Here we need to look at the `Content-Length` field and the `Transfer-Encoding` field. The `fieldGet()` function is for looking up the field value by name. Note that field names are case-insensitive. The implementation is left to the reader.

```
function fieldGet(headers: Buffer[], key: string): null|Buffer;
```

We will only implement the case where the `Content-Length` field is present, the other cases are left for later chapters.

```
// BodyReader from a socket with a known length
function readerFromConnLength(
    conn: TCPConn, buf: DynBuf, remain: number): BodyReader
{
    return {
        length: remain,
        read: async (): Promise<Buffer> => {
            if (remain === 0) {
                return Buffer.from(''); // done
            }
            if (buf.length === 0) {
                // try to get some data if there is none
                const data = await soRead(conn);
                bufPush(buf, data);
                if (data.length === 0) {
                    // expect more data!
                    throw new Error('Unexpected EOF from HTTP body');
                }
            }
            // consume data from the buffer
            const consume = Math.min(buf.length, remain);
            remain -= consume;
            const data = Buffer.from(buf.data.subarray(0, consume));
            bufPop(buf, consume);
            return data;
        }
    };
}
```

The `readerFromConnLength()` function returns a `BodyReader` that reads exactly the number of bytes specified in the `Content-Length` field. Note that the data from the socket goes into the buffer first, then we drain data from the buffer. This is because:

- There may be extra data in the buffer before we read from the socket.
- The last read may return more data than we need, so we need to put the extra data back into the buffer.

The `remain` variable is a state captured by the `read()` function to keep track of the remaining body length.

Step 6: The Request Handler

We can now handle the request according to its URI and method. Here we will show you 2 sample responses.

```
// a sample request handler
async function handleReq(req: HTTPReq, body: BodyReader): Promise<HTTPRes> {
    // act on the request URI
    let resp: BodyReader;
    switch (req.uri.toString('latin1')) {
```

```
    case '/echo':
        // http echo server
        resp = body;
        break;
    default:
        resp = readerFromMemory(Buffer.from('hello world.\n'));
        break;
    }

    return {
        code: 200,
        headers: [Buffer.from('Server: my_first_http_server')],
        body: resp,
    };
}
```

If the URI is `'/echo'`, we simply set the response payload to the request payload. This essentially creates an echo server in HTTP. You can test this by `POST`ing data with the `curl` command.

```
curl -s --data-binary 'hello' http://127.0.0.1:1234/echo
```

The other sample response is a fixed string `'hello world.\n'`. To do this, we must first create the `BodyReader` object.

```
// BodyReader from in-memory data
function readerFromMemory(data: Buffer): BodyReader {
    let done = false;
    return {
        length: data.length,
        read: async (): Promise<Buffer> => {
            if (done) {
                return Buffer.from(''); // no more data
            } else {
                done = true;
                return data;
            }
        },
    };
}
```

The `read()` function returns the full data on the first call and returns EOF after that. This is useful for responding with something small and already fits in memory.

Step 7: Send the Response

After handling the request, we can send the response header and the response body if there is one. In this chapter, we will only deal with the payload body of known length; the chunked encoding is left for later chapters. All we need to do is to add the `Content-Length` field.

```
// send an HTTP response through the socket
async function writeHTTPResp(conn: TCPConn, resp: HTTPRes): Promise<void> {
    if (resp.body.length < 0) {
        throw new Error('TODO: chunked encoding');
    }
    // set the "Content-Length" field
    console.assert(!fieldGet(resp.headers, 'Content-Length'));
    resp.headers.push(Buffer.from(`Content-Length: ${resp.body.length}`));
    // write the header
    await soWrite(conn, encodeHTTPResp(resp));
    // write the body
    while (true) {
        const data = await resp.body.read();
        if (data.length === 0) {
            break;
        }
        await soWrite(conn, data);
    }
}
```

The `encodeHTTPResp()` function encodes a response header into a byte buffer. The message format is almost identical to the request message, except for the first line.

```
status-line = HTTP-version SP status-code SP [ reason-phrase ]
```

Encoding is much easier than parsing, so the implementation is left to the reader.

Step 8: Review the Server Loop

There is still work to be done after sending the response. We can provide some compatibility for HTTP/1.0 clients by closing the connection immediately, since the connection cannot be reused anyway.

And most importantly, before continuing the loop to the next request, we must make sure that the request body is completely consumed, because the handler function may have ignored the request body and left the parser at the wrong position.

```
async function serveClient(conn: TCPConn): Promise<void> {
    const buf: DynBuf = {data: Buffer.alloc(0), length: 0};
    while (true) {
        // try to get 1 request header from the buffer
        const msg: null|HTTPReq = cutMessage(buf);
        if (!msg) {
            // omitted ...
            continue;
        }

        // process the message and send the response
        const reqBody: BodyReader = readerFromReq(conn, buf, msg);
        const res: HTTPRes = await handleReq(msg, reqBody);
        await writeHTTPResp(conn, res);
        // close the connection for HTTP/1.0
        if (msg.version === '1.0') {
            return;
        }
        // make sure that the request body is consumed completely
        while ((await reqBody.read()).length > 0) { /* empty */ }
    } // loop for IO
}
```

Our first HTTP server is now complete.

7.2 Testing

The simplest test case is to make requests with **`curl`**. The server should greet you with “hello world”. You can also POST data to the **`'/echo'`** path and the server should echo the data back.

```
curl -s --data-binary 'hello' http://127.0.0.1:1234/echo
```

Large HTTP Body

The `curl` command can also post data from files. We can post a really big file to verify that our server is only using constant memory and not triggering OOM.

```
curl -s --data-binary @a_big_file http://127.0.0.1:1234/echo | sha1sum
```

Connection Reuse & Pipelining

Another important thing to test is the ability to handle multiple requests per connection. You can test this either interactively via `socat`, or automatically via shell scripting.

```
(cat req1.txt; sleep 1; cat req2.txt) | socat tcp:127.0.0.1:1234,crnl -
```

Note the `crnl` option in the `socat` command, this is to make sure that lines end with CRLF instead of just LF.

If you remove the `sleep 1` in the above script, you will also be testing pipelined requests.

7.3 Discussion: Nagle's Algorithm

Optimization: Combining Small Writes

When sending the response, we used the `encodeHTTPResp()` function to create a byte buffer of the header before writing the response to the socket. Some people may skip this step and write to the socket line by line.

```
// Bad example!
await soWrite(conn, Buffer.from(`HTTP/1.1 ${msg.code} ${status}\r\n`));
for (const h of msg.headers) {
    await soWrite(conn, h);
    await soWrite(conn, Buffer.from('\r\n'));
}
await soWrite(conn, Buffer.from('\r\n'));
```

The problem with this is that it generates many small writes, causing TCP to send many small packets. Not only does each packet have a relatively large space overhead, but more computation is required to process more packets. People saw this optimization opportunity and added a feature to the TCP stack known as "Nagle's algorithm" — the TCP stack *delays* transmission to allow the send buffer to accumulate data, so that multiple consecutive small writes can be combined.

Premature Optimization

However, this is not a good optimization. Many newer network protocol designs, such as TLS, have put a lot of effort into reducing RTTs because many performance problems are latency problems. Adding delays to TCP to combine writes now looks like anti-optimization. And the intended optimization goal can easily be achieved at the application level instead; applications can simply combine small data themselves without delays.

Well-written applications should manage buffers carefully, either by explicitly serializing data into a buffer, or by using some buffered IO interfaces, so that Nagle's algorithm is not needed. And high-performance applications will want to minimize the number of syscalls, making Nagle's algorithm even more useless.

What People Actually Do in Practice

When developing networked applications:

1. Avoid small writes by combining small data before writing.
2. Disable Nagle's algorithm.

Nagle's algorithm is often enabled by default. This can be disabled using the `noDelay` flag in Node.js.

```
const server = net.createServer({
    noDelay: true,  // TCP_NODELAY
});
```

7.4 Discussion: Buffered Writer

Alternative: Make Buffering Semi-Transparent

Instead of explicitly serializing data into a buffer, as we do with the response header, we can also add a buffer to the `TCPConn` type and change the way it works.

```
// append data to an internal buffer
function soWrite(conn: TCPConn, data: Buffer): Promise<void>;
// flush the buffer to the runtime
function soFlush(conn: TCPConn): Promise<void>;
```

In the new scheme, the `soWrite()` function is changed to append data to an internal buffer in `TCPConn`, and the new `soFlush()` function is used to actually write the data. The buffer size is limited, and the `soWrite()` function can also flush the buffer when it is full.

This style of IO is very popular and you may have seen it in other programming languages. For example, `stdio` in C has a built-in buffer which is enabled by default, you must use `fflush()` when appropriate.

Alternative: Add a Buffered Wrapper

Alternatively, you can leave the `TCPConn` as is, and add a separate wrapper type like this:

```
type BufferedWriter = {
    write: (data: Buffer) => Promise<void>,
    flush: () => Promise<void>,
    // ...
};

function createBufferedWriter(conn: TCPConn): BufferedWriter;
```

This is similar to the `bufio.Writer` in Golang. This scheme is more flexible than adding buffering to the socket code, because the buffered wrapper is also applicable to other forms of IO. And the Go standard library was designed with well-defined interfaces (`io.Writer`[1]), making the buffered writer a drop-in replacement for the unbuffered writer.

There are many more good ideas to steal from the Go standard library. One of them is that the `bufio.Writer` is not just an `io.Writer`, but also exposes its internal buffer[2] so that you can write to it directly! This can eliminate temporary buffers and extra data copies when serializing data.

[1] https://pkg.go.dev/io#Writer

[2] https://pkg.go.dev/bufio#example-Writer.AvailableBuffer

CHAPTER 08

Dynamic Content and Streaming

Historically, the WWW has consisted mostly of static web pages with hyperlinks, and HTTP was originally designed for that. Then came Web-based apps that serve dynamic content, which is what we'll consider in this chapter.

8.1 Chunked Transfer Encoding

To serve dynamic content, we need to implement the chunked encoding which was missing from the last chapter. Because when we generate the web page on the fly, we don't know the length of the output in advance.

The Chunked Message Format

Let's start with the specification — RFC 9112.

```
chunked-body   = *chunk
                 last-chunk
                 trailer-section
                 CRLF

chunk          = chunk-size [ chunk-ext ] CRLF
                 chunk-data CRLF
chunk-size     = 1*HEXDIG
last-chunk     = 1*("0") [ chunk-ext ] CRLF

chunk-data     = 1*OCTET ; a sequence of chunk-size octets
```

Add an example as a reference:

```
4\r\nHTTP\r\n5\r\nserver\r\n0\r\n\r\n
```

The format is pretty straightforward with the example.

- Each chunk starts with a hexadecimal number that ends with CRLF; this is the length of the chunk data.
- The following chunk data also ends with CRLF; this CRLF serves no purpose other than to make the protocol human-readable.
- A zero-length chunk marks the end of the HTTP body. We have seen this convention often!

Obscure HTTP Features

There are still a few constructs that we haven't touched in the BNF specification — the optional `chunk-ext` on the `chunk-size` line and the `trailer-section` at the end. These are some obscure features that were designed into the specification, but not used much in practice.

The `chunk-ext` is designed to extend the chunk format by adding additional key-value pairs, but in practice there is no need for such extensions.

The `trailer-section` is designed to put some header fields after the HTTP body. You may wonder why would one do that, this is for some rare use cases. Some people like to put application-specific stuff in HTTP header fields, such as the checksum of the data. The problem is that when streaming data, the checksum is only known after the data has been streamed, so a mechanism has been designed to append some header fields after the last chunk.

Many HTTP server or client implementations simply ignore these obscure features as they have little use or value. On the other hand, it is unwise to design your application to use any obscure features, even if they are backed by RFC, because you are more likely to run into problems with clients and middleware. We will ignore these in our implementation as well.

8.2 Generating Chunked Responses

The first step is to add chunked encoding to the `writeHTTPResp()` function, which pulls the response body from a `BodyReader` interface.

```
// an interface for reading/writing data from/to the HTTP body.
type BodyReader = {
    // the "Content-Length", -1 if unknown.
    length: number,
    // read data. returns an empty buffer after EOF.
    read: () => Promise<Buffer>,
};
```

The interface from the last chapter was designed with chunked encoding in mind. What we need to do is to implement the case where the length is -1 (unknown).

```
// send an HTTP response through the socket
async function writeHTTPResp(conn: TCPConn, resp: HTTPRes): Promise<void> {
    // set the "Content-Length" or "Transfer-Encoding" field
    if (resp.body.length < 0) {
        resp.headers.push(Buffer.from('Transfer-Encoding: chunked'));
    } else {
        resp.headers.push(Buffer.from(`Content-Length: ${resp.body.length}`));
    }
    // write the header
    await soWrite(conn, encodeHTTPResp(resp));
    // write the body
    const crlf = Buffer.from('\r\n');
    for (let last = false; !last; ) {
        let data = await resp.body.read();
        last = (data.length === 0); // ended?
        if (resp.body.length < 0) { // chunked encoding
            data = Buffer.concat([
                Buffer.from(data.length.toString(16)), crlf,
                data, crlf,
            ]);
        }
        if (data.length) {
            await soWrite(conn, data);
        }
    }
}
```

Note that the chunked message is created with **`Buffer.concat()`** before we send it to the socket with a single write. This follows the advice from the last chapter on buffered IO.

8.3 JS Generators

The next step is to connect the application code that generates the response body (producer) to the **`BodyReader`** interface (queue), which the **`writeHTTPResp()`** function (consumer) pulls data from.

```
// pseudo code!
async function produce_response() {
    // ...
    await output(data1);    // consumed from a BodyReader
    // ...
    await output(data2);    // consumed from a BodyReader
    // ...
}
```

Producers, Consumers and Queues

Using a queue is a common solution to producer-consumer problems. And a queue can be implemented using promises. But we'll explore that later, because implementing a blocking queue is non-trivial compared to using generators.

JS Generators as Producers

JavaScript generators are well suited for producer-consumer problems. The producer generator uses the `yield` statement to pass data and control to the consumer. And when the consumer pulls data again, execution resumes from the last `yield` statement. Below is a sample response generator.

```
type BufferGenerator = AsyncGenerator<Buffer, void, void>;

// count to 99
async function *countSheep(): BufferGenerator {
    for (let i = 0; i < 100; i++) {
        // sleep 1s, then output the counter
        await new Promise((resolve) => setTimeout(resolve, 1000));
        yield Buffer.from(`${i}\n`);
    }
}
```

The asterisk after the `function` keyword is the syntax for JS generators. You can think of generators as functions with multiple returns (`yield`). The statement is called `yield` because they yield to the runtime, like a `return` in a normal function, or an `await` in an async function. And there are both normal and async generators.

The `AsyncGenerator` TypeScript interface is used to specify the type of an async generator, it has 3 type parameters:

1. The type of the `yield` value.
2. The type of the `return` value.
3. The type of the optional argument of the `next()` method, we'll explain this later.

Consume from JS Generators

Let's add a new URI handler for the response generator. The `readerFromGenerator()` function converts a JS generator into a `BodyReader`.

```
    case '/sheep':
        resp = readerFromGenerator(countSheep());
        break;
```

To pull data from a generator, use the **`next()`** method. The **`next()`** method returns when the generator **`yield`**s or **`return`**s, which is differentiated by the **`done`** flag, and the data can be retrieved from the **`value`** member.

```
function readerFromGenerator(gen: BufferGenerator): BodyReader {
    return {
        length: -1,
        read: async (): Promise<Buffer> => {
            const r = await gen.next();
            if (r.done) {
                return Buffer.from(''); // EOF
            } else {
                console.assert(r.value.length > 0);
                return r.value;
            }
        },
    };
}
```

The **`next()`** method can take an optional argument that is passed to the producer as the result of the **`yield`** statement, which means that JS generators are bi-directional. Although we don't need this capability for now.

Test the **`/sheep`** URI with **`curl`** and you should see the counter output every 1s, which means the chunked encoding is working. There is no limit to the total body length when using chunked encoding, you can even generate an infinite byte stream.

8.4 Reading Chunked Requests

The chunked encoding is mostly used for responses instead of requests. A web browser always knows the length of the upload, unless the client JS is using the experimental Streams API[1] to send requests[2]. To support such clients, let's add code to read chunked requests.

[1] https://developer.mozilla.org/en-US/docs/Web/API/ReadableStream

[2] https://developer.mozilla.org/en-US/docs/Web/API/fetch#body

Chunk Parser as a Producer

The code to decode the chunked request must be connected to the **BodyReader** interface. This is exactly the same task when we connected the response generator to the **BodyReader** interface. We will use a JS generator again.

```
// decode the chunked encoding and yield the data on the fly
async function* readChunks(conn: TCPConn, buf: DynBuf): BufferGenerator;
```

The `readChunks()` generator should behave just like the sample response generator we coded before. So we can just use `readerFromGenerator()` to convert it into a `BodyReader`.

```
// BodyReader from an HTTP request
function readerFromReq(
    conn: TCPConn, buf: DynBuf, req: HTTPReq): BodyReader
{
    // omitted ...
    if (bodyLen >= 0) {
        // "Content-Length" is present
        return readerFromConnLength(conn, buf, bodyLen);
    } else if (chunked) {
        // chunked encoding
        return readerFromGenerator(readChunks(conn, buf));
    } else {
        // read the rest of the connection
        return readerFromConnEOF(conn, buf);
    }
}
```

The `readerFromConnEOF()` function is for compatibility with HTTP/1.0 clients. We'll skip the code listing.

Decode the Chunk Format

The implementation of `readChunks()`:

```
// decode the chunked encoding and yield the data on the fly
async function* readChunks(conn: TCPConn, buf: DynBuf): BufferGenerator {
    for (let last = false; !last; ) {
        // read the chunk-size line
        const idx = buf.data.subarray(0, buf.length).indexOf('\r\n');
        if (idx < 0) {
            // need more data, omitted ...
            continue;
        }
        // parse the chunk-size and remove the line
        let remain = /* omitted ... */;
        bufPop(buf, /* omitted ... */);
        // is it the last one?
        last = (remain === 0);
        // read and yield the chunk data
        while (remain > 0) {
            if (buf.length === 0) {
                await bufExpectMore(conn, buf, 'chunk data');
            }

            const consume = Math.min(remain, buf.length);
            const data = Buffer.from(buf.data.subarray(0, consume));
            bufPop(buf, consume);
            remain -= consume;
            yield data;
        }
        // the chunk data is followed by CRLF
        // omitted ...
        bufPop(buf, 2);
    } // for each chunk
}
```

This is similar to the header-body structure of HTTP itself. You wait for the header (`chunk-size`) and find out the length of the body (`chunk-data`). The header is of variable length, so we need to limit its size while parsing, just like what we did with the HTTP header.

Chunked Encoding Produces a Byte Stream

Decoding the format is easy. What you need to pay attention to is the inner loop for reading the chunk data. The code doesn't wait for the full chunk data to arrive, instead it yield the data whenever it arrives. Remember that the chunked encoding is still supposed to present the application with a *byte stream* intead of messages. If we waited for the full chunk data, we would have to store the whole chunk in memory and impose a maximum chunk size limit.

Chunked requests can be tested using `curl`:

```
curl -T- http://127.0.0.1:1234/echo
```

The `curl` will read data from stdin till EOF, without knowing the request length beforehand.

8.5 Discussion: Debugging with Packet Capture Tools

We have been writing non-trivial networking code. It is important to know how to debug your code.

Adding print statements is a quick way to inspect what's happening. However, there is a cheaper way for network applications: Packet capture tools, such as **`tcpdump`**, **`ngrep`**, and Wireshark, can intercept TCP data from the network. This is especially useful in production where you cannot just edit the code.

Capture & Inspect Data with `tcpdump`

An example: **`tcpdump -X -i lo port 1234`**

- The `-X` flag tells **`tcpdump`** to do a hex dump.
- The `-i lo` flag restricts the network interface. This is the loopback interface `lo` as we only care about connections to **`127.0.0.1:1234`**.
- **`port 1234`** is a boolean expression to filter out irrelevant packets.

A sample output from **`tcpdump`**:

```
12:00:00.222444 IP 127.0.0.1.1234 > 127.0.0.1.59872:
  0x0000:  4500 0079 c492 4000 4006 77ea 7f00 0001  E..y..@.@.w.....
  0x0010:  7f00 0001 04d2 e9e0 42d3 38e1 bb89 becb  ........B.8.....
  0x0020:  8018 0200 fe6d 0000 0101 080a 8f86 a39c  .....m..........
  0x0030:  8f86 a395 4854 5450 2f31 2e31 2032 3030  ....HTTP/1.1.200
  0x0040:  204f 4b0d 0a53 6572 7665 723a 206d 795f  .OK..Server:.my_
  0x0050:  6669 7273 745f 6874 7470 5f73 6572 7665  first_http_serve
  0x0060:  720d 0a43 6f6e 7465 6e74 2d4c 656e 6774  r..Content-Lengt
  0x0070:  683a 2031 330d 0a0d 0a                   h:.13....
12:00:00.222666 IP 127.0.0.1.59872 > 127.0.0.1.1234:
...
12:00:00.333444 IP 127.0.0.1.1234 > 127.0.0.1.59872:
  0x0000:  4500 0041 c493 4000 4006 7821 7f00 0001  E..A..@.@.x!....
  0x0010:  7f00 0001 04d2 e9e0 42d3 3926 bb89 becb  ........B.9&....
  0x0020:  8018 0200 fe35 0000 0101 080a 8f86 a39d  .....5..........
  0x0030:  8f86 a39c 6865 6c6c 6f20 776f 726c 642e  ....hello.world.
  0x0040:  0a                                       .
...
```

Analyze Packets with Wireshark

You can also use the `-w FILE` flag to save the captured packets to a file instead of printing them. The file format can be read by all other packet analysis software, including `tcpdump` itself.

If you prefer GUIs, you can open the captured file with Wireshark, which, in addition to showing the hex dump, can also …

- Filter packets with boolean expressions. This takes effect immediately, so you can do it by trial and error.
- Highlight individual connections.
- Disassemble the protocol, show and highlight each element and structure.
  - Quickly learn new protocols using real data.
  - See what's wrong with your own implementation.

Search Packet Data with `ngrep`

For textual protocols like HTTP, hex dumps can be overkill. You can use `ngrep` to print the data as text.

```
ngrep -qt -d lo '' port 1234
```

- The `-d lo` is for the network interface.
- The `port 1234` is similar to the boolean expression used by `tcpdump`.
- The empty string `''` is a regular expression that filters packets through the TCP data. We use the empty regex to show all packets.

`ngrep` is like `grep`, but for sockets instead of files, hence its name.

It's worth noting that all of these tools can write packets to a file in an interchangeable format (pcap). You can capture packets once and analyze them later.

8.6 Discussion: WebSocket

Polling for Updates

Some web-based apps need to receive live updates from the server. The most naive way to do this is via *polling*. The client JS issues AJAX calls periodically to check for updates. This wastes resources and cannot be used for low latency scenarios.

Real-Time Server Push

Polling can be implemented less naively: The server can hold the connection and only send the response when the new update arrives. If we think further, the chunked encoding could allow for an infinite number of updates in a single response. This is called *long polling*.

For example, the server could generate a stream of JSON messages, separated by new lines, delivered via the chunked encoding. The client only needs to issue a single request, and the messages can be read by the client without the extra latency introduced by polling.

```
 client      server
 ------      ------
|header| ==>
         <== |header|
         ...
         <== |JSON\n|
         <== |JSON\n|
         <== |JSON\n|
         ...
```

However, this use case is obsoleted by WebSocket — an HTTP protocol extension.

WebSocket is Message-Based

There are multiple motivations behind the design of WebSocket. One of them is the *message-oriented* design as opposed to the byte stream. The one-JSON-per-line example sounds simple enough, but you still have to split the byte stream into lines. The No. 1 mistake in coding networked applications is the failure to understand the byte stream. WebSocket comes with a built-in framing format that outputs messages instead of bytes, which saves programmers from repeating rookie mistakes.

WebSocket is NOT Request-Response

Another contribution of WebSocket is bi-directional and *full-duplex* communication, which means that both the client and the server can message each other at the same time. Technically, you can emulate bi-directional messaging[3] without WebSocket via 2 HTTP connections (1 for push and 1 for pull). But WebSocket can make this easier for both the client and the server.

[3] https://en.wikipedia.org/wiki/BOSH_(protocol)

```
 client            server
 ------            ------
|header|  ==>
              <== |header|
          ... ...
|message| ==>
|message| ==> <== |message|
|message| ==>
              <== |message|
|message| ==>
          ... ...
```

We'll implement WebSocket in a later chapter, after we have explored other important uses of HTTP.

CHAPTER 09

File IO & Resource Management

Serving static files is one of the uses for which HTTP was designed. And there are a few more topics to explore:

1. Ranged requests. Fetching a portion of a file. This allows a transfer to be interrupted and resumed later.
2. Caching. To reduce unnecessary transfers.
3. Compression. To reduce transmission time and cost.

We'll start by implementing a basic file server, which will familiarize us with the File API.

9.1 File IO in Node.JS

As mentioned earlier, the file API in Node.JS is available in 3 types:

- The synchronous API, which we cannot use.
- The callback-based API, which we don't want to use.
- The promise-based API, which we'll use.

```
import * as fs from "fs/promises";
```

We'll use 4 file operations:

1. Open a file.
2. `stat` the file to get the metadata such as the size.
3. Read the file data.
4. Close the file.

Like sockets, disk files are represented by opaque handles wrapped in JS objects. Here's the simplified TypeScript API definition we'll use.

```
function open(path: string, flags?: string): Promise<FileHandle>;

interface FileReadResult {
    bytesRead: number;
    buffer: Buffer;
}
interface FileReadOptions {
    buffer?: Buffer;
    offset?: number | null;
    length?: number | null;
    position?: number | null;
}
interface Stats {
    isFile(): boolean;
    isDirectory(): boolean;
    // ...
    size: number;
    // ...
}
interface FileHandle {
    read(options?: FileReadOptions): Promise<FileReadResult>;
    close(): Promise<void>;
    stat(): Promise<Stats>;
}
```

9.2 Serving Disk Files

Step 0: Add a Handler for Static Files

Our first step is to write the code to serve disk files. We will add a handler to serve files from the current working directory.

```
    const uri = req.uri.toString('utf8');
    if (uri.startsWith('/files/')) {
        // serve files from the current working directory
        // FIXME: prevent escaping by `..`
        return await serveStaticFile(uri.substr('/files/'.length));
    }
```

For a production web server, there is additional work to ensure that the URI path is really contained by the intended directory (URI normalization) and that the file is actually accessible. We will skip this work.

Step 1: Open and Close a File

The `'r'` flag is used to open the file in read-only mode. Opening a file also checks if the file is accessible. `fs.open()` will throw an exception if the file doesn't exist or cannot be accessed for other reasons.

```
async function serveStaticFile(path: string): Promise<HTTPRes> {
    let fp: null|fs.FileHandle = null;
    try {
        // open the file
        fp = await fs.open(path, 'r');
        // later ...
    } catch (exc) {
        // cannot open the file or whatever
        console.info('error serving file:', exc);
        return resp404();
    } finally {
        // make sure the file is closed
        await fp?.close();
    }
}
```

Like sockets, disk files must be closed manually, the try-finally block is used to ensure this.

Step 2: `stat()` a File

The `stat()` method is used to retrieve the metadata from a file handle. The result includes the file type (regular file, directory, or other special types), size, time, and other information.

```
        // open the file
        fp = await fs.open(path, 'r');
        // get its size
        const stat = await fp.stat();
        if (!stat.isFile()) {
            return resp404();   // not a regular file?
        }
        const size = stat.size;
```

There is also the `fs.stat(path)` function, which takes a path instead of a handle as an argument. It can `stat` a file by its path without opening it first. However, it is preferable to use `stat` on a file handle. This is because the path can refer to different files. If you check a path with **stat** and then open it later, the path may be replaced by another file (by renaming or deleting), which is a case of a *race condition*[1]. Opening a file first ensures that you are working on the same file.

Step 3: Construct the `BodyReader`

The **readerFromStaticFile()** function returns a **BodyReader** from a file handle. It is important to set the **fp** to null before we return, because the **BodyReader** will read the file later, the **finally** block cannot close the file after this point.

```
async function serveStaticFile(path: string): Promise<HTTPRes> {
    let fp: null|fs.FileHandle = null;
    try {
        // open the file
        fp = await fs.open(path, 'r');
        // ...
        // the body reader
        const reader: BodyReader = readerFromStaticFile(fp, size);
        fp = null;  // the reader is now responsible for closing it instead
        return {code: 200, headers: [], body: reader};
    } catch (exc) {
        // ...
    } finally {
        // make sure the file is closed
        await fp?.close();
    }
}
```

The **read()** function of the **BodyReader** is directly wired to the **fp.read()** method. However, there are a few gotchas.

- The **fp.read()** method will automatically create a buffer if it's not supplied, but surprisingly, it returns the buffer as is, without trimming it to the data size.
- When serving static files, we must make sure that the file we send matches the **Content-Length**. A change in file size is unrecoverable, we can only close the connection in this state.

[1] https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use

```
function readerFromStaticFile(fp: fs.FileHandle, size: number): BodyReader {
    let got = 0;    // bytes read so far
    return {
        length: size,
        read: async (): Promise<Buffer> => {
            const r: fs.FileReadResult<Buffer> = await fp.read();
            got += r.bytesRead;
            if (got > size || (got < size && r.bytesRead === 0)) {
                // unhappy case: file size changed.
                // cannot continue since we have sent the `Content-Length`.
                throw new Error('file size changed, abandon it!');
            }
            // NOTE: the automatically allocated buffer may be larger
            return r.buffer.subarray(0, r.bytesRead);
        },
    };
}
```

Step 4: Make Sure the File is Closed

If the handler function successfully returns a `BodyReader`, the file is still open and must be closed later. We'll add a `close()` function to the `BodyReader` interface to do the cleanup.

```
// an interface for reading/writing data from/to the HTTP body.
type BodyReader = {
    // the "Content-Length", -1 if unknown.
    length: number,
    // read data. returns an empty buffer after EOF.
    read: () => Promise<Buffer>,
    // optional cleanups
    close?: () => Promise<void>,
};
```

The `close()` member is made optional, so we only need to modify the `readerFromStaticFile()` function.

```
function readerFromStaticFile(fp: fs.FileHandle, size: number): BodyReader {
    return {
        // ...
        close: async () => await fp.close(),
    };
}
```

The code after the handler function call is wrapped in a try-finally block to ensure that the cleanup function is called.

```
        const res: HTTPRes = await handleReq(msg, reqBody);
        try {
            await writeHTTPResp(conn, res);
        } finally {
            await res.body.close?.();   // cleanups
        }
```

That's the most rudimentary file server. Time to stop and test.

9.3 Discussion: Manual Resource Management

JS is a GC language, which frees you from the unproductive and error-prone work of manually allocating and freeing memory, However, there are still *non-memory* resources that must be managed manually, such as files and sockets that must be closed. If you have no experience with manual memory management, you probably don't know how to manage non-memory resources in general.

Resources Are Owned by Something

The *ownership* of a resource is the most important concept in manual resource management. That is, who is responsible for terminating and cleaning up the resource. The owner of a resource can be either:

- A function, which terminates the resource before exiting.
- A code scope, which terminates the resource before exiting.
- An object, which terminates the resource when itself is being terminated.

Ownership can be static or dynamic. Dynamic means that the owner changes at run time. No matter what you do, **a resource always has exactly 1 owner**.

Resources Owned by Functions or Scopes

In the `serveStaticFile()` function, the opened file is initially owned by the function, so the function must close it. The entire function is wrapped in a try-finally block, and the cleanup code is placed in the `finally` block at the end of the function.

Other languages have similar constructs for function-level cleanup.

- In Golang, you can use the `defer` function.
- In C, it is common practice to put the cleanup code at the end of the function and `goto` the cleanup code instead of `return`ing.
- In C++, you can use destructors, especially when exceptions are used.
- In Python, you have the `with` block in addition to the try-finally block.

All of these mechanisms, except Go's `defer`, can also be used in a smaller scope than the entire function.

Ownerships Can be Transferred

In the `serveStaticFile()` function, after constructing the `BodyReader` and before returning, the file object is immediately set to null, signifying that the ownership of the file is changed from the function to the `BodyReader` object. The "1 owner" rule is satisfied by maintaining only 1 reference to the resource.

```
const reader: BodyReader = readerFromStaticFile(fp, size);
fp = null;  // the reader is now responsible for closing it instead
```

References to a resource can be classified into owning and non-owning references. You must maintain exactly 1 owning reference. The owning reference is attached to a scope or object, which terminates the resource unless the reference is nullified by a transfer to another owner.

The `readerFromStaticFile()` function in this chapter can never throw an exception, but if it does, it should close the file because it owns the file. A problem with the caller code is that the file reference is not set to null when an exception is thrown, so `fp` is closed twice. A more robust pattern is to use a try-finally block:

```
try {
    const reader: BodyReader = readerFromStaticFile(fp, size);
    return {code: 200, headers: [], body: reader};
} finally {
    fp = null;  // transferred to the BodyReader
}
```

Owners Are Chained

When an object owns a resource, the object itself must be owned, either by another object or by a code block. All object ownership is traced back to a code block, otherwise there will be resource leaks.

In the **`serveClient()`** function, after the handler returns the response object, there is an example of the ownership chain.

```
| serveClient() | ==> | HTTPRes | ==> | BodyReader | ==> | file |
    function              object          object           resource
```

Study the source code to understand how each owner holds the reference and does the cleanup.

How to Do Cleanups in a Generator

A function call will either return or throw, but this is not true for generators. Generators can be suspended and then ignored indefinitely; if you wrap the generator code in a try-finally block, the **`finally`** block may never be executed.

```
// count to 99
async function *countSheep(): BufferGenerator {
    try {
        for (let i = 0; i < 100; i++) {
            // sleep 1s, then output the counter
            await new Promise((resolve) => setTimeout(resolve, 1000));
            yield Buffer.from(`${i}\n`);
        }
    } finally {
        console.log('cleanup!');
    }
}
```

You can test this case with the **`'/sheep'`** URI; if the client disconnects midway, the **`finally`** block will not be executed. Fortunately, there is a way to fix this.

```
function readerFromGenerator(gen: BufferGenerator): BodyReader {
    return {
        // ...
        close: async (): Promise<void> => {
            // force it to `return` so that the `finally` block will execute
            await gen.return();
        },
    };
}
```

The `return()` method[2] of JS generators is used to force a generator to return, as if a `return` statement were inserted at the point where it was suspended (the `yield` statement). This ensures that the `finally` block is executed if there is one. Here we learned a rule for using generators with resource ownerships: **the owner of a generator must ensure that the generator is returned.**

There is also the `throw()` method, which generates an exception at the `yield` statement. This will also execute the `finally` block. You can even use a special type of exception to communicate to the generator that you intend to abort it midway, so that the generator can deal with this particular case.

Python generators have similar methods[3]. And surprisingly[4], the Python garbage collector automatically executes the `finally` block for generators, although it is debatable whether the GC should handle non-memory business.

9.4 Discussion: Reusing Buffers

Buffer Allocations Have Costs

The `readerFromStaticFile()` function allocates and returns a new buffer for each read. This is not very efficient because buffer allocations have costs:

1. The costs from the memory allocator. For many implementations, this cost is *constant* in most cases. In the best case, the allocator either returns an object from a free list or allocates from a large chunk.
2. The cost of initializing with zeros. Scales *linearly* with buffer size.

Using Uninitialized Buffers

Larger buffers are often desirable for potentially higher throughput due to the reduced number of syscalls. However, allocating oversized buffers has a linear initialization cost, which can be avoided by using *uninitialized* buffers in Node.JS.

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator/return

[3] https://docs.python.org/3/reference/expressions.html#generator.close

[4] https://docs.python.org/2.5/whatsnew/pep-342.html

```
const buf = Buffer.allocUnsafe(65536);
```

This function is called **`allocUnsafe`** because a buggy program has a greater chance of leaking sensitive internal information to users. Uninitialized data bugs are also harder to debug due to their unpredictability.

Reusing Large Buffers

Uninitialized buffers are not easily obtainable in some languages, such as Go or Python. What you can do is reuse the same buffer for multiple IO operations.

```
function readerFromStaticFile(fp: fs.FileHandle, size: number): BodyReader {
    const buf = Buffer.allocUnsafe(65536);  // reused for each read
    return {
        length: size,
        read: async (): Promise<Buffer> => {
            const r = await fp.read({buffer: buf});
            // ...
            return r.buffer.subarray(0, r.bytesRead);
        },
    };
}
```

This will amortize the cost of the buffer allocations.

Pooled Buffers

Instead of allocating buffers locally, you can have a global pool of *used* buffers. Whenever you are done with a buffer, you put it into that pool. And when you need a buffer, you try to grab one from the pool first, and only create a new one if the pool is empty.

The advantages of this are:

- Grabbing stuff from the pool can cost even less than the memory allocator.
- Different code paths all benefit from the pool. There are fewer real allocations overall.

An object pool is trivial to code. The Node.JS **`Buffer`** type uses a built-in pool[5], but only for *small* buffers. The Go standard library also includes an object pool[6].

[5] https://nodejs.org/api/buffer.html#static-method-bufferallocunsafeslowsize
[6] https://pkg.go.dev/sync#Pool

Mind the Lifetime When Reusing Buffers

Buffers can be allocated by either the producer or the consumer. In all cases we have seen, the producer allocates the buffer and returns it to whoever uses the data. This can be problematic because the buffer may still be referenced and used somewhere while the next `read()` is running.

```
function readerFromStaticFile(fp: fs.FileHandle, size: number): BodyReader {
    const buf = Buffer.allocUnsafe(65536);  // reused for each read
    return {
        length: size,
        read: async (): Promise<Buffer> => {
            const r = await fp.read({buffer: buf});
            // CAUTION: the lifetime of the buffer is unclear!
            return r.buffer.subarray(0, r.bytesRead);
        },
    };
}
```

Let's apply the lesson from the resource management discussion: who owns the buffer? For the code in this chapter, we know that the buffer returned by the producer (`BodyReader`) is written to the socket, and our code waits for the write to complete before invoking the next `read()`, so the producer is the only place the buffer is referenced, so we can reason that the consumer doesn't need to own the buffer, so the buffer reuse is valid. However, this may not be true for more complicated consumers!

Another case is when the consumer allocates the buffer, the lifetime of the buffer is clear as long as the data is processed in place so the consumer is the sole owner.

```
// pseudo code!
const buf = Buffer.allocUnsafe(65536);      // reused for each read
while (more_data()) {
    const nbytes = await read_data(buf);    // read from the producer
    await process(buf.subarray(0, nbytes));
    // `buf` is not used anymore, so it's safe to reuse.
}
```

In Golang, when pulling data from the `io.Reader` interface, the buffer is provided by the consumer. This encourages buffer reuse and discourages unnecessary allocations by design.

CHAPTER 10

Range Requests

The next step is to add range requests to our file server, which will allow resumeable transfers.

10.1 How Range Requests Work

The `Range` Header Field Syntax

A range request is initiated by the client with the **`Range`** header field. For example, **`Range: 0-9`** retrieves the first 10 bytes of the payload. The header field is fully specified in RFC 9110[1].

```
ranges-specifier = range-unit "=" range-set
range-set        = 1#range-spec
range-spec       = int-range
                 / suffix-range
                 / other-range

int-range     = first-pos "-" [ last-pos ]
first-pos     = 1*DIGIT
last-pos      = 1*DIGIT
suffix-range  = "-" suffix-length
suffix-length = 1*DIGIT
```

`range-unit` is always **`bytes`**, **`range-set`** is a comma-separated list of ranges, and there are 2 possible formats for a single range:

1. An inclusive interval like **`12-34`**, which means the range from the 12th byte to the 34th byte.
2. A negative integer like **`-12`**, which means the last 12 bytes.

The Effective Range with Examples

The effective range is the intersection with the real file range. Suppose we have a 50-byte file, here are some examples and their effective ranges.

`Range: bytes=x`	Effective range
0-0	[0, 1)
0-1	[0, 2)
10-	[10, 50)
10-60	[10, 50)

[1] https://www.rfc-editor.org/rfc/rfc9110.html

`Range: bytes=x`	Effective range
0-60	[0, 50)
4-3	invalid
70-60	invalid
-10	[40, 50)
-60	[0, 50)
-0	invalid
--1	invalid
foobar	invalid
60-	out of range
60-70	out of range

The server must also handle unhappy cases:

- If the intersection is empty, the server responds with status code 416 (Range Not Satisfiable).
- If any of the ranges are invalid, the server ignores the header field (and serves the full file).

Range Requests Are Optional

Once you start adding features to your HTTP server, you'll find that most HTTP features are optional, and optional features can make things more complicated than they seem.

A server may not support range requests at all, and even if it does, they are not applicable to all requests. For example, it makes no sense to return a ranged response for dynamically generated content, because the length of the content is not even known in advance, and the server cannot tell if the request is out of range.

Range Responses Are Indicated by 206

How does a client know if the server supports range requests? That's the job of the 206 (Partial Content) status code, which indicates a partial response instead of a full one (200 OK).

`Content-Range` Returns the Effective Range

As seen in the examples above, the effective range may be different from the requested range, so the server needs to tell the client the exact range returned. That's the job of the `Content-Range` header field. It contains the effective range of the response, along with the total content length.

```
Content-Range: bytes 12-34/1234
```

This header field is also used for 416 (Range Not Satisfiable) responses to tell the client the correct length of the content.

```
Content-Range: bytes */1234
```

Announcing `Range` Support

There are ways to let clients know that range requests are available. The server can include the **`Accept-Ranges: bytes`** header field in a non-range response to indicate that this URI supports range requests, so the client knows that it can resume the download with a range request if interrupted.

Probing with the `HEAD` Method

And there is a way to tell if range requests are supported *without* retrieving any part of the response — the **`HEAD`** HTTP method.

If the server supports the **`HEAD`** method, it should behave like a **`GET`** but *without* the response body, returning the response header field and the status code as if the request is being fully served. The client can then check for **`Accept-Ranges`** or **`Content-Range`** without actually receiving the content.

Remember that the way payload length is determined in HTTP is messy. The **`HEAD`** method adds a special case for clients: The server may return the **`Content-Length`**, but it doesn't tell the payload length this time; the **`HEAD`** method has no response body, regardless of what the header fields say!

Multiple Ranges with the Multipart Message

Although rarely used, the **`Range`** header field can be multiple ranges separated by commas. Here is an example.

```
GET / HTTP/1.1
Range: bytes=0-5,8-13

HTTP/1.1 206 Partial Content
Accept-Ranges: bytes
Content-Length: 227
```

```
Content-Type: multipart/byteranges; boundary=SEPARATOR1234567

--SEPARATOR1234567
Content-Type: text/html; charset=UTF-8
Content-Range: bytes 0-5/1234

<html>
--SEPARATOR1234567
Content-Type: text/html; charset=UTF-8
Content-Range: bytes 8-13/1234

<head>
--SEPARATOR1234567--
```

The response is completely different from the single-range response:

- `Content-Range` is not used. Instead, it uses `Content-Type: multipart/byteranges; boundary=x`.
- The response is encoded in a new message format:
  - Indicated by the `Content-Type` header field.
  - Multiple parts are separated by a random string delimiter.
  - Each part is a header-payload structure, similar to HTTP itself. The header contains the `Content-Range` header field for each part.

The `Content-Length` header field still serves the same purpose, which is the length of the payload (the multipart message), not the sum of the ranges.

Here we learned a new idea to separate data — random string delimiters. The delimiter should be long enough so that the payload is unlikely to contain it. The maximum length is 70 bytes according to RFC 2046[2]. Still, delimiters are generally not a good idea.

Discussion: Is Multipart Message Technically Necessary?

Why not simply concatenate multiple ranges without this message encapsulation? This can work if `Content-Range` could return the range list.

While many computer problems are solved by an extra encapsulation, it's also common to see encapsulations that do nothing.

[2] https://datatracker.ietf.org/doc/html/rfc2046

Possible Cases for Clients

A summary of range requests from the client's point of view:

- 200 OK. A full response.
- 206 Partial Content.
  - A partial response with `Content-Range`.
  - A multipart message. Expect `Content-Type: multipart/byteranges; boundary=x`.
- 416 Range Not Satisfiable. Out of range.

10.2 Implementing Range Requests

Step 1: Parsing the Range Header Field

This step is just to follow the RFC, we'll skip the code listing as it's not very interesting.

```
// range-spec       = int-range
//                  / suffix-range
//                  / other-range
// int-range     = first-pos "-" [ last-pos ]
// suffix-range  = "-" suffix-length
type HTTPRange = [number, number|null] | number;

function parseBytesRanges(r: null|Buffer): HTTPRange[];
```

Step 2: Reading a Portion of a File

We'll modify the `readerFromStaticFile()` function to read a portion of a file instead of the entire file.

```
// read from [start, end)
function readerFromStaticFile(
    fp: fs.FileHandle, start: number, end: number): BodyReader;
```

You can read from the desired file position using the `position` argument of the `fp.read()` method. File IO APIs often include a `seek()` method to set the read/write position, this is unnecessary in Node.JS because the position can be specified in the `read()` method, and the underlying OS interface is a single syscall or API call in popular operating systems anyway.

```
            const maxread = Math.min(buf.length, end - offset); // may be 0
            const r: fs.FileReadResult<Buffer> = await fp.read({
                buffer: buf, position: offset, length: maxread,
            });
```

The rest of the code is omitted. Remember to apply the discussion from the last chapter when coding this.

Step 3: Generating Range Responses

We're going to add a lot of code. It's time to extract a new function from **`serveStaticFile()`**. Remember to use the resource management guide from the last chapter.

```
        try {
            return staticFileResp(req, fp, size);
        } finally {
            fp = null;  // transferred to staticFileResp()
        }
```

Inside the new **`staticFileResp()`** function, we need to do the following:

1. Check the **`Range`** header field and compute the effective range.
    - Return 416 (Range Not Satisfiable) if the requested range does not intersect with the file.
2. Add the **`Content-Range`** header field that contains the effective range.
3. Respond with 206 (Partial Content).

This is the code path for a single-range response. We'll skip the case for multiple ranges because there is nothing more to learn.

Step 4: Adding the `HEAD` Method

In the **`serveClient()`** function, just don't write the HTTP body for the **`HEAD`** method. The **`writeHTTPResp()`** function is split into 2 to make this possible.

```
        const res: HTTPRes = await handleReq(msg, reqBody);
        try {
            await writeHTTPHeader(conn, res);
            if (msg.method !== 'HEAD') {    // omit the body
```

```
                await writeHTTPBody(conn, res.body);
            }
        } finally {
            await res.body.close?.();       // cleanups
        }
```

The **BodyReader** is still closed regardless, because in our case the **BodyReader** is still owning a file resource. You can also handle the **HEAD** method in the request handler and close the file earlier (and not create the **BodyReader** in the first place).

CHAPTER 11

HTTP Caching

We'll explore HTTP caching in this chapter, which reduces unnecessary transfers.

11.1 Cache Validator

A web browser will cache responses by default, even without much effort from the server. The problem is: how do clients know if a cached item is valid? There are mechanisms to query the server for this information.

Validate by Timestamp

One way to do this is through the `Last-Modified` and `If-Modified-Since` header fields. This works in the following steps:

1. The server returns the filesystem modification time of the file in the `Last-Modified` header field.
2. The client caches the response along with the timestamp.
3. The client needs to validate the cached response before reusing it by sending the request with the `If-Modified-Since` header field set to the cached timestamp.

   - If the file has been modified on the server (indicated by a different timestamp), the server returns a response as usual.
   - If the server doesn't support this validation method and ignores the header field, it also returns as usual.
   - If the timestamp is the same on the server, the server returns the status code 304 (Not Modified), this status code has no payload body, it just tells the client to reuse the cache.

4. The client will reuse the cached response if the status code is 304, or use the normal response and update its cache.

Validate by ETag

The resolution of the timestamp is only 1 second, which is inadequate for some use cases, and filesystem timestamps are not a reliable way to identify the content anyway. Another way to identify the content is through the `ETag` header fields. This works the same way as `Last-Modified`, except that its value is arbitrary, you can use the hash of the content, or keep a version number whenever you update the content.

`ETag` is validated by `If-None-Match`, which is analogous to `If-Modified-Since`.

Validate for Range Requests

The above 2 validations work for the scenario where the client has cached an old version and expects the server to return the full response only with a newer version. However, the scenario for range requests is different: the server can only return the partial response if the content is the same as what the client already has. In this case, the client cannot use `If-Modified-Since` or `If-None-Match`; instead, it uses the `If-Range` header field, which contains either an `ETag` or a timestamp. The server must ignore the `Range` header field if the `If-Range` value *does not* match (resulting in a full response).

Summary of Validation Headers

Server to client	Client to server	Comment
Last-Modified	If-Modified-Since	Timestamp.
ETag	If-None-Match	Hash, version number, etc.
ETag/Last-Modified	If-Range	Partial or full?

Implementing the Timestamp Validator

As a simple file server, adding the `Last-Modified` and `If-Modified-Since` header fields is trivial.

```
const ts = Math.floor(stat.mtime.getTime() / 1000); // modified ts
const headers: Buffer[] = [
    // indicate the support for range requests
    Buffer.from('Accept-Ranges: bytes'),
    // for cache validation
    Buffer.from(`Last-Modified: ${stat.mtime.toUTCString()}`),
];
// check conditions
const ifm = fieldGet(req.headers, 'If-Modified-Since');
if (ifm && parseHTTPDate(ifm.toString('latin1')) === ts) {
    const empty = readerFromMemory(Buffer.from(''));
    return {code: 304, headers: headers, body: empty};
}
let hrange = fieldGet(req.headers, 'Range');
const ifr = fieldGet(req.headers, 'If-Range');
if (ifr && parseHTTPDate(ifr.toString('latin1')) !== ts) {
    hrange = null;  // ignore the `Range` field
}
// use `hrange` to check the requested range ...
```

The `ETag` scheme will likely require a separate subsystem to keep track of the content, whether it's a hash value or a version number, so we won't code this.

11.2 Discussion: Server-Side Cache Control

We have talked about how clients validate their cache, the next thing is to control the caching behavior from the server side. That is, how long the clients should keep the cache.

The server uses the `Cache-Control` header field to advise the client about caching. Its value is a comma-separated list of directives. For example:

```
Cache-Control: no-store, no-cache, max-age=0, must-revalidate
```

When to Revalidate Cache

The `max-age=x` directive specifies how long the cached item will live in seconds (time-to-live, or TTL). However, the cache does not simply delete the item when it expires. It's complicated, so complicated that it has its own document: RFC 9111 — HTTP Caching[1].

Cached items are described as *fresh* or *stale* depending on whether they are older than their `max-age`. The fresh or stale state determines when the item should be validated. This also depends on other cache directives, as shown in the table.

`Cache-Control` from server	Validate fresh	Validate stale	TTL
`max-age=x`	maybe	should	`x`
`max-age=x, must-revalidate`	maybe	must	`x`
`max-age=x, immutable`	should not	should	`x`
`max-age=0, must-revalidate`	(not fresh)	must	`0`
`no-cache`	(not fresh)	must	`0`
(not present)	heuristic	heuristic	heuristic

Control the TTL

The `max-age` directive alone doesn't give much control over the lifetime of the cache as seen from the table, because the table uses the ambiguous words "maybe" and "should", what's the difference between them?

[1] https://datatracker.ietf.org/doc/html/rfc9111

A browser will normally reuse a fresh item, but it MAY revalidate the item even if the cache is fresh, this can happen when the user hits the reload button.

But why is "Validate stale" a "should"? What's the point of setting the TTL if the effect is not even guaranteed? It turns out that there are 2 levels of guarantee.

1. Without `must-revalidate`, a stale item may still be reused, this can happen when the server is unreachable.
2. With `must-revalidate`, the cache TTL is actually respected.

Implementations do not necessarily distinguish between "should" and "must" cases, though.

Make the Client Always Check the Server

The `no-cache` directive is confusing, it does not mean "do not cache", it's a shorthand for `max-age=0, must-revalidate`, which means "cache but always validate". It is used when the latest data is desired.

Prevent Caching

To prevent caching altogether, use the `no-store` directive, which instructs the client not to touch the cache at all. (This also won't cause the existing cache entry to be deleted.)

Reduce Fresh Validations

The `immutable` directive is an optimization; it tells the client not to revalidate a fresh item, even when reloading. This can be used when the URL is immutable, such as when the resource is retrieved by its hash value. Not all browsers have implemented this, though.

Heuristic Caching

Even with these directives, the client has lots of freedom. Without real use cases, it's impossible to implement *useful* software just from specifications. The freedom of clients is maximized when the server does not use `Cache-Control` at all. This is called heuristic caching[2] in web browsers.

For example, a browser may use the `Last-Modified` time as a heuristic, using a fraction of the last-modified-to-now duration as the cache TTL.

[2] https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching#heuristic_caching

Non-Client Caching

So far, we have assumed that caching is done by a user-facing client, such as a browser. In practice, a cache can also be provided by:

- By the server application. This is especially useful for dynamically generated content. The app stores the generated content for itself to reuse. The goal is to save the latency and/or the cost of content generation. The `ETag` scheme can also be easily used to aid client-side caching, since the app controls the content.
- By a transparent proxy or middleware. The proxy acts as a client of the origin server, serving requests on behalf of the real client. It may cache responses just like other clients. This form of proxy can be deployed by the service developer, by the CDN, or by the ISP (before HTTPS was common).

Shared and Private Caches

A problem with the caching proxy is that the response may be personalized, how does a caching proxy know which content is for which user?

This problem does not exist for a browser because it represents a single user. For caches shared by multiple users, there are directives to explicitly mark the response as either `public` or `private`. The `private` directive is for personalized content that should not be cached by proxies.

```
Cache-Control: private
```

There are additional rules for determining whether to cache or not. For example, the `Authorization` header field normally prevents a proxy from caching the response. But the `public` directive can override this rule and make the content cacheable.

Caching in the Cloud

The RFC only defines the intended caching behavior in a very loose sense. For developing or understanding real-world projects, you should rely on the software/service manual instead.

There are many cloud providers that offer a CDN or caching service, some even offer additional caching controls. Read their documentation to understand how things work in *practice*. Some examples:

- Cloudflare[3]
- AWS CloudFront[4]

[3] https://developers.cloudflare.com/cache/concepts/cache-control/
[4] https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Expiration.html

- Azure CDN[5]
- Google Cloud CDN[6]

[5] https://learn.microsoft.com/en-us/azure/cdn/cdn-how-caching-works

[6] https://cloud.google.com/cdn/docs/caching

CHAPTER 12
Compression & the Stream API

The next major feature is compression. We'll also explore the stream abstraction in Node.JS (which we ignored previously).

12.1 How HTTP Compression Works

Negotiating with `Accept-Encoding`

HTTP compression, like other features, is optional, so the client have to explicitly request the server to compress the response. This is done via the `Accept-Encoding` header field, which contains a comma-separated list of compression methods. Commonly used methods include:

- `gzip`
- `deflate`
- `br`. Brotli[1]; newer but less widely supported.

The method may be followed by an optional weight to indicate its priority. See the RFC for the exact syntax.

```
Accept-Encoding: deflate, gzip;q=1.0
```

Compress with `Content-Encoding`

If the server chooses one of the suggested compression methods, it will report back with `Content-Encoding`.

```
Content-Encoding: gzip
```

More than one `Content-Encoding` can be applied. However, there is no practical use for this. Some content types are already compressed, such as PNG, JPG, or ZIP. It's best to exclude them from compression, as they are not compressible.

[1] https://www.rfc-editor.org/rfc/rfc7932.html

Compression and `Content-Length`

The purpose of `Content-Length` does not change with respect to compression, that is the length of the HTTP body, not the length of the uncompressed data.

When compression is applied on the fly, the compressed size is not known in advance (unless the entire output is buffered, which does not work for large content), so we need to also use chunked encoding.

Compression and Caching

Compression is CPU intensive, it is not always applied on the fly; a caching proxy can cache and serve compressed responses (if the `Accept-Encoding` allows it).

If a caching proxy doesn't care about compression, it must at least not mix responses from different compression methods. That's the purpose of the `Vary` header field, to inform proxies that the response will vary if certain header fields are used.

For example, if `Vary: content-encoding` is used, the cache item will be keyed with the value of `Content-Encoding`, so the cache lookup will take that into account.

Compressed Uploads

Due to the nature of content negotiation, there is no standard way to compress the request body. If a client does this with `Content-Encoding`, it is probably the application, rather than the HTTP server, to decompress the request.

Compression and Range Requests

`Content-Encoding` is a property of content. Therefore, the `Range` header field works on the compressed data, which is NOT the desired behavior.

Common practice is not to compress when serving range responses.

Compress with `Transfer-Encoding`

There is another rarely[2] supported[3] way to do compression. `Transfer-Encoding` is usually used for chunked encoding, it can also compress the payload.

```
Transfer-Encoding: gzip, chunked
```

[2] https://bugs.chromium.org/p/chromium/issues/detail?id=94730
[3] https://bugzilla.mozilla.org/show_bug.cgi?id=68517

The above header field means that the payload is first gzipped and then chunked.

What's the difference from `Content-Encoding`? The difference is that `Transfer-Encoding` is NOT a property of the content, so range requests can work with `Transfer-Encoding` but not with `Content-Encoding`.

Unlike other obscure HTTP features, compression with `Transfer-Encoding` is actually more practical and thus superior, but unfortunately implementations have gone the other way.

The `TE` header field is for negotiating `Transfer-Encoding`, as summarized below.

Table 1: HTTP compression negotiation.

Request header	Response header	`Range`	Widely used?
`Accept-Encoding`	`Content-Encoding`	No	Yes
`TE`	`Transfer-Encoding`	Supported	No

12.2 Data Processing with Pipes

Data Processing and IO

We'll use the built-in zlib[4] module to compress the response body. There are 2 types of API for compressing data, the simplest is to supply the input as a single buffer and return the compressed data as a single buffer. This won't work if the input is large because it may not fit in memory. And the latency will be bad even if memory is not a concern.

```
zlib.gzip(buffer, callback);
```

The second type is to produce output *on the fly*, you first initiate the stateful compressor object, then you feed the compressor input, the compressor will produce output simultaneously.

```
const obj = zlib.createGzip();  // read from it & write to it
```

The compressor has backpressure, so it won't blow up memory if used properly. And you must drain the output simultaneously to keep the input flowing.

The way to process data on the fly is with stateful processors that can *read* and *write* simultaneously, which is similar to what we do with a socket.

[4] https://nodejs.org/api/zlib.html

Unix Pipes for Composable Data Processors

It's common to have multiple data processing steps, and Unix pipes are a *high-level* way to program and compose these steps. A concrete example:

```
tar -c -f - ./directory | gzip | nc 1.2.3.4 1234
```

It has 3 steps connected by 2 pipes:

1. `tar`, the producer of the archive.
2. `gzip`, consumes the archive and produces the gzip data.
3. `nc`, consumes the gzip data and sends it over the network.

A pipe automatically connects an output to an input, without the user writing the code to move data between steps.

Abstract Pipes

We could use a pipe-like abstraction in our HTTP server:

```
produce-response | gzip-compress | chunk-encode | write-to-socket
```

There are advantages of this pattern:

- No need to explicitly read and write data; less code to write!
- Backpressure is automatic (if each processor is implemented properly).
- It's less error-prone, as we will see.

12.3 Exploring the Stream API in Node.JS

List of Stream Interfaces

Pipes do exist in Node.JS, you can pipe stuff with the Stream API[5].

```
import * as stream from "stream";
```

It defines a set of interfaces that you can implement.

[5] https://nodejs.org/api/stream.html

Name	Description	Example
`Readable`	Consume output from this.	`fs.createReadStream()`
`Writable`	Feed input to this.	`fs.createWriteStream()`
`Duplex`	`Readable` + `Writable`.	`net.Socket`
`Transform`	Extends `Duplex`.	`zlib.createGzip()`

`Readable` and **`Writable`** are the 2 basic abstractions. Both **`Duplex`** and **`Transform`** are a combination of **`Readable`** and **`Writable`**, how they differ is not relevant to us at this point.

Some built-in Node.JS modules already implement them, as seen in the table.

Using Pipes on Stream Interfaces

Our goal in exploring these abstractions is to use pipes on them.

```
import { pipeline } from "stream/promises";
```

The **`pipeline()`** function connects a **`stream.Readable`** to a **`stream.Writable`**. After creating a pipe, data automatically flows from a producer to a consumer, and backpressure is present, just like in a real Unix pipe. This function is named after Unix pipes, do not confuse it with request pipelining.

```
function pipeline(src: stream.Readable, dst: stream.Writable): Promise<void>;
```

You can also put one or more **`stream.Duplex`** between the source and destination.

```
function pipeline(
    src: stream.Readable,
    transform1: stream.Duplex,
    transform2: stream.Duplex,
    // more ...
    dst: stream.Writable,
): Promise<void>;
```

12.4 Implementing HTTP Compression

The **`zlib`** module implements **`stream.Duplex`** for compression objects. It’s a great opportunity to learn how to use the Node.JS Stream API.

Step 1: Check Header Fields and Enable Compression

Do this according to our understanding of header fields. We'll unconditionally gzip the response if requested by the client. It's also a good idea to let applications control when not to use compression.

```
function enableCompression(req: HTTPReq, res: HTTPRes): void {
    // inform proxies that the response is variable
    res.headers.push(Buffer.from('Vary: content-encoding'));
    // check header fields
    if (fieldGet(req.headers, 'Range')) {
        return; // incompatible
    }
    const codecs: string[] = fieldGetList(req.headers, 'Accept-Encoding');
    if (!codecs.includes('gzip')) { // TODO: parse the weight: `gzip;q=0`
        return;
    }
    // transform the response using gzip
    res.headers.push(Buffer.from('Content-Encoding: gzip'));
    res.body = gzipFilter(res.body);
}
```

The **gzipFilter()** function is what we'll implement next, which returns a wrapper of a **BodyReader** to compress the data.

Step 2: Understanding the Gotchas without Pipes

Without using the **pipeline()** function, you may simply image the solution like this:

```
// pseudo code!
function gzipFilter(reader: BodyReader): BodyReader {
    const gz: stream.Duplex = zlib.createGzip();
    return {
        length: -1,
        read: async (): Promise<Buffer> => {
            const data = await reader.read();
            await write_input(gz, data);    // deadlock by backpressure!
            return await read_output(gz);   // deadlock by buffering!
        },
    }
}
```

Unfortunately, this pseudo code does not work for 2 reasons.

1. The compressor implements backpressure, which means that if you write a large piece of input, the code will block until the input is processed and *drained*. But in the above pseudo code, the data is drained *after* the write is completed, which is a case of deadlock! (We discussed a similar deadlock in the "Pipelined Requests" section.)
2. Most data compression methods inherently require buffering; 1 data input to the compressor does not result in 1 compressed output. Without enough data, the compressor won't generate output because it hasn't decided how to compress the data. Reading from the compressor will stuck when it is expecting more data!

In a Unix pipe, reading and writing happen in different processes, the pipe is fed and drained *simultaneously*, so there are no deadlocks like in the above solution. That's why I mentioned that the pipe abstraction is less error-prone to use.

Step 3: Implementing `stream.Readable`

We're back on track. Before we can use the `pipeline()` function, we must implement the `stream.Readable` interface for the data source `BodyReader`.

Here is how to implement a `stream.Readable`:

1. A `stream.Readable` has an internal queue, use the `push()` method to add data to the queue, then the data is available for consumption.
2. When should you add data to the queue? You need to implement a `read()` callback, which is invoked when the queue is empty and someone is consuming. This is essential for maintaining backpressure.
3. Use the `destroy()` method to propagate an error to the consumer.

```
function body2stream(reader: BodyReader): stream.Readable {
    let self: null|stream.Readable = null;  // make TS happy
    self = new stream.Readable({
        read: async () => {
            try {
                const data: Buffer = await reader.read();
                self!.push(data.length > 0 ? data : null);
            } catch (err) {
                self!.destroy(err instanceof Error ? err : new Error('IO'));
            }
        },
    });
    return self;
}
```

Note: The `push()` method uses `null` to mark the end of the stream, which is different from what we are using (0-length buffer).

Step 4: Creating a Pipe

Let's try to create a pipe between a `BodyReader` and the compressor.

```
function gzipFilter(reader: BodyReader): BodyReader {
    const gz: stream.Duplex = zlib.createGzip();
    const input: stream.Readable = body2stream(reader);
    (async () => {
        try { await pipeline(input, gz); }
        catch (err) { gz.destroy(err); }
    })();   // not awaiting it
    // return the wrapped `BodyReader` ...
}
```

The `pipeline()` function will move data for us, and it returns a promise for us to `await` on it. We must also catch any exceptions while `await`ing on a promise, and the caught error must be propagated to the next stream, otherwise its consumer will hang forever.

In the `gzipFilter()` function, we cannot wait for the pipe to complete because we must return a wrapped `BodyReader` to the caller, so the try-catch block is wrapped in an anonymous async function call without `await`. This can be simplified by registering an error handling callback to the promise.

```
    pipeline(input, gz).catch((err: Error) => gz.destroy(err)); // no await
```

Step 5: Reading from `stream.Readable`

Now that the pipe is running, we can read the output from the compressor. Remember that a `net.Socket` is also a `stream.Readable`; the `soRead()` function can be used without modification (except for type annotations) because various socket events such as `'data'` and `'end'`, `pause()` and `resume()` methods, are actually defined on `stream.Readable`.

But the `stream.Readable` interface already has a promise-based method for reading. Its `iterator()` method returns an `AsyncIterator`, which you can use in a `for await...of` loop like a generator. The `AsyncIterator` also has a `next()` method that is similar to a generator.

```
function gzipFilter(reader: BodyReader): BodyReader {
    // ...
    const iter: AsyncIterator<Buffer> = gz.iterator();
    return {
        length: -1, // the compressed "Content-Length" is not known
        read: async (): Promise<Buffer> => {
            const r: IteratorResult<Buffer, void> = await iter.next();
            return r.done ? Buffer.from('') : r.value;
        },
        close: reader.close,
    };
}
```

The returned `BodyReader` wrapper reads from the compressor using the new method. If not for learning, we could have skipped the `soRead()` function.

Step 6: Test It

We're almost done, let's enable compression for all responses and test it.

```
        const res: HTTPRes = await handleReq(msg, reqBody);
        try {
            enableCompression(msg, res);
            await writeHTTPHeader(conn, res);
            if (msg.method !== 'HEAD') {    // omit the body
                await writeHTTPBody(conn, res.body);
            }
        } // ...
```

`curl` can request compressed responses:

```
curl -vvv --compressed http://127.0.0.1:1234/
```

Step 7: Flush the Compressor Buffer

Compression works. Except for one problem: when you test the `'/echo'` or the `'/sheep'` URI, the response doesn't show up immediately. This is due to the buffering done by the compressor. From the "Buffered Writer" discussion, we know that there must be a way to *flush* the internal buffer to force the compressor to output immediately.

```
gz.flush(callback);
```

But since we are using `pipeline()` instead of explicit IO, there is no place to insert the `flush()` call. However, there is an option to make the compressor automatically flush every input.

```
const gz = zlib.createGzip({flush: zlib.constants.Z_SYNC_FLUSH});
```

Flushing the compressor frequently reduces the effectiveness of the compression. So it's a good idea to let applications control whether to flush or not. For example, we don't need to flush the compressor when serving static files because the data source is generated as fast as the system can.

For applications that generate very small chunks of data, it may be better not to compress at all, because small data likely won't compress well with frequent flushing.

12.5 Discussion: Refactoring to Stream

You can try to refactor the code to use more streams and pipes. For starters, replace the `read()` function in the `BodyReader` interfaces.

```
// an interface for reading/writing data from/to the HTTP body.
type BodyReader = {
    // the "Content-Length", -1 if unknown.
    length: number,
    // read data.
    reader: stream.Readable,
};
```

Lots of code can be saved by using more streams and pipes:

1. The implementation of the `soRead()` function is not needed.
2. The code for reading disk files can be replaced by `fp.createReadStream()`.
3. The `writeHTTPBody()` function can simply create a pipe between the response and the socket.
4. Since we are no longer dealing with explicit reads and writes, the chunked encoder can be replaced by implementing a `stream.Transform`,
5. Generators can be converted to `stream.Readable` via `stream.Readable.from()`.

And the whole process of responding to a request becomes a series of pipes:

```
produce-response | gzip-compress | chunk-encode | write-to-socket
```

12.6 Discussion: High Water Mark and Backpressure

One-Item Queue

The way we maintain backpressure is by waiting for the queue or buffer to drain. From the producer and consumer point of view, a queue has 2 states:

Queue	Can produce	Can consume
Empty	Yes	No
Not Empty	No	Yes

This limits a queue to a maximum of 1 item or 1 pending write.

Backpressure by High Water Mark

However, some backpressure implementations are slightly more sophisticated — the queue is limited by the number of bytes instead of just 1 write. This limit is often called a *high water mark*. The queue has 3 states when using a high water mark for backpressure:

Queue	Can produce	Can consume
Empty	Yes	No
Half	Yes	Yes
Full	No	Yes

This is more like Unix pipes, a Unix pipe is a bounded buffer, as long as it's not full you can add more bytes to it; you do not have to wait until it's completely drained.

Batching Data with Queues

The reason for using the high water mark instead of the one-item queue is that the queue can combine multiple small writes. For example, here are some possible optimizations when moving data from a queue to a socket:

- The queue could be a single buffer; whenever you write to it, the data is appended to the buffer. This automatically combines multiple small writes, and all the data is sent to the socket in one go.
- The queue stores individual buffer objects, multiple buffers can be combined into a single larger buffer before being sent to the socket.
- The queue stores individual buffer objects, the `writev()`[6] syscall can be used to send multiple buffers in one go without manually combining them.

All of these optimizations implement the semi-transparent buffering discussed earlier. See also: `stream.Writable._writev`[7].

High Water Mark in Node.JS

Both `stream.Readable` and `stream.Writable` have a built-in high water mark that can be changed by the `highWaterMark` option when implementing them. And it's often desirable to use larger values for high throughput applications.

```
const r = new stream.Readable({
    highWaterMark: 4096,  // also for `Writable`
    read: async () => { /* produce data and feed it to r.push() */ },
});
```

When using the high water mark for backpressure, check the return value of `Readable.push()` and `Writable.write()`:

- They return `false` when the queue is full, so you can wait for it to drain.
- Otherwise, you can push more data into it.

We didn't use their return values because we simply treated the high water mark as 0 and drained the queue after each write.

Table 5: High water mark in Streams.

Interface	Backpressure indicator	Wait until queue is drained
`Readable`	`.push(data)`	Via the `read()` callback
`Writable`	`.write(data, cb)`	Via the `'drain'` event.

Stream interfaces support, but do not mandate backpressure; you can still push data into them even if the queue is full, which is a footgun for beginners.

[6] https://linux.die.net/man/2/writev
[7] https://nodejs.org/api/stream.html#writable_writevchunks-callback

CHAPTER 13

WebSocket & Concurrency

Let's explore something different — WebSocket (RFC6455[1]). We will also take a closer look at producer-consumer problems as they are common in networked applications.

13.1 Establishing WebSockets

High Level Overview

We have already discussed WebSockets in the "Dynamic Content" chapter. From a user's point of view, a WebSocket is:

- A bi-directional, full-duplex channel.
- Message-based instead of a byte stream.
- An extension to existing HTTP servers.

Upgrade From HTTP/1.1

A WebSocket starts with an HTTP header, that's the only thing it has in common with HTTP. The client uses the **`Upgrade: websocket`** header field to indicate that it intends to create a WebSocket, and the server responds with this header and status code 101 to indicate that the WebSocket is established. The rest of the connection is then taken over by the WebSocket protocol.

An example:

```
GET /chat HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

If a server knows nothing about WebSocket, it will respond like a normal HTTP request. This is one way to *extend* an existing protocol. It is also used to switch from HTTP/1.1 to HTTP/2.

[1] https://datatracker.ietf.org/doc/html/rfc6455

WebSocket Handshake

There are some extra headers besides `Upgrade: websocket`.

- The client must send a random string via `Sec-WebSocket-Key`.
- The server must respond with the hash of the string via `Sec-WebSocket-Accept`.

The rationales for this handshake are:

1. To prove that the server understands WebSocket.
2. To prevent non-WebSocket clients from creating WebSockets by forbiding the `Sec-*` header fields.
3. To prevent caching proxies from caching WebSocket data by using random values.

You may find these points confusing when you think about it, because these points are weak.

1. The server already proved itself with the 101 status code.
2. If a non-WebSocket client wants to prevent users from creating WebSockets, it will likely just forbid the `Upgrade` header field instead.
3. Proxies are unlikely to cache the 101 status code.

The hash is computed by the following function.

```
function wsKeyAccept(key: Buffer): string {
    return crypto.createHash('sha1')
        .update(key).update("258EAFA5-E914-47DA-95CA-C5AB0DC85B11")
        .digest().toString('base64');
}
```

The `Connection` Header Field

The `Connection: Upgrade` header field is used to instruct non-transparent proxies to consume and remove the `Upgrade` header field so that a non-transparent proxy will not forward WebSocket handshakes unless it understands the protocol.

This is one of the hop-by-hop[2] header fields. We do not care about this since we're the server, not a proxy.

[2] https://datatracker.ietf.org/doc/html/rfc9110#field.connection

13.2 WebSocket Protocol

Frame Format

The rest of the connection after the hanshake consists of a series of *frames*.

```
|     byte0     |     byte1     |     byte2     |     byte3     |
 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               | Masking-key, if MASK set to 1 |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |        Payload Data ...       |
+---------------------------------------------------------------+
```

Note: The 0th bit is the most significant bit (MSB) in a byte.

The structure is header + payload like HTTP. But unlike HTTP …

- The format is binary, no delimiters are used.
- The payload length is trivially indicated in the header.

Handling WebSocket frames is much easier than handling HTTP.

1. The first byte contains the FIN flag and a 4-bit opcode.
2. The MASK bit in the second byte indicates the optional masking-key.
3. The next field is the payload length, which is encoded as a *variable-length* integer starting with a 7-bit integer in the second byte.
    - The length from 0 to 125 is stored in that 7-bit integer.
    - 126 indicates a following 16-bit big-endian integer.
    - 127 indicates a following 64-bit big-endian integer.
4. If the MASK bit is set, a 4-byte random data mask follows.

The 4-byte random data mask is XORed with the payload data. The mask must be used for client-to-server frames, but not for the other way around. The purpose of the XOR mask is to prevent a type of cache poisoning attack[3].

[3] http://www.adambarth.com/papers/2011/huang-chen-barth-rescorla-jackson.pdf

Types of Frames

Opcodes for data message:

- 0x01: A text data message. For applications.
- 0x02: A binary data message. For applications.
- 0x00: Message fragmentation. Discussed later.

The distinction between text and binary is up to the application.

Opcodes for control messages:

- 0x08: A control message for graceful termination. The optional payload contains a status code and a string message.
- 0x09 and 0x10: Ping and pong, or heartbeats. For keeping the connection alive and probing dead connections. Can be transparent to applications.

Fragmented Messages

WebSocket also supports sending messages without knowing their length in advance, like the chunked transfer encoding in HTTP. A "logical" data message can be broken into multiple frames, but the application sees the combined message.

- The first frame has opcode 0x01 or 0x02, while the rest frames use 0x00.
- The last frame has the FIN flag set.
- Only data messages (opcode 0x01 and 0x02) can be fragmented.
- Frames of different messages cannot be interleaved, except for control messages.

An example with 3 fragments:

```
| opcode=0x01, FIN=0 | opcode=0x00, FIN=0 | opcode=0x00, FIN=1 |
```

An unfragmented message is encoded as a single frame with opcode 0x01 or 0x02 and the FIN flag is set.

Like the design of chunked transfer encoding, you can stream arbitrarily large data as a single message. But many implementations represent the message data as a single buffer, so there is a limit to the message size.

13.3 Introduction to Concurrent Programming

Race Conditions

In message-based designs, it's common to have multiple sources of messages consumed by a single consumer, such as multiple concurrent tasks sending messages over a single WebSocket. Since WebSocket messages are serialized into a single TCP byte stream, concurrent writes to the socket will mess up the protocol.

Let's use a hypothetical example to illustrate the problem.

```
// pseudo code!
async function send(conn, msg) {
    await write_header(conn, msg);
    await write_body(conn, msg);
}
```

In this example, writing 1 message involves 2 socket writes, and the code yields to the runtime at each `await` statement. The problem is that while back to the runtime, the runtime may schedule another task that is also sending a message, resulting in *interleaved* socket writes.

This is called a *race condition* in concurrent programming. Note that this has *nothing* to do with the single-threaded runtime or the number of CPU cores.

Atomic Operations

You may think that the solution is to use only 1 socket write for each message. However, this solution is …

- Correct only if socket writes are atomic, which may not be guaranteed.
- Not always possible and not convenient. Consider that WebSocket messages can be fragmented and used like streams.

Although operating systems try[4] to[5] implement concurrent writes as atomic operations, no sane application will depend on this behavior, and a byte stream is not about messages anyway.

[4] https://man7.org/linux/man-pages/man2/write.2.html#BUGS

[5] http://web.archive.org/web/20120623124411/http://www.almaden.ibm.com/cs/people/marksmith/sendmsg.html

Mutual Exclusion with Mutexes

One solution is to use a lock (mutex) to limit concurrent writes to the socket.

```
// pseudo code!
async function send(mutex, msg) {
    await mutex.lock();
    try {
        // write to the socket
    } finally {
        mutex.unlock();
    }
}
```

A mutex is "mutual exclusion" because it disallows more than one task from entering the locked state. In the case of concurrent access, one task holds the lock and the rest block on the `await` statement, and releasing the lock will unblock one of the waiters. The mutex is one of the *synchronization primitives* in many concurrent programming environments.

Multiplexing with Queues

Another solution to preventing concurrent access is to use a queue. The queue accepts WebSocket messages from concurrent producers, and a dedicated task consumes from the queue and writes to the socket. This is what we will do.

```
// pseudo code!
let queue = createQueue();

// for multiple producers
async function produce(msg) {
    await queue.pushBack(msg);
}

async function single_consumer() {
    while (running()) {
        const msg = await queue.popFront();
        // write to the socket
    }
}
```

A mutex is also a form of queue that doesn't pass data, it passes control instead.

Flow Control with Blocking Queues

A JS array is a queue that you can push and pop. But it is not useful for concurrent programming. How do you, as a consumer, know when the queue is not empty? There needs to be a mechanism that allows consumers to wait for producers.

This mechanism can be implemented in the queue! That's why the above pseudo code uses **`await`** to consume from the queue. This is called a *blocking queue* because consumers are blocked when the queue is empty.

Also, backpressure requires that the queue be bounded in capacity. This is achieved by having the producer block when the queue is full. In contrast to the Stream API in Node.JS, this removes the footgun of not mandating backpressure.

```
// both producers and consumers can block.
type Queue<T> = {
    pushBack(item: T): Promise<void>;
    popFront(): Promise<T>;
};
```

A blocking queue not only passes data between producers and consumers, it also passes *control*, which is similar to a mutex. In fact, while passing data is convenient, passing control is more fundamental in concurrent programming.

Cancellation with Closeable Queues

A blocking queue is similar to a Unix pipe in terms of blocking behavior. Except for one thing: a pipe can be *closed*!

- If closed by the producer, the consumer gets EOF.
- If closed by the consumer, the producer gets errors when writing to it.

In concurrent programming, sometimes you need to make tasks quit their jobs. For example, if a WebSocket is closed while the application task is blocking on it (either reading or writing), the task should be woken up (to get an EOF or an error) instead of hanging forever.

If the task is a consumer of a queue, you can put special values into the queue to notify consumers to quit. But for producers waiting on a queue, this is not so easy. Fortunately, we can learn from Unix pipes — add a **`close()`** method to unblock all producers and consumers.

```
type Queue<T> = {
    pushBack(item: T): Promise<void>;   // throws if closed
    popFront(): Promise<null|T>;        // returns null if closed
    close(): void;
};
```

We can throw exceptions to producers and return nulls to consumers, mimicking closed pipes.

The Blocking Queue as a Synchronization Primitive

A synchronization primitive is something used to block and unblock tasks (pass control). Some traditional synchronization primitives include:

- Mutex.
- Semaphore.
- Condition variable.
- Event.

In addition to traditional synchronization primitives, Golang also employs the "channel" as its primary synchronization primitive, which is mostly similar to the blocking queue we're discussing.

Many simple concurrency problems are solved by passing data, which is favored in Go. We will demonstrate this with our WebSocket implementation.

13.4 Coding a Blocking Queue

Node.JS comes with none of these synchronization primitives, and concurrency problems are often masked by mountains of callbacks. Fortunately, with the addition of `async/await`, these primitives can be copied to JS. But we have to first create them using promises and callbacks.

Step 1: Analyze the Problem

Let's consider a blocking queue with just push and pop, and no buffering capacity. The queue should be usable with multiple producers and multiple consumers, which is more versatile than generators.

```
type Queue<T> = {
    pushBack(item: T): Promise<void>;
    popFront(): Promise<T>;
};
```

Creating a promise results in the `resolve` callback that is used to fulfill it later. A producer should either wait for a consumer or wake up a waiting consumer.

- If no consumer is waiting, it stores a callback somewhere. A future consumer will invoke the callback to take the data and fulfill the producer.
- If there are waiting consumers, fulfill one of them by invoking its callback.

The situation for a consumer is similar:

- It should either store a callback to wait for a producer,
- or invoke the callback of a waiting producer.

Step 2: Push and Pop

There can be multiple waiting producers or consumers, so we need to store their callbacks in a list.

For consumers, just store their own `resolve` callback to receive the data.

For producers, the callback does 2 things:

1. Take the consumer's callback to fulfill it (with the data).
2. Fulfill its own promise.

```
// a multi-producer, multi-consumer, and 0-capacity queue.
function createQueue<T>(): Queue<T> {
    type Taker = (item: T) => void;         // fulfill a consumer
    type Giver = (take: Taker) => void;     // wake up a producer
    const producers: Giver[] = [];
    const consumers: Taker[] = [];
    return {
        pushBack: (item: T): Promise<void> => {
            return new Promise<void>((done: () => void) => {
                const give: Giver = (take: Taker) => {
                    take(item);
                    done();
                };
                if (consumers.length) { // consumers are waiting
                    give(consumers.shift()!);
                } else {                // wait for a producer
                    producers.push(give);
                }
            });
        },
        popFront: (): Promise<T> => {
            return new Promise<T>((take: Taker) => {
                if (producers.length) { // producers are waiting
                    producers.shift()!(take);
                } else {                // wait for a consumer
                    consumers.push(take);
                }
            });
        },
    };
}
```

Step 3: Close the Queue

The **close()** method will come in handy later.

```
type Queue<T> = {
    pushBack(item: T): Promise<void>;   // throws if closed
    popFront(): Promise<null|T>;        // returns null if closed
    close(): void;
};
```

Since we may throw exceptions at producers, the **reject** callback is also stored.

```
    type Rejector = (err: Error) => void;
    const producers: {give: Giver, reject: Rejector}[] = [];
    let closed = false;
```

We will also remember the closed state. This state should be checked before pushing or poping. And closing a queue unblocks everyone.

```
        close: () => {
            // unblock any waiting producers or consumers
            closed = true;
            while (producers.length) {
                producers.shift()!.reject(new Error('queue closed.'));
            }
            while (consumers.length) {
                consumers.shift()!(null);
            }
        },
```

Step 4: Add a Buffer

The queue is sufficient for our purposes. For some other use cases, the queue may need to buffer a bounded amount of items without blocking producers. This is left to you as an exercise.

```
function createQueue<T>(capacity: number): Queue<T>;
```

13.5 WebSocket Server

Step 1: Design the Message Interface

WebSocket data messages are represented to applications as the `WSMsg` type. The payload data is consumed using the `read()` function like the `BodyReader` interface to support arbitrarily large messages and receiving data on the fly.

```
const WS_DATA_TEXT = 0x01;
const WS_DATA_BINARY = 0x02;

// WebSocket message type
type WSMsg = {
    type: number,       // WS_DATA_TEXT, WS_DATA_BINARY
    length: number,     // in bytes; -1 if fragmented
    read: () => Promise<Buffer>,    // empty after EOF
};
```

While the **read()** function does not require the whole message to be stored in memory, it does have a downside: the application must ensure that the message data is consumed, otherwise the next frame won't be parsed.

Step 2: Design the Server Application Interface

The **WSServer** is the application interface for sending and receiving data messages.

```
// The WebSocket API
type WSServer = {
    send(msg: WSMsg): Promise<void>,    // supports multiple producers
    recv(): Promise<null|WSMsg>,        // null after EOF
    close(): void,
};
```

This is similar to what we do with a socket — read from it and write to it. Except that concurrent reads and writes have valid use cases and must be supported.

- Concurrent writes: Multiple app tasks generate messages independently.
- Concurrent reads: Concurrent message processing with a task pool.

You can also use callbacks to deliver messages instead of actively reading them. But this makes the backpressure less obvious, because you'll also need the **pause()** and **resume()** methods. This is why we ditched callback-based IO early on.

Step 3: Design the Server Tasks

Unlike request-response, sending and receiving are independent. So we need:

- A task to parse the incoming byte stream: **wsServerRecv()**.
- A task to write messages to the socket: **wsServerSend()**.

These 2 tasks run concurrently with the application tasks. Two queues are used to connect the 2 tasks to the application (send and receive). Another queue is used to connect **wsServerSend()** to the socket.

```
App <= qrecv (Queue) <= wsServerRecv <= reqBody (BodyReader)
App => qsend (Queue) => wsServerSend => qsock (Queue) => resBody (BodyReader)
```

There can be multiple app tasks concurrently producing or consuming with a single WebSocket, but the tasks for reading and writing to the socket must be sequential. This is an example of solving concurrency problems by passing data.

Step 4: Start Server Tasks

Let's code up the previous step.

```
function createWSServer(reqBody: BodyReader): [WSServer, BodyReader] {
    // WS API
    const qrecv: Queue<WSMsg> = createQueue<WSMsg>();
    const qsend: Queue<WSMsg> = createQueue<WSMsg>();
    const ws: WSServer = {
        send: qsend.pushBack,   // throws if closed
        recv: qrecv.popFront,   // returns null if closed
        close: (): void => {
            qsend.close();      // generates a WS_CTRL_CLOSE
            qrecv.close();
        },
    };
    // task 1: reading from the socket
    wsServerRecv(reqBody, qrecv, ws)
        .finally(ws.close)      // closes the WS API
        .catch(console.error);  // no await
    // task 2: writing to the socket
    const qsock: Queue<Buffer> = createQueue<Buffer>();
    wsServerSend(qsend, qsock)
        .finally(ws.close)      // closes the WS API
        .finally(qsock.close)   // closes the socket
        .catch(console.error);  // no await
    const resBody: BodyReader = {
        length: -1,
        read: async () => await qsock.popFront() || Buffer.from(''),
    };
    return [ws, resBody];
}
```

The `finally()` callback registered on the promise behaves like a try-finally block. This is used to close queues so that tasks won't hang in a queue forever.

Note: If you do not `await` an `async` function, it's vital to catch exceptions with a `catch()` callback, otherwise your program will crash. In this case, we simply log and ignore the exception, but sometimes you need to propagate errors.

Step 5: Write Messages to the Socket

The `wsServerSend()` task takes `WSMsg` input from a queue and outputs data to the `qsock` queue, which is pulled by a `BodyReader`. The frame code is omitted.

```
// format WS messages and send them to the socket
async function wsServerSend(
    qsend: Queue<WSMsg>, qsock: Queue<Buffer>): Promise<void>
{
    while (true) {
        const msg = await qsend.popFront();
        if (!msg) { // close it
            // omitted. send a "close" frame ...
            break;
        }
        // omitted. write frame data to `qsock` ...
    }
}
```

Note how similar this is to using Unix pipes:

- A unified interface for input and output.
- Composable.

Step 6: Parse Frames From the Socket

`wsServerRecv()` loops for each frame and feeds `WSMsg`s into the output queue.

```
// parse WS frames from the client
async function wsServerRecv(
    reqBody: BodyReader, qrecv: Queue<WSMsg>, ws: WSServer): Promise<void>;
```

We'll use another queue to send frame data to the `read()` function of a `WSMsg`.

```
                // create a WSMsg
                const q = data = createQueue<Buffer>();
                const msg: WSMsg = {
                    // omitted ...
                    read: async () => await q.popFront() || Buffer.from(''),
                };
                await qrecv.pushBack(msg);
                // omitted. feed payload data into `q` ...
```

A data message can span multiple frames, so we need to store the queue outside the loop and feed the queue as more frames arrive. Remember to make sure the queue is closed, either when the FIN flag is set or when exiting the loop.

```
    // provide data to the current message
    let data: null|Queue<Buffer> = null;
    try {
        // loop for each frame
        while (true) {
            // omitted ...
        }
    } finally {
        // close the message data in case the app is blocking on it.
        data?.close();
    }
```

Step 7: Integrate with the HTTP Server

A request is handled either as a WebSocket or as an HTTP request. This is decided in `getWSApp()`, which returns an application logic function or `null`.

```
type WSApplication = (ws: WSServer) => Promise<void>;

// map the request to the WS app.
function getWSApp(req: HTTPReq): null|WSApplication
```

WebSockets are handled differently:

- The request is handled by `handleWS()`, which calls the application function.
- The rest of the socket data is passed to the protocol without further processing by the `readerFromConnEOF()` function.
- The response data is written directly to the socket. So don't mess it up with response compression, extra header fields, and chunked encoding.

```
        // process the message and send the response
        let reqBody: BodyReader;
        let res: HTTPRes;
        const wsapp: null|WSApplication = getWSApp(msg);
        if (wsapp) {    // upgrade to WebSocket
            reqBody = readerFromConnEOF(conn, buf);
            res = await handleWS(msg, reqBody, wsapp);
        } else {        // normal HTTP connection
            reqBody = readerFromReq(conn, buf, msg);
            res = await handleReq(msg, reqBody);
        }
        try {
            if (!wsapp) {
                enableCompression(msg, res);
            }
            await writeHTTPHeader(conn, res);
            if (msg.method !== 'HEAD') {    // omit the body
                await writeHTTPBody(conn, res.body, !!wsapp);
            }
        } finally {
            await res.body.close?.();       // cleanups
        }
        // close the connection for HTTP/1.0 or WebSocket
        if (msg.version === '1.0' || wsapp) { return; }
```

The `handleWS()` function puts it all together.

- Launch tasks to handle the protocol.
- Launch the application task.
- Return the `Upgrade` header.

```
async function handleWS(
    req: HTTPReq, reqBody: BodyReader, app: WSApplication): Promise<HTTPRes>
{
    // handle the WS protocol
    const [ws, resBody]: [WSServer, BodyReader] = createWSServer(reqBody);
    // launch the WS appliction
    app(ws).finally(ws.close).catch(console.error);     // no await
    // the upgrade response header
    const key: Buffer = fieldGet(req.headers, 'Sec-WebSocket-Key')!;
    return {
        code: 101,  // Switching Protocols
        headers: [
            Buffer.from('Upgrade: websocket'),
            Buffer.from('Connection: Upgrade'),
            Buffer.from(`Sec-WebSocket-Accept: ${wsKeyAccept(key)}`),
        ],
        body: resBody,
    };
}
```

Step 8: Test and Debug

Use a web browser to create WebSockets for testing. Remember to use packet capturing tools. Wireshark can disassemble the protocol, so you can use it to …

- Learn what the correct format looks like if you failed to parse the frames.
- See what's wrong with the frames you've generated.

13.6 Discussion: WebSocket in the Browser

WebSocket is symmetric with respect to the functionality of both peers. So the application interface should be usable for both clients and servers. Let's explore the WebSocket API in the browser and compare it to ours.

No Backpressure for WebSocket in the Browser

The most important difference is that the browser API is callback-based. The browser invokes the event handler[6] as messages arrive. This is similar to `net.Socket` in Node.JS, except that there are no `pause()` and `resume()` methods, so backpressure is impossible for receiving.

For sending messages, the `send()` method[7] just buffers the data and returns nothing. There is an attribute `bufferedAmount` to tell the buffer size, so applications can still control backpressure.

There is an alternative API design[8] that uses `async/await` for reading and writing, which is the direction we took.

[6] https://developer.mozilla.org/en-US/docs/Web/API/WebSocket/message_event
[7] https://developer.mozilla.org/en-US/docs/Web/API/WebSocket/send
[8] https://developer.chrome.com/docs/capabilities/web-apis/websocketstream

Application Level Backpressure

Missing backpressure is probably the No. 2 mistake in network programming. But even if backpressure is impossible in the browser, applications can implement it at a higher level:

- The consumer acknowledges processed messages to the producer.
- The producer uses the acknowledgements to implement flow control.

There are messaging protocols that run on top of WebSocket that implement backpressure, such as RSocket[9].

Message Size is Limited

While you can treat a WebSocket message as an unlimited stream by using fragmented frames, most software, including browsers, stores the message as a single buffer. So there is a limit to both message size and frame size. (We don't have these limits because we use the `read()` function to consume data on the fly).

Applications that send large chunks of data over WebSockets should have a fragmentation mechanism to keep the message size small.

13.7 Conclusion: What We Have Learned

Important topics since our basic HTTP server:

- HTTP semantics:
  - `Content-Length` and chunked transfer encoding.
  - Range requests.
  - Caching.
  - Compression.
- Manual resource management based on ownership.
- Techniques for using buffers.
- Abstractions for producer-consumer problems and backpressure:
  - Generators.
  - Streams.
  - Blocking queues.

[9] https://rsocket.io/about/protocol#flow-control

Table 1: Solutions to producer-consumer problems.

	Blocking Queue	Stream API	Async generator
Producer	`await q.pushBack(x)`	`this.push(x)`	`yield x;`
Consumer	`await q.popFront()`	`'data'` event	`await g.next()`
Multi-producer	Yes	Yes	No
Backpressure	Auto	Manual	Auto